International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-1 Number-1 Issue-1 September 2011

Object Oriented Mutation Applied in Java Application programming Interface and C++ Classes

Titu Singh Arora[1], Ravindra Gupta[2]
M.Tech Scholar, Department of Computer Science, SSSIST Sehore, India[1]
Professor in CSE Department at SSSIST Sehore[2]

Abstract

Mutation analysis is a powerful and computationally expensive technique that measures the effectiveness of test cases for revealing faults. The principal expense of mutation analysis is that many faulty versions of the program under test, culled mutants, must be repeatedly executed. We survey several aspects of reconstruction of complex object-oriented faults on the java API. Application of object-oriented mutation operators in java programs using a parser-based tool can be precise but requires compilation of mutants. In this paper we approach the mutation on Object Oriented features to test the functionality. For this we consider java and C++ programs.

Keywords

Java, Java API, C++, Mutation, Object Oriented Functionality

1. Introduction

Mutation testing is considered one of the promising testing techniques [1]. In the mutation testing, small syntactic modifications are introduced into a program P. A set of similar programs called "mutants" is obtained after applying a single mutation operator to a single location in P. These mutants are run with an input data from a given test set. If for a test case the output of program P differs from that of mutant P', this test is said to "kill" mutant P'. The mutants that generate the same output for any test case are called "equivalent" mutants. Typically the equivalent mutants are distinguished approximately after testing or identified by hand. "Mutation score", the adequacy of a test set, is measured as a ratio of the number of mutants killed over the total number of non-equivalent mutants.

Software testing involves exercising a program on a set of test case input values and comparing the actual output results with expected ones [2]. Since exhaustive testing is usually not tractable, test strategies are faced with a problematic task that is: how to select a minimum set of test cases that is sufficiently effective for revealing potential faults in a program? An evaluation criterion for test strategies is to measure the effectiveness of generated test cases. Mutation analysis [3] is an evaluation technique that assesses the quality of test cases by examining whether they can reveal certain types of faults.

In object oriented paradigm, research is mainly concerned with analysis, design and programming techniques. Software testing could not get much attention of researcher for object oriented paradigm. These newly introduced features need some way to verify their correctness. Traditional standard testing techniques are inadequate for object oriented systems. Mutation testing is basically used to measure the accuracy of test suite, to assess the effectiveness of testing technique and to compare them is also called, mutation analysis [4].

Mutation testing is time consuming, complex and manually impractical but it is more powerful than statement coverage, branch coverage and data flow testing in finding faults [5]. Cost of mutation is usually assessed in terms of number of mutants which depends on number of mutation operators. This problem can be solved by finding a smaller subset of mutation operators that have equal effectiveness as the full set retains.

The key idea that makes mutation analysis feasible is that the set of competent programs can be approximated by making small changes to the given program under test. Such changes typically include the replacement of a program variable with some other variable or the replacement of an arithmetic or relational operator by some other compatible operator. The resulting programs are known as mutants of the given program and the modification rules are known as mutation operators.

The remaining of this paper is organized as follows. We discuss about java in Section 2. In Section 3 we discuss about Mutation for Java API. In section 4 we discuss about Failure Reasons. In section 5 we discuss about the recent scenario. The conclusions are given in Section 6. Finally references are given.

2. Java

In 1991, a group of Sun Microsystems engineers led by James Gosling decided to develop a language for consumer devices (cable boxes, etc.). They wanted the language to be small and use efficient code since

these devices do not have powerful CPUs. They also wanted the language to be hardware independent since different manufacturers would use different CPUs. The project was code-named Green. These conditions led them to decide to compile the code to an intermediate machine-like code for an imaginary CPU called a virtual machine. (Actually, there is a real CPU that implements this virtual CPU now.) This intermediate code (called byte code) is completely hardware independent. Programs are run by an interpreter that converts the byte code to the appropriate native machine code.

Thus, once the interpreter has been ported to a computer, it can run any byte coded program. Sun uses UNIX for their computers, so the developers based their new language on C++. They picked C++ and not C because they wanted the language to be object-oriented. The original name of the language was Oak. However, they soon discovered that there was already a programming language called Oak, so they changed the name to Java.

The Green project had a lot of trouble getting others interested in Java for smart devices. It was not until they decided to shift gears and market Java as a language for web applications that interest in Java took off. Many of the advantages that Java has for smart devices are even bigger advantages on the web. Currently, there are two versions of Java. The original version of Java is 1.0.

At the heart of Java technology lies the Java virtual machine--the abstract computer on which all Java programs run. Although the name "Java" is generally used to refer to the Java programming language, there is more to Java than the language. The Java virtual machine, Java API, and Java class file work together with the language to make Java programs run.

Java's architecture arises out of four distinct but interrelated technologies:

- the Java programming language
- the Java class file format
- the Java Application Programming Interface
- the Java virtual machine

When you write and run a Java program, you are tapping the power of these four technologies. You express the program in source files written in the Java programming language, compile the source to Java class files, and run the class files on a Java virtual machine. When you write your program, you access system resources (such as I/O, for example) by calling methods in the classes that implement the Java Application Programming Interface, or Java API. As your program runs, it fulfills your program's Java API calls by invoking methods in class files that implement the Java API. You can see the relationship between these four parts in Figure 1.

Together, the Java virtual machine and Java API form a "platform" for which all Java programs are compiled.

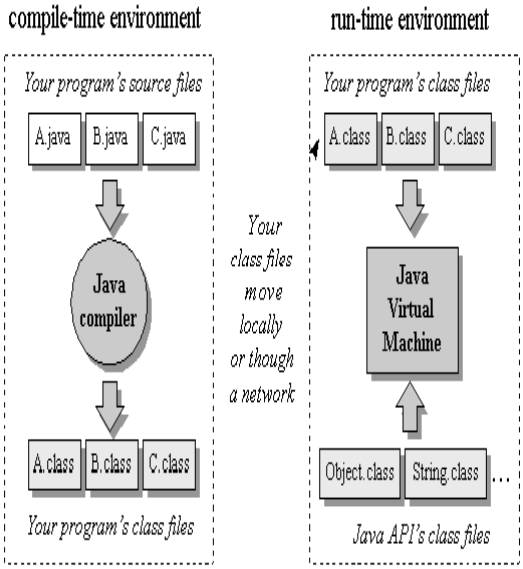

Figure 1 Java Programming Environment

3. Mutation for Java API

To estimate the number of fish of a certain species in a lake, one way to do it is letting some marked fish out in the lake (say, 20) and then catches some fish and count the marked ones. If we catch 40 fish and 4 of them are marked, then 1 out of 10 is marked and the population in the entire lake could be estimated to about 200. If we catch all marked fish, we would as a side-effect end up with almost the entire population in our nets.

Fault-based testing does something similar. We let some "marked" bugs loose in the code and try to catch them. If we catch them all, our "net "probably caught many of the other, fishier, fish. The unknown bugs, that is one of the fault-based testing strategies is mutation testing. There are many variations of mutation testing such as weak mutation, interface mutation and specification-based mutation testing .

The method described in this thesis is strong mutation testing, but the idea is the same for all of them, namely to "mutate" the original program under test.

To mutate a program, an error is put somewhere in the code. And just like the fish in the lake, we will try

to catch it. A typical mutation would be to replace < with > in one and only one expression.

Example: **theprogram P =**

1. if (x > 0)
2. doThis();
3. if (x > 10)
4. doThat();

A mutation of P would be (line 1)

1. if (x < 0)
2. doThis();
3. if (x > 10)
4. doThat();

Another mutation (line 3):

1. if (x > 0)
2. doThis();
3. if (x < 10)
4. doThat();

Now we have made several copies of P and introduced a single mutation into each copy. These copies are called mutants. Let D denote the input domain. Assume we have a passing test set, T ⊂ D, that is P satisfies or passes every test in T . To get a measure of its mutation adequacy, we run the test set against each mutation and count the number of mutants for which T fails.

If T fails for a certain mutant, we call that mutant killed. The idea is that if T detects this fault (kills the mutant), it will detect real, unknown faults as well. If T kills all mutants, it potentially detects many unknown faults. Mutants that are not killed are called alive and mutants (denoted mu) such that ∀x ∈ D, P.x = μ.x are called equivalent. We will write μ ≡ P if the mutant μ is equivalent to P . P.x represents the evaluation of the program P on the input x. Mutation adequacy or mutation score is defined as (number of killed mutations)/(total number of non-equivalent mutations) * 100 %. Why would this method work? makes two fundamental assumptions; (a) the competent programmer hypothesis and (b) the coupling effect.

The traditional approach to software testing is to find some subset T (called the test set) of the input domain D, such that

∀x ∈ T, P.x = f(x) → ∀x ∈ D, P.x = f(x),
where f is a functional specification of the program P . (This is called a reliable test set.) To be able to reach this conclusion, some exhaustive testing strategy would be necessary. This is too strong a conclusion and is proven to be an undecidable problem. That is why mutation testing weakens the above:
either P is "pathological" or
∀x ∈ T, P.x = f(x) → ∀x ∈ D, P.x = f(x)
"pathological" program
P is "pathological" ↔ P /∈ Φ,
where Φ is the set of programs in a "neighbourhood" of a correct program. We expect programmers to be competent enough to produce programs in this neighbourhood. We can now reformulate

∀x ∈ T, P.x = f(x) ∧
∀Q ∈ Φ (Q ≡ P ∨ ∃x ∈ T, Q.x =6 P.x)
→ ∀x ∈ D, P.x = f(x),

4. Failure Reasons

To visualize failure regions, we define D = (x, y, z), where x and y are integers in the interval [1, 10] and z = 5, execute two mutants of the TRIANGLE program (see the appendix) and compare the output with the original, unmutated program P.

We see surface plots of the different function

$$\Delta_{P,P_1}(x) = \begin{cases} 1, & if\ P.x \neq P_1.x \\ 0, & otherwise \end{cases},$$

where P is the program under test and P1 -> NULL. The semantic size, would then be

$$s(P, P_1) = \frac{\sum_{x \in D} \Delta_{P,P_1}(x)}{c(D)},$$

where c(D) is the cardinality of D. The failure region F is the set

$$F_{P,P_1} = \{x \mid \Delta_{P,P_1}(x) = 1\}$$

The coupling function _ for the two faulty versions P1 and P2 of P:

$$\delta_P^{P_1,P_2} = \begin{cases} 1, & if\ x \in F_{P,P_1}\ and\ x \in F_{P,P_2} \\ 0, & otherwise \end{cases}$$

Finally, we will define the coupling effect ratio of P1 with respect to P2:

$$CeR_P(P_1, P_2) = \frac{s(\delta_P^{P_1,P_2})}{s(\Delta_{P,P_1})}$$

In words, we could express this as an estimate of the probability of "a test point detecting P1 also detecting P2". Normally, we will not have the luxury to see the entire input domain at once.

The most important functionality of the program would of course be to create mutants. This section explains how to do that. The problem is reduced to mutate individual program elements, since a mutant normally differs from the program under test in one program element only.
Consider this statement in the program under test:

```
z = x + y;
```

How do we mutate this statement? One approach is to create a metamutant. A meta mutant is one program containing all mutants. To declare which mutant is executing, an environment variable is set.

The metamutant version of the above statement could be something like

```
z = plusIntInt(x, y, 230, 232);
```

Each binary expression eligible for mutation is replaced with a function similar to the one above. The automatically generated plusIntInt function

```
plusIntInt(int x, int y, int firstMut, int lastMut)
{
if (getCurrentMutation() >= firstmut &&
getCurrentMutation() <= lastmut)
{
if (getCurrentMutation() == firstmut)
return x - y;
if (getCurrentMutation() == firstmut + 1)
return x * y;
if (getCurrentMutation() == firstmut + 2)
return x / y;
return x + y;
}
else
return x + y;
}
```

Detecting equivalent mutants requires a constraint solver. Constraints are kept track of just like any scoped variable. Consider this code snippet:

```
1. public someFunc(int x, int y)
2. if (y == 0)
{
...
3. z = plusIntInt(x, y, 230, 232);
...
}
```

A type checker knows that in line 3, the variables x and y are available. With not too much effort we can teach the type checker to handle constraints so that it also know that in the entire code block after line 2, the constraint y = 0 holds (unless y is modified, of course).

5. Recent Scenario

In 2003, Anna Dereziska [6] states that the quality of a test suite can be measured using mutation analysis. Groups of OO mutation operators are proposed for testing object-oriented features. The OO operators applied to UML specification and C++ code are illustrated by various examples.

In 2010, Zaheed Ahmed et al. [7] survey some of the traditional mutation operators which are incorporated in mutation testing of object oriented systems. Recently class level mutation operators are also defined; they focus with particular consideration of the OO programming (OOP) language JAVA. A number of automated tools have been developed to generate the defective versions of program and to execute them against test suit. Classification, evaluation of the mutation operators against some proposed parameters and identification of some research areas is a result of this survey.

In 2011, Stefan Endrikat et al. [8] describe an empirical, socio-technical study with Java and AspectJ where developers needed to perform changes on their code base multiple times. It shows that frequent changes in the crosscutting code which do not change the concern's underlying structure compensate an initial higher development time for those concerns.

In 2011, Anna Dereziska et al. [9] proposed reconstruction of complex object-oriented faults on the intermediate language level. The approach was tested in the ILMutator tool implementing few object-oriented mutation operators in the intermediate code derived from compiled C# programs. Exemplary mutation and performance results are given and compared to results of the parser-based mutation tool CREAM.

6. Conclusion

This paper presents discuss several concepts and on how to Classes in object-oriented systems, written in different programming languages, contain identifiers and comments which reflect concepts from the domain of the software system. This information can be used to measure the cohesion of software. The above phenomena show the need of mutation.

We survey several aspects of reconstruction of complex object-oriented faults on the java API. Application of object-oriented mutation operators in java programs using a parser-based tool can be precise but requires compilation of mutants. In this paper we approach the mutation on Object Oriented features to test the functionality. For this we consider java and C++ programs.

References

[1] J.M. Voas, G. McGraw, Software fault injection,Inoculating Programs Against Errors, J. Wiley & Sons, 1998.
[2] B. Beizer. Software Testing Techniques. Van Nostrand Reinhold,New York, USA, Seconde Edition, 1990.

[3] R. DeMillo, R. Lipton, and E Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. Computer, 11(4):34-41, Apr. 1978.
[4] Namin, A. S., and Andrews, J. 2007. On Sufficiency of Mutants, 29th International Conference on Software Engineering (ICSE'07 Companion).
[5] Namin, A. S., and Andrews, J. 2007. On Sufficiency of Mutants,29th International Conference on Software Engineering (ICSE'07 Companion).
[6] Anna Dereziska," Object-Oriented Mutation to Asses the Quality of Tests", 2003 IEEE.
[7] Zaheed Ahmed, Muhammad Zahoor and Irfan Younas, "Mutation Operators for Object-Oriented Systems: A Survey", 2010, IEEE.
[8] Stefan Endrikat , Stefan Hanenberg, "Is Aspect-Oriented Programming a Rewarding Investment into Future Code Changes? ASocio-Technical Study on Development and Maintenance Time", 2011 19th IEEE International Conference on Program Comprehension.
[9] Anna Dereziska, Karol Kowalski," Object-oriented Mutation Applied in Common Intermediate Language Programs Originated from C# ", 2011 Fourth International Conference on Software Testing, Verification and Validation Workshops.