Journal homepage: http://www.journalijar.com

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH

RESEARCH ARTICLE

An Empirical based Mutation Testing through Effective Test Data

.Dr.Srinivas Prasad,
Professor, Dept. of CSE GMR Institute of Technology, Andhra Pradesh

Manuscript Info

Manuscript History:

Received: 18 March 2015
Final Accepted: 22 April 2015
Published Online: May 2015

Key words:

mutation testing, mutation operators, mutant, object- oriented programs, test automation, test coverage

****Corresponding Author***

Dr.Srinivas Prasad

Abstract

Mutation testing is a powerful but complicated and computationallyexpensive testing method. Testing of object-oriented software has a number of features that make it different from conventional software testing. Traditional testing methods are not very effective in testing object oriented software. The factors like inheritance, encapsulation, polymorphism, which are the specialized phenomena of object- oriented software bring in the point of different Mutation testing is a fault based testing technique that has the advantage of easy automation, and can check the thoroughness of testing performed using other testing approaches. Mutation operators are an important consideration while designing the mutation testing methodologies. In this paper, we designed set of new mutation operators.Our experimentation reveals that our approach is able to detect faults and is not detected by traditional code-based testing approaches.

Copy Right, IJAR, 2015,. All rights reserved

INTRODUCTION

Testing object oriented software has exclusive testing issues.ManyResearchwork having sited that traditional testing methods are least effective in object oriented software testing. Object oriented features such as inheritance, polymorphism, dynamic binding and object state are key attributes in objectorientedtesting are challenges. Object oriented mutation testing can indicate how “good” a given suite is. It is very difficult to quantitatively measure the effectiveness of a designed test suite in detectingbugs. This led researchers to the concept of mutation testing. Mutation testing [4, 1] is a fault-based technique that helps a tester to increase the effectiveness of his test suite. Design test cases that can detect it let the tester to design additional test cases to detect certain faults that were not detected by the originally designed test suite.

Mutation testing is an effective way to detect and fix programming errors though it is not very effective in detecting and fixing algorithmic or logic errors. However, often critical and difficult to detect errors in a program occur due to simple programming errors.

In mutation testing terminology simple syntactic changes to a program are called mutations. Mutation operators are applied to a program to create mutated program. For example, a simple mutation operator can be replacing a “<” operator with a “<=” operator. Mutations can be used to test how effective a test suite is. Mutation testing introduces faults into software by creating many different versions of the program. Each version differ from the other in a very minor way (which can introduce a fault) compared to the actual implementation. The idea is to see if the test cases that were written can detect the new fault. The concept of mutation testing can be explained as follows: - If we run a mutant against the test suite, we will have two possible scenarios

1. The test suite detects the error introduced by the code change. In this case, the mutant is said to be a killed mutant.

2. The test suite does not detect any mutation i.e. the program output does not change. Such a mutant is called an equivalent mutant

The ratio of killed mutants to the total mutants created measures how sensitive a program is to code changes and how effectives is the test suite .
The effectiveness of mutation testing depends heavily on the types of faults that the mutation operators are designed to represent. Mutation operators represent how a program would be changed to get a mutated program. Some types of operators can cause more severe errors than the others. For example, a simple change to a condition expression in the following code may cause serious errors:

1. Original Code: if (x==1) {….}

2. Mutated Code: if (x<=1) {….}

In the above example the „equality condition" has been changed to "less than equal to" by the mutation operator.
How effective mutation testing is, depends on the type of faults the mutation system can bring in. The classical works on mutation testing were done in the context of procedural programs. These results therefore cannot straight way be applied to object oriented programs. The reason being that the errors that can be introduced due to specific to object orientationfeaturessuch as inheritance, polymorphism, and so on that the current mutation systems fail to adequately handle. To over come this problem, a method called class mutation has been developed to target faults that are likely to arise from features specific to object oriented programming. The features specific to object orientation that are normally considered by researchers [2] are classdeclarations, references, single inheritance,information hiding, and polymorphism [2]. In cases where the number of possible faults for a givenprogramis large, mutation testing uses two principles to restrict the total no of mutated programs that arecreated:the competent programmer hypothesis[3] and the coupling effect[4].
Test coverage analysis is a structure testing technique. It finds codes not executed by a suite of test cases and creates additional test cases to, increase coverage. It also identifies redundant test cases which do not increase coverage, and measure the quality of test cases, etc [12].

Many coverage measures, such as function coverage,call coverage, statement coverage, branch coverage,condition coverage, branch/condition coverage, path coverage, data flow coverage, etc have been proposed in literatures [12]. Thesecoveragemeasures have their own advantages and disadvantages when analyzing coverage. The remainder of this paper is organized as follows.InSection 2, types of mutation testing of object-oriented programs are discussed. In Section 3, we discuss classical approach to mutation testing of object oriented program. The mutation testing of object oriented programs that would be useful in understanding different mutation operators of object-oriented programs. In Section 4, we discussed effectiveness of Mutation Testing.. In Section 5 and 6,we discussed an Implementation of our approach and experimental results. Section 7 concludes the paper.

2.Types of Mutation Testing

In this section, we review different types of mutation testing.

2.1 Strong Mutation Testing

Strong mutation testing [3, 4] involves making many small changes, one at a time, to a given program. Strong mutation testing tests for certain types of faults.One can create a very large number of "mutants" of a program. A mutant can be created by changing one token in the original program. For example, you might change a "<" to a ">" or you might replace a use of variable i with a use of variable j. Strong mutation depends on the specific mutation operators implemented, which in turn depend on the specific language used in a given program.

2.2Weak Mutation Testing

Weak mutation testing can be ascribed to the pioneering work done by Howden [5]. Let us assumethat C1 is a mutated version of simple component C. P is a program containing component C, and P1 is a mutated program containing C1. While testing component C with a set of test data for P if at least one execution of component C gives a different output as compared to the mutant component C1, the test data can be considered as adequate. Howden's work [6] does not provide a precise definition of program components but describes components as "normally corresponding to elementary computational structures in a program". He discusses five types of components. 1) variable references, 2) variable assignments, 3) arithmetic expressions, 4) relational expressions, and 5) Boolean expressions. Although the idea is further refined in Howden"sbook on functional testing [6], some examples of weak mutation components like relational expression, variable reference etc., have not previously been defined precisely enough for a generic implementation.

The advantages of the weak over the strong mutation testing are:

- In weak mutation program execution iscomparatively less, so it is cheaper.
- In weak mutation testing, each mutation doesnot necessarily require a different test run.

2.3 Firm Mutation Testing

Firm mutationtesting was introduced as an intermediate type of mutation testing. It represents the middle ground between weak and strong mutation testing. Weak mutation testing is an approximation technique that compares the internal statesofamutant and the original program immediately after the execution of the mutated portion of the program [5]. In case of firm mutation testing, execution of partial program could be considered instead of execution of each component as in the case of strong mutation testing. Firm mutation testing has the advantage of being cheaper as execution time is less and has precision as in the case of strong mutation. Still then, it is mostly unexplored [7].

3. Classical approach to mutation testing of object oriented program

Mutation operators are usually certain predefined program modification rules.They are typically designed to achieve coverage of certain programfeatures. Mutation operators are to a large extentdesigned based on the language of the program being tested. There are several mutation operators, each corresponding to a different class of simple errors that may arise in the specific language.

3.1 Class Mutation

These operators modify a program at the class level based on the language features that are affected; the class mutation operators are classified into four groups [8]. Class mutation for Java programsnormally considers encapsulation, inheritance, polymorphism and Java-Specific Features Faults. In the following we discuss the different class mutation operators. These mutation operators correspond to mistakes frequently committed by programmers.

3.2 Method mutation

Method-level mutation operators usually consider mutation operators which change expression by replacing, deleting, and inserting primitive operators. We aim to present mutation operators thatreplace, insert, and delete the primitive operators. A set of possible mutation operators, are shown in Table 2. These mutation operators correspond to mistakes frequently committedby programmers.

4. Effectiveness of Mutation Testing

Like other fault based approaches, efficiency of mutation testing, depends on types of faults the mutation system is intended to represent. At present the tools that are currently available are inadequate to test. Object-oriented specific features such as inheritance, polymorphism, and so on.

Mutation testing can be fully automated to bring a new level of error-detection capability to the software developer. At present the tools are semi-automatic in the sense that through mutant program are automatic the test suite has to be manually executed and the program results have to be manually checked to detect any faulty behavior of the program. Automated mutation testing tools can find Errors that are, once time-consuming to find and fix can be foundautomatically [9]. By using tools that incorporate mutation testing into state of-the-art error-detection technology, developers can to flush out more faults than with any other technology. Study shows that a complex error results from a combination of simple errors [10]. This shows that there is a correlation between simple syntax errors and complex syntax errors in a program. A set of mutant programs that are capable of finding out simple syntax errors will also be able to find out most of the complex syntax errors.

Comprehensive testing can be possible by proper choice of mutant operators. Research result shows that with an appropriate choice of mutation operators, mutation testing can be more powerful than path testing or domain analysis [11]. A additional advantage is that Mutation analysis is more easily automated than other types of testing [10].

5. Experimental Results

Operator	Description
AMC	Access modifier change
IHI	Hiding variable insertion
HD	Hiding variable deletion
IOD	Overriding method deletion
IOP	Overriding method calling position change
IOR	Overriding method rename
ISI	super keyword insertion
ISD	super keyword deletion
PC	Explicit call of a parent's constructor deletion
PNC	new method call with child class type
PMD	Member variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PCC	Cast type change
PRV	Reference assignment with other comparable variable
OMR	Overloading method contents replace
OMD	Overloading method deletion
OAC	Arguments of overloading method call change
JTI	this keyword insertion
JTD	this keyword deletion
JSI	static modifier insertion
JSD	static modifier deletion
JID	Member variable initialization deletion
JDC	Java-supported default constructorCreation

Table 1: Class-level Mutation Operators

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
AOR	Assignment Operator Replacement

Table 2: Method-level Mutation Operators

A set of possible class and method level mutation operators, are shown in Table1 & 2.
Test data is applied to the mutated program. The output is checked and compared with the original program. If the output is same as that of the original program, then the mutant is live otherwise it is killed or dead.

In Table 3 some new operators are listed. They are applied to the proposed model.

M1	Change switch expression to produce switch case constant integer.
M2	Change Comparison to produce the same result
M3	Change Call sequence to method name can causedeadlock in wait()
M4	Change Private to Protected
M5	Change in New type
M6	Modify Increment Decrement Change
M7	Change in Argument Number
M8	Change Bean classes abstract
M9	Change Bean classes as final
M10	Replace Return Statement
M11	Change l inner classes to public

Table3: Description of mutation operators

Testcases were created for each type of identified operators. When these test cases were executed against the mutants, using path oriented data set generator thefollowing results were obtained. Table 4 describes the results of application,

Mutants	**TD_1**	**TD_2**	**TD_3**	**TD_4**	**TD_5**
M1	Live	Dead	Dead	Live	Dead
M2	Dead	Dead	Dead	Dead	Dead
M3	Dead	Dead	Dead	Dead	Dead
M4	Dead	Dead	Dead	Dead	Dead
M5	Dead	Dead	Dead	Live	Dead
M6	Dead	Dead	Dead	Dead	Dead
M7	Dead	Dead	Dead	Live	Dead
M8	Dead	Dead	Dead	Dead	Dead
M9	Dead	Dead	Dead	Dead	Dead
M10	Dead	Dead	Dead	Dead	Dead
M11	Live	Dead	Dead	Live	Dead

Table4: Results of application on mutants

Result of Execution of Mutant on Program P using table
MS: Mutation Score is the ratio of mutants killed over the total number of non-equivalent mutants. Mutation Score = (Number of dead Mutants/ Total Number of Mutants)*100.

Below Table 5 describes details of mutation score result.

Path of Oriented Test Data	Number of Dead Mutant	Number of Live Mutant	Mutant Score
TD1	9	2	9/11*100=81.81
TD2	11	0	11/11*100=1
TD3	11	0	11/11*100=1
TD4	7	4	7/11*100=63.63

TD5	10	1	10/11*100=90.90

Table5: Mutation score result

The testers main aim is to achieve mutation score to 1.00 .Depending on the mutation score for test case, eliminate the test cases which satisfy the following condition MS=0.

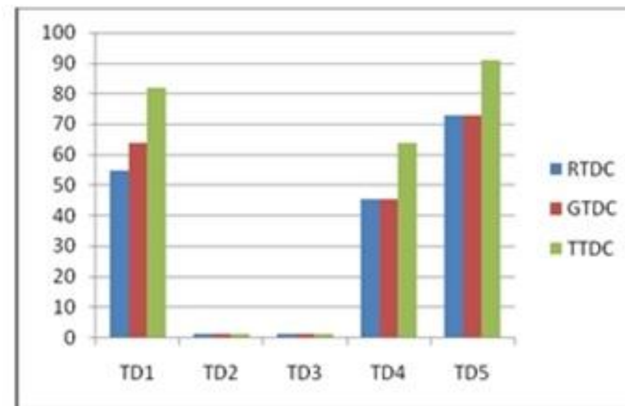

Figure 1 Result of comparison study

Figure 1 experimental results identifies comparison between different sets of creation of test data for our model.

6. Conclusion

High quality software cannot be developed without resorting to high quality object oriented testing. Testing object oriented programs, the objectoriented features need to be tested specifically. Mutation Testing is especially useful for testing of large software systems. Our preliminary results show that, using proper Path Oriented Test Databetter mutant score can be accomplished.

7. References

[1] R.G. Hamlet (1977), "Testing Programs with the Aid of a Compiler", IEEE Transactions on Software Engineering, Vol. 3, No. 4, pp. 279-290.

[2]Kim S., Clark J., Mcdermid J.(2000), "class mutation: mutation testing for object-oriented programs", published in ooss: object-oriented software systems, net.objectdays"2000, germany.

[3] A.T Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayword(1979), "Mutation analysis," Tech. Rep. GIT-ICS-79/08, School of Inform. and Computer. Science, Georgia Institute of Technology, Atlanta GA.

[4] R.A. DeMillo, R.J. Lipton, F.G. Sayward(1978), "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer, Vol. 11, No. 4, pp. 34-41.

[5] William E. Howden (1982), "Weak mutation testing and completeness of test sets", IEEE Trans.On Software Engineering, Vol. SE-8, pp. 371-379.

[6] William E. Howden (1987),"Functional Programming Testing and Analysis." ,New York: McGraw-Hill.

[7] J.R. Horgan and A.P. Mathur(1990) , "weak mutation is probably strong mutation", tech. rep. serc-tr-83-p, software eng. res. center, purdueuniv., w. laffayette.

[8] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt (2002), "Inter-class mutation operators for java". In proc. 13th international symposium on software reliability engineering, pp. 352-363.

[9] A.J.Offutt, "A Practical System for Mutation Testing: Help for the Common Programmer" (Parasoft Insure++®).

[10] David Banks, William Dashiell, Leonard Gallagher, Charles Hagwood, Raghu Kacker and Lynne Rosenthal:

"Software Testing by Statistical Methods Preliminary Success Estimates for Approaches based on Binomial Models", Coverage Designs, Mutation Testing, and Usage Models
[11] P.G. Frankl and E.J. Weyuker(1993) "A formal analysis of the fault-detecting ability of testing methods", IEEE Transactions on Software Engineering.
[12] Z. Chen, B. Xu, and J. Guan, "Test Coverage Analysis Based on Program Slicing", Chinese J. Electronics, 2003,20(3): 232-236