Volume 3, Issue 10, October 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Investigation of Mutation Testing & its Operators for Testing Case Generations

Pawar Sujata G, Idate Sonali R.
BVDU's College Of Engineering,Pune
Bharati Vidyapeeth University, India

Abstract: ***The program mutation is a testing technique that assesses the quality of test input data by examining whether the test data can distinguish a set of alternate programs (representing specific types of faults) from the program under test. Mutation based testing is used to discover new possible errors in software applications. This is since in this testing approach, intentional incorrect lines of codes are injected to check the soft- ware ability to produce results that are deferent from the correct or original code. In this paper a technique to generate valid and mutant test cases is proposed and developed. In most mutation techniques, one or more values or parameters in the specification, code, model, etc are intentionally modified and then test cases are generated to see if injected modifications can be detected. The goal of this approach is to discover possible errors or problems in the program that may not be discovered by other methods. A robust model is expected to differentiate between a valid and invalid sequence of events. An automatic execution and verification technique is also developed to evaluate the test cases that were rejected by the model and calculate coverage based on the number of rejected test cases to the total number of test cases.***

Index Terms: Mutation testing, Test case generation, test case execution and verification & Random test case generation.

I. INTRODUCTION

The mutation method is a fault-based testing strategy that measures the quality/adequacy of testing by examining whether the test set (test input data) used in testing can reveal certain types of faults [1]. It is widely acknowledged that testing activities consume a significant amount of software project resources [2]. This is accomplished by changing a small part of the code or the specification. Test cases are then applied to see the test cases that can kill (i.e. discover) those mutants. In this paper, mutation is used to evaluate the reliability of the GUI model. In traditional code mutation processes, mutation coverage can show whether test cases would expose the use of wrong operators and also wrong operands [3].

It works by reporting coverage of conditions derived by mutating (i.e. substituting) the program's expressions with alternate operators, such as "less than" substituted for "more than". In the traditional mutation, mutation is occurred to the code or the specification and test cases are expected to discover this mutation [4].

In this paper, the process is reversed. Mutation occurs in test cases and the GUI model is expected to discover those mutations. Such approach may fall under model based testing techniques where the GUI model is tested for its ability to kill (i.e. reject, in the scope of this paper) wrong test cases [5] [6]. Mutation testing is a fault based testing technique firstly proposed in 1978. It measures the effectiveness of test cases. Mutation testing is based on the concept of fault seeding in a program that will be completely tested and all faults are detected and removed. The set of faults which are introduced in a program are called mutants. A test input distinguishes two programs if the programs produce different output for this input. Mutation testing is based around a simple notion: if a test set is good at distinguishing our program from other similar programs then it is likely to be good at finding faults. The justification for this is that if the program is faulty, then testing may be seen as trying to distinguish the faulty program from some (slightly different, correct) version of the program [5]. Generally speaking, mutation analysis uses well defined rules (called mutation operators) that are defined on syntactic structures (such as grammars or program source) to make systematic changes to the syntax or to the objects developed from the syntax.

More precisely, given a program p, a mutant is some variant p‟ of p. typically we generate mutants by applying mutation operators: rules that allow us to transform programs. For example, the rule that says

“replace the occurrence of the arithmetic operator + by the arithmetic operator *” is a mutation operator.

Another example is replacing > by in a predicate. Normally we produce a mutant by applying one instance of one operator only: such mutants are called first-order mutants [6].

Typically for testing, only first order mutants are considered. If we apply a mutation operator to a mutant, we generate a mutant of a mutant. This is called a second order mutant. If we mutate a second order mutant, we obtain a third order mutant and so on. These “higher order” (i.e. higher than first order) mutants are not normally considered in Mutation Testing [7] [8].

Generally speaking, mutation analysis uses well defined rules (called mutation operators) that are defined on syntactic structures (such as grammars or program source) to make systematic changes to the syntax or to the objects developed from the syntax. When applied to programs, mutation analysis makes systematic changes to the program, and then asks the tester to design inputs that cause the mutated program (mutant) to create output that is different from the original version of the program. We customarily think of these mutants as being faults, although it is possible that the original program was faulty and the mutant is correct, or the mutant has no affect on the functional behavior of the program (equivalent). Most mutation systems introduce one change at a time (changing one terminal symbol in the grammar), although it is possible to make multiple changes (called higher-order mutants). This paper is focused on program mutation and assumes all mutants are single-order [10].

