International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2015): 78.96 | Impact Factor (2015): 6.391

Survey on Object Oriented Mutation Testing

Snehal R. Takawale[1], Sandeep Kadam[2]

Abstract: ***Software testing is very important phase in software development. Various testing techniques are used with intention of finding software bugs. Different approaches are suggested to perform application testing, testers shall choose testing techniques in terms of cost and efficiency. Mutation testing is fault based testing technique which is widely used over decades. Several module and class testing techniques have been applied to object-oriented programs, but researchers have only recently begun developing test criteria that evaluate the use of key OO features such as inheritance, polymorphism, and encapsulation. Mutation testing is a powerful testing technique for generating software tests and evaluating the quality of software. However, the cost of mutation testing has traditionally been so high it cannot be applied without full automated tool support. Paper presents a method to reduce the execution cost of mutation testing for OO programs by using two key technologies, Mutant Schemata Generation (MSG) and byte code translation. This method adapts the existing MSG method for mutants that change the program behavior and uses byte code translation for mutants that change the program structure. A key advantage is in performance: only two compilations are required and both the compilation and execution time for each is greatly reduced. A mutation tool based on the MSG/byte code translation method has been built and used to measure the speedup over the separate compilation approach.***

Keywords: Software testing, Mutation testing, Mutants, Mutation testing tools

1. Introduction

Software testing is a practical technique to efficiently detect errors in software systems. Mutation testing is fault based testing techniques which is used to measure effectiveness of test suits. Using mutation testing, efficiency of test suits is measured. Mutation testing technique can be used in order to estimate the fault coverage of test suits. The idea of mutation testing was introduced by Richard Lipton in 1978.

Mutant generation is the first step of mutation testing process. Mutant is a copy of original program containing one fault which is syntactically correct. These faults are introduced using pre defined set of faults called mutation operator. After mutant generation, next step is to execute test cases against mutants and compare output with original program's output. When test case produces different output on mutant then mutant is said to be Killed mutant otherwise mutant is said to be live mutant. If no test case can distinguish its output from original program's output then, that is said to be Equivalent mutant. It is not possible to kill an equivalent mutant, as it is semantically equivalent to original program. The mutation score can be calculated by ratio of killed and live mutant. Mutation score indicates how effective given test case or test suit is. Testers can regenerate test cases to kill the remaining alive mutants and to raise the mutation score because a test case set with higher mutation score is considered more effective.

Object oriented program and language that solves the problem and provide the solution for old problem.

Mutation testing is based on the assumption that a program will be well tested if a majority of simple faults are detected and removed. Simple faults are introduced into the program by creating a set of faulty versions, called mutants. These mutants are created from the original program by applying mutation operators, which describe syntactic changes to the programming language. Test cases are used to execute these mutants with the goal of causing each mutant to produce incorrect output. A test case that distinguishes the program from one or more mutants is considered to be effective at finding faults in the program. Mutation testing involves many executions of programs; thus cost has always been a serious issue. Many techniques for implementing mutation testing have proved to be too slow for practical adoption. This paper presents a design and results from an implementation of a mutation system that is based on a novel execution strategy that combines mutation schemata with byte code translation.

Several approaches have been developed to reduce the computational expense of the mutation testing. Untch categorized the approaches into three strategies, do fewer, do smarter, and do faster. The do fewer approaches try to run fewer mutant programs without incurring intolerable loss in effectiveness. The do smarter approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs. The do faster approaches focus on ways to generate and run mutant programs as quickly as possible. These methods have been developed for traditional programming languages, and are not all applicable to OO languages. This paper presents a do faster method for OO inter-class mutation testing. This involves examining whether existing do faster methods can be applied to object-oriented programs. This approach primarily attempts to reduce the compilation time. These ideas have been implemented in an automated OO mutation system, which has been compared with previous execution techniques. Most of the OO mutation operators are independent of language; however, they have only been implemented in Java and so have some Java dependencies. The implementation method depends on the use of reflection, so can only be used in languages that support reflection.

A major difference for testers is that OO software changes the levels at which testing is performed. In OO software, unit and integration level testing can be classified into four levels: (1)

intra-method, (2) inter-method, (3) intra-class, and (4) inter-class.

Intra-method Level: Intra-method level faults occur when the functionality of a method is implemented incorrectly. Testing within classes corresponds to unit testing in conventional programs. So far, researchers have assumed that traditional mutation operators for procedural programs will saucefor this level (with minor modifications to adapt to new languages).

Inter-method Level: Inter-method level faults are made on the connections between pairs of methods of a single class. Testing at this level is equivalent to integration testing of procedures in procedural language programs. Interface mutation, which evaluates how well the interactions between various units have been tested, is applicable to this level.

Intra-class Level: Intra-class testing is when tests are constructed for a single class, with the purpose of testing the class as a whole. Intra-class testing is a specialization of the traditional unit and module testing. It tests the interactions of public methods of the class when they are called in various sequences. Tests are usually sequences of calls to methods within the class, and include thorough tests of public interfaces to the class.

Inter-class Level: Inter-class testing is when more than one class is tested in combination to look for faults in how they are integrated. Inter-class testing specializes the traditional integration testing and seldom used subsystem testing, where most faults related to polymorphism, inheritance, and access are found.

2. Class Mutation Operators

Class Mutation Operators the first attempt to define mutation operators to detect faults related to OO-specific features. They designed thirteen class mutation operators that were extended to sixteen. A subsequent systematic classification of OO specific faults in terms of language syntax revealed several types of OO faults that the previous operators do not model. Proposed a different scheme for mutating objects. Their approach relies on making changes to the data state of objects during execution rather than the program.

1) **Information Hiding** (Access Control) in our experience, access control is a common source of mistakes among OO programmers. The semantics of the various access levels are often poorly understood, and access for variables and methods is not always considered during design. Poor access definitions do not always cause faults initially, but can lead to faulty behavior when the class is integrated with other classes, modified, or inherited from. The AMC mutation operator has been developed for this category.
2) **Inheritance:** Although inheritance is a powerful and useful abstraction mechanism, incorrect use can lead to a number of faults. Seven mutation operators have been defined to test the various aspects of using inheritance, covering variable hiding, method overriding, the use of super, and definition of constructor s.
3) **Polymorphism:** Polymorphism and dynamic binding allow object references to take on different types in different executions and at different times in the same execution. That is, object references may refer to objects whose actual types differ from their declared types. In most languages (including Java and C++), the actual type can be any type that is a subclass of the declared type. Polymorphism allows the behavior of an object reference to differ depending on the actual type. Four operators have been developed for this category.
4) **Overloading Method** overloading allows two or more methods of the same class or type family to have the same name as long as they have different argument signatures. Just as with method overriding (polymorphism), it is important for testers to ensure that a method invocation invokes the correct method with appropriate parameters. Four mutation operators have been defined to test various aspects of method overloading.

Language Features	*Operator*	*Description*
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISK	Super keyword deletion
	IPC	Explicit call of parent's constructor deletion
Polymorphism	PNC	New method call with childe class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method content change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java specific features	JTD	This keyword deletion
	JSC	Static modifier change
	JID	Member variable initialization deletion
	JDC	Java supported default constructor creation
Overloading	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Access method change
	EMM	Modifier method change

3. Existing systems and Approaches

Roger T. Alexander, James M. Bieman, SudiptoChosh, and BixiaJi

They develop mutation operators and support tools that can mutate Java library items that are heavily used in commercial software. Mutation engine can support reusable libraries of mutation components to inject faults into objects that instantiate items from these common Java libraries.

T. Alexander and A. Jefferson Offutt

The emphasis in object-oriented programs is on defining abstractions that have both state and behavior. This emphasis causes a shift in focus from software units to the way software components are connected. Thus, they are finding that they need less emphasis on unit testing and more on integration testing. The compositional relationships of inheritance and aggregation, especially when combined with polymorphism, introduce new kinds of integration faults. This paper presents results from an ongoing research project that has the goal of improving the quality of object-oriented software. New testing criteria are introduced that take the effects of inheritance and polymorphism into account. These criteria are based on the new analysis technique of quasi- inter procedural data flow analysis. These testing criteria can improve the quality of object-oriented software by ensuring that integration tests are high quality.

Michelle Cartwright and Martin Shepperd

This paper describes an empirical investigation into an industrial object-oriented (OO) system comprised of 133,000 lines of C++. The system was a subsystem of a telecommunications product and was developed using the Shlaer-Mellor method. From this study, they found that there was little use of OO constructs such as inheritance and, therefore, polymorphism. It was also found that there was a significant difference in the defect densities between those classes that participated in inheritance structures and those that did not, with the former being approximately three times more defect-prone. They were able to construct useful prediction systems for size and number of defects based upon simple counts such as the number of states and events per class. Although these prediction systems are only likely to have local significance, there is a more general principle that software developers can consider building their own local prediction systems. Moreover, we believe this is possible, even in the absence of the suites of metrics that have been advocated by researchers into OO technology. As a consequence, measurement technology may be accessible to a wider group of potential users.

T. E. Cheatham and L. Mellinger

Object-oriented Software Systems present a particular challenge to the software testing community. This review of the problem points out the particular aspects of object-oriented systems which makes it costly to test them. The flexibility and reusability of such systems is described from the negative side which implies that there are many ways to use them and all of these ways need to be tested. The solution to this challenge lies in automation. The review emphasizes the role of test automation in achieving adequate test coverage both at the unit and the component level. The system testing level is independent of the underlying programming technology.

H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen

Several techniques have been proposed for class-level regression testing. Most of these techniques focus either on white- or black-box testing, although an integrated approach can have several benefits. As similar tasks have to be carried out for both white- and black-box testing, an integrated approach can improve efficiency and cost effectiveness. The article explains an approach for class-level regression testing, integrating existing techniques.

P. Chevalley and P. Trevino-Fosse

JavaMut (2001) developed a tool to perform the mutation analysis. It is a GUI (Graphical User Interface) based tool. The tool is implemented using compile time reflective system called OpenJava. One good feature of this tool is that tester can view the mutated code in the tool. *JavaMut* implements 26 mutation operators; six selective mutation operators fifteen from Class Mutation, and five new operators that authors created.

I. Moore

Jester (2001) developed tool for mutation testing. This tool provides support for java programs. This is simple as it supports mutation operator that can be run on single class. It uses JUnit as base that supports unit testing of java programs. Due to unit level testing, this can not apply on object oriented features.

R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia

Object Mutation Engine (2003) it performs mutation testing on objects of java APIs.This tool has complex architecture.

J. Offutt, Y.-S. Ma, and Y.-R. Kwon

MuJava this is a GUI (Graphical User Interface) based tool that allows both generation of mutants and execution of them automatically. The first version of tool implements 29 mutation operators in total; five selective traditional mutation operators [10], and twenty-four object oriented mutation operators from the work of [11]. There are three main modules in the MuJava tool. First is *Mutants Generator* (creates mutants), second module is *Mutants Executor* (executes mutants), and third module is *Mutants Viewer* (displays results). This tool has been used to perform experiments to evaluate the object oriented mutation testing and operators used and results are available in [12].

Authors have launched extended version of MuJava [13] with some changes, omissions and additions to class mutation operators (see Section III.F for details). Now MuJava supports up to 34 mutation operators including 5 mutation operators from conventional paradigm and 29 mutation operators from object oriented paradigm.

J.S. Bradbury, J.R. Cordy, and J. Dingel

ExMan (2006) propose a generalized approach for experimental mutation analysis. In this approach artifacts and components can be interchanged with each other in order to compare quality assurance tools. To use this tool we have to pass through a setup phase that involves creation of profile that tells the tools about command-line usage and purpose of using this application. Secondly, it involves selection of project to be run. Then it selects original source code and generates mutants. After that mutants and original source code are compiled. The benchmarks to compare results or assertions are provided to the ExMan and finally it performs analysis on the basis of benchmarks and produces results.

Sourceforge

Jumble (2007) this is mutation testing tool for Java programs. Jumble is quite simple in nature as it only supports mutation operators that can be run on a single Unit (class). It uses JUnit as base that supports Unit testing of Java programs. Due to unit level testing this tool cannot apply

Mutation operators that are designed for other features like inheritance and polymorphism and work at integration (system) level.

B. Grun, D. Schuler, A. Zeller

JavaLanche (2009) propose a framework and have implemented in a tool called JavaLanche. The main purpose of the study is to check the impact of equivalent mutants in the mutation testing. JavaLanche implements selective mutation operators. This tools uses coverage data about a test set to run only those test cases that execute the mutated statement in the code.

Tools	Complier	Strong mutation	Functional qualification	Mutant sampling	Selective mutation	Higher order mutation	Mutant clustering	integration	Mutant schemata	Byte code translation	Parallel execution	Weak mutation	Flexible weak mutation
AjMutator	Yes	No	Yes	No	Yes	No	No	No	No	No	No	No	No
Bacterio	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes
ExMan	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	No
Javalance	No	No	Yes	No	Yes	No	No	No	Yes	Yes	Yes	No	No
JavaMut	Yes	Yes	No	No	Yes	No	No	Yes	No	No	No	No	No
Judy	Yes	No	Yes	No	Yes	No	No	No	Yes	No	No	No	No
Jumble	No	No	Yes	No	Yes	No	No	No	No	Yes	No	No	No
MuClipse	Yes	No	Yes	No	Yes	No	No	No	No	No	No	No	No
Mugamma	Yes	No	No		Yes			No	Yes	No	No	Yes	No
muJava	Yes	Yes	No	No	Yes	No	No	No	Yes	Yes	No	No	No
Testooj	Yes	Yes	No		Yes	Yes		No	No	No	No	No	No

4. Proposed System Algorithm

Problem statement

There are problem for identifying mutants that change the program behavior, parsing the program, change the behavior of a program during execution and dynamically initiate object. Methods not invoked dynamically. We analyze there are several method for mutation testing but those method not proving the accuracy

Problem solution

This helps solve the first problem in implementing a mutation analysis system, parsing the program. Second, it provides an API to easily change the behavior of a program during execution. This can be used to create mutated versions of the program. Third, it allows objects to be instantiated and methods to be invoked dynamically. Java provides a built-in reflection capability with a dedicated API. This allows Java programs to perform functions such as asking for the class of a given object, finding the methods in that class, and invoking those methods. However, the Java language as defined does not provide full reflective capabilities. Specifically, Java only supports introspection, which is the ability to introspect data structures, but does not support alteration of the program behavior. Several reflection systems have been proposed to complement the Java reflection API.

MSG method

The MSG method used to encodes all mutants from Meta mutants. Meta mutants are one kind of parameterized program. Meta mutants are identifying from program and Meta mutants compile using standard compiler used to compile p.

We used MSG for compile-time reflection to generate a Meta mutant program. To execute mutants, mutants are loaded into JVM.

Byte code Translation:

This technique used for generating structural mutants. Structural mutants used to changing the structure of program like data structure of variable and method declaration. We used Byte code engineering library and it supports all structure mutation operator.

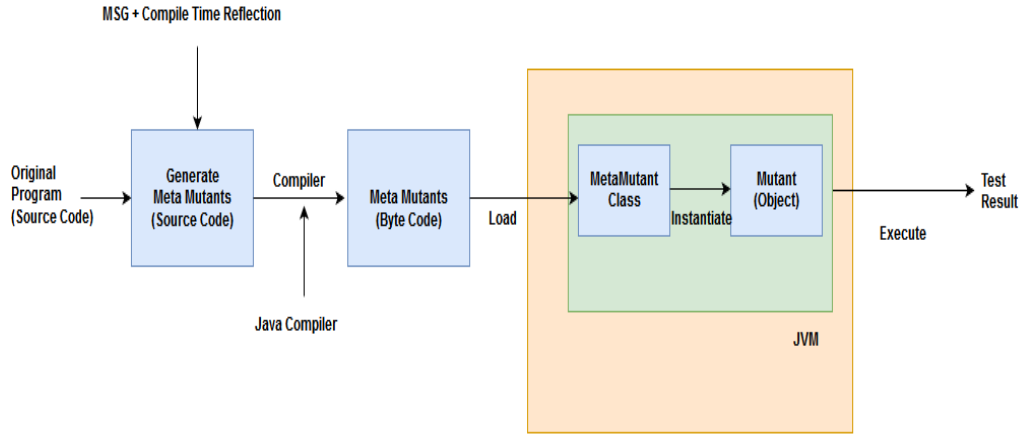

Figure: Proposed system Architecture

5. Mathematical Model

Input: .class
Output: Test Cases and Mutation Score

Process:

- Collects Mutants from Given Program. (We use Java Reflection API for collecting method name and data structure of variable.)
- Generate T Test cases from Given Mutants.
- Analyze the Mutant's Operator.
- Generate Mutants Score MS(p, TS) = K / (T - EQ)

P- Program under Test
TS- Test Suite
K- Killed Mutants
T- Total Mutants
EQ- Number of Test Case

6. Conclusion

In this project two techniques to reduce time for mutant generation. It requires two compilation one original program source code compilation and second for compilation of MSG met mutants. This technique used for improving quality of software. Byte translation to make system portable because system can work with any standard java compiler.

References

[1] Roger T. Alexander, James M. Bieman, SudiptoChosh, and BixiaJi. Mutation of Java objects. In Proceedings of 13th International Symposium on Software Reliability Engineering, pages 341{351, Annapolis MD, November 2002. IEEE Computer Society Press.
[2] Roger T. Alexander and A. Jefferson offutt. Criteria for testing polymorphic relationships. In Proceed- ings of the 11th International Symposium on Software Reliability Engineering, pages 15{23, San Jose CA, October 2000. IEEE Computer Society Press.
[3] Michelle Cartwright and Martin Shepperd. An empirical investigation of an object-oriented software system. IEEE Transactions on Software Engineering, 26(8):786{796, August 2000
[4] T. E. Cheatham and L. Mellinger. Testing object-oriented software systems. In 1990 ACM Eighteenth Annual Computer Science Conference, pages 161{165, February 1990
[5] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing. ACM Transactions on Software Engineering Methodology, 7(3):250{295, 1998.
[6] P. Chevalley and P. Trevino-Fosse, "A Mutation Analysis Tool for Java Programs," LAAS Report No 01356, Toulouse, France, September 2001, M. Daran and P. Trevino-Fosse. Software Error
[7] I. Moore. Jester. http://jester.sourceforge.net/, 2001. Accessed April 2012
[8] R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia, "Mutation of Java objects," In Proceedings of IEEE 13th International Symposium on Software Reliability Engineering, 2003
[9] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "An Experimental Mutation System for Java," ACM SIGSOFT Software Engineering Notes, vol.29 no.5, September 2004
[10] J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, "An experimental determination of sufficient mutant operators," ACM Trans Software Eng Methodol 5, pp 99–118, 1996
[11] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class Mutation Operators for Java," In Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering, pp 352-363, Annapolis MD, November 2002
[12] Y.-S. Ma, M. J. Harrold, and Y.-R. Kwon, "Evaluation of Mutation Testing for Object-Oriented Programs," Proceedings of the 28th international conference on Software engineering, May 20-28, 2006, Shanghai, China
[13] J. Offutt, Y.S. Ma, and Y.R. Kwon. The class-level mutants of mujava. In AST '06: Proceedings of the 2006 International Workshop on Automation of software test, pages 78–84, NY, USA, ACM 2006
[14] J.S. Bradbury, J.R. Cordy, and J. Dingel, "Exman: A generic and customizable framework for experimental mutation analysis", Mutation Analysis, Workshop on, 0:4, 2006

[15] Sourceforge. Jumble. http://jumble.sourceforge.net/, 2007. Accessed April 2012
[16] B. Grun, D. Schuler, A. Zeller, "The impact of equivalent mutants", In Proceedings of the 4th International Workshop on Mutation Testing, 2009