II. LITERATURE REVIEW

In this literature survey, several relevant papers are discussed. Those papers discussed using mutation for evaluating coverage and test case effectiveness. All papers listed as references in this paper focus on generating mutation operators based on one aspect of software products and then evaluate test coverage or effectiveness from this mutation process. The major difference that distinguishes one paper from the others is the software aspect that mutation operators are generated from (e.g. requirements, software model, code, state diagram, etc) [2]. One of the prominent researchers in the area of testing in general and mutation testing in particular is Jeff Offutt at George Mason University (cs.gmu.edu/ Offutt). He has several books, book chapters and relevant papers sole or with friends and students. He also developed the widely used mutation tool mu Java (http://cs.gmu.edu/ offcutt/mujava/). Examples of some of those contributions include. These papers discussed developing and using mutation tools such as Java and Mothra. They also discussed Mutation operators and using mutation in source code, Web applications and object oriented code. Coverage (e.g. code, and path) was a criterion to evaluate the effectiveness of mutation against it. Mutation can be divided based on the software stage where it occurs, or based on the software product component(s) upon which the mutation process occurs. For example, there are several papers that discuss: source code mutation (e.g. Java, object oriented code), Windows or Web mutation, test cases' mutation, database, integration testing, design or requirement mutation. The selection of the papers in this literature review of related work is based on selected variations of these different mutations [5].

In Choiet al presented one of the earliest mutations based testing environment. Other examples of mutation tools include: Java, Clipse, Javalanche, Jumple, Certitude, Jester, Proteum, and SQLMutation. In our paper, a new mutation tool is developed for the particular mutation in user interface components. The Mothra testing project was initiated in 1986 by members of the Georgia Institute of Technology's Software engineering research center. Mothra is written in FORTRAN and consists of a collection of individual tools, each of which implements a separate, independent function for the testing system. Examples of some of its mutation operators include relation operators' replacement. In original Mothra, the tool converts the tested code to an intermediate code in order to execute its mutated version by an interpreter.

In a recent paper, Mateo et al proposed mutation operators that are somewhat related to GUI components as some of these operators were trying to evaluate whether a component is interchanged with an earlier version of the same component which is something that may occur frequently especially with an evolving application. This system level mutation is a continuation for a work done in this area previously by Delamaro et al work on MuJava. On the GUI level, the mutation operators proposed consider a small subset of the possible GUI mutants such as GUI components' position or order inter- change, component deletion and modification. However, as the paper had a large scope, extensive evaluation and implementation of these mutations were not mentioned. As our paper came as an extension of a GUI test automation tool which includes the automation of all GUI testing activities: generation, execution and verification, the tool is utilized and extended to generate GUI mutants along with implementing the ability to automatically execute these mutants and evaluate their results [3] [4].

In Offutt et al described how to use the information of equivalent mutation for the problem of some paths' feasibility. An equivalent mutant is the one that will always produce the same output as the original program, so no test case can kill it. This affects the mutation score and causes it always to be less than complete coverage. Earlier, in his dissertation, Offutt proposed using Constraint-Based Testing (CBT) for detecting equivalent mutants [5].The paper presented a method to detect equivalent mutants in code through using constraints. These constraints are applied on the input domain to narrow its scope. GUI mutation has some possible equivalent mutation operators that will be discussed in this paper. Li et al evaluated mutation as a coverage criterion in comparison with other testing coverage criteria such as path, edge, etc. Coverage is a test case metric that is used to measure the ability of a particular test approach to cover one or more aspects of the software code that may include: statement, branch, path, etc. coverage. The study focus was on unit testing and showed that mutation can actually be more effective in terms of coverage from many other test criteria [9]. Unlike mutation score, mutation coverage calculates the number of faults that can be detected using the mutation process. In our GUI testing approach, the GUI is serialized from the actual application dynamically using a reverse engineering process. (i.e.

.NET reflection). As a result, the GUI model is represented by an XML ¯le which includes the GUI components, hierarchy and attributes. Mutations are created based on the XML ¯le and applied on the actual GUI during the execution process. However, since a GUI test case looks like a GUI path (e.g. File, Save, Exit), mutation can be also applied on or generated from the test cases. In typical cases, the program is mutated and the test cases are used to detect this mutation. However, in GUI, it is possible to reverse the process through generating a mutated test case (that may include for example an invalid GUI component) and then execute it on the application where its failure is an indication that the mutant is killed. If the mutant test case failed then this is a possibility of two: either the mutant test case is

equivalent and looks like a normal test case to the GUI, or the test case is killable but the current state of the GUI failed to kill this mutant.

III. PROPOSED ALGORITHM

3.1 Mutation Testing

The program mutation is a testing technique that assesses the quality of test input data by examining whether the test data can distinguish a set of alternate programs (representing specific types of faults) from the program under test. The mutation method is a fault-based testing strategy that measures the quality adequacy of testing by examining whether the test set (test input data) used in testing can reveal certain types of faults. Unlike other fault-based strategies that directly inject artificial faults into the program, the mutation method generates simple syntactic deviations (mutants) of the original program, representing „typical" programming errors.

Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. A finite set of mutant operators are applied to a given type of expression in the code. For example, a mutation system replaces an arithmetic operator (say +) in the original program with other operators (such as -, *, and /), which is intended to represent the programmer using a wrong operator. If a test set can distinguish a mutant from the original program (i.e. produce different execution results), the mutant is said to be killed. Otherwise, the mutant is called a live mutant. A mutant may remain live because either it is equivalent to the original program (i.e. it is functionally identical to the original program although syntactically different) or the test set is inadequate to kill the mutant. If the mutant is an equivalent mutant, it would always produce the same output, hence it cannot be killed. If a test set is inadequate, it can be improved by adding test cases to kill the (non-equivalent) live mutant. A test set that can kill all non-equivalent mutants is said to be adequate.

3.2 Mutation Operators for C++

For a program P, mutation testing produces a set of alternate programs. Each alternate program, P i , known as a mutant of P , is formed by modifying a single statement of P according to some predefined modification rule. These modification rules are called mutation operators. The syntactic change itself is called the mutation, and the resulting program is the mutant program, or simply mutant. This report defines a set of mutation operators for the C++ programming language. These operators are based on the operators defined for Ada, and Fortran operators used by Mothra. I organized my c++ operators primarily on the basis of what type of elements are modified; this gives us four types of operators. Mutation operators within these groups have reasonably uniform semantics and rules for applications.

3.2.1 Operand Replacement Operators:

Each operand replacement operator starts with the letter O. There are 18 operand replacement operators. These operators cause each operand to be replaced by each other syntactically legal operand. There are four kinds of operands included here:

Variables
Constants

Array References Pointers

Operand Replacement Operators

OVV Variable replaced by a variable
OVC Variable replaced by a constant
OVA Variable replaced by an array reference

OVP Variable replaced by a pointer reference
OCV Constant replaced by a variable
OCC Constant replaced by a constant
OCA Constant replaced by an array reference

OCP Constant replaced by a pointer reference
OAV Array reference replaced by a variable

OAC Array reference replaced by a constant
OAA Array reference replaced by an array reference

OAP Array reference replaced by a pointer reference
OAN Array name replaced by an array name
OPV Pointer reference replaced by a variable

OPC Pointer reference replaced by a constant
OPA Pointer reference replaced by an array reference

OPP Pointer reference replaced by a pointer reference

OPN Pointer name replaced by a pointer name

1. Object declared with the keyword CONSTANT is considered as CONSTANT and are mutated using OVC, OC?, OAC, and OPC.
2. The 18 simple replacement operators are all uniform and merely replace one type with another
3. OVV: Variable replaced by a variable.

Replace the variable in the program with each other variable when the types are the same.

4. OVC: Variable replaced by a constant.
   Replace the variable with each other constant when the types are the compatible.

5. OVP: Variable replaced by a pointer reference. Replace the variable with each other pointer when the base types are the compatible.
6. OCV: Constant replaced by a variable.

   Replace constant with each other variable when the types are compatible.

7. OCC: Constant replaced by a constant

   Replace constant with each other constant when the types are compatible.

8. OCA: Constant replaced by an array reference Replace constant with each other array reference when the types are compatible.
9. OCP: Constant replaced by a pointer reference Replace constant with each other pointer reference when the types are compatible.

10. OAV: Array reference replaced by a variable Replace array reference with each other variable in the program.

11. OAC: Array reference replaced by a constant Replace array reference with each other constant in the program.

12. OAA: Array reference replaced by an array reference

    Replace array reference with each other distinct array reference in the program.

13. OAP: Array reference replaced by a pointer reference

    Replace array reference with each other pointer reference in the program.

14. OAN: Array name replaced by an array name Replace array name with each other array name when base types are compatible, and the index types are the same.

15. OPV: Pointer reference replaced by a variable

    Replace pointer reference with each other variable in the program.

16. OPC: Pointer reference replaced by a constant Replace pointer reference with each other constant.

17. OPA: Pointer reference replaced by an array reference

    Replace pointer reference with each other array reference.

18. OPP: Pointer reference replaced by a pointer reference

    Replace pointer reference with each other distinct pointer reference in the program.

19. OPN: Pointer name replaced by a pointer name

    Replace pointer name in a pointer reference by other pointer names when field names and types are compatible.

3.2.2 Operators Insertion Operators:

Each operator insertion operator starts with the letter I. There are 2 operator insertion operators. These operators cause each operator to be inserted by each other syntactically legal operator. There are two kinds of operators included here:
Binary
Unary

IBO: Binary Operators Insertion

IUO: Unary operator insertion

1 IBO: Binary Operators Insertion

Binary operators include: Multiplicative operators: Multiplication (*), Division (/), Modulus (%). Additive operators: Addition (+), Subtraction (-).

IBO1 replaces each binary operators and additive operators with each other distinct binary operators and additive operators.

Relational and equality operators: Less than (<), Greater than (>), Less than or equal to (<=), Greater than or equal to (>=), Equal to (= =), Not equal to (!=).

IBO2 replaces each of these operators with each other distinct operators.

Logical operators: Logical AND (&&). Logical OR (||)

IBO3 replaces each of these operators with each other distinct operators.

Assignment operators: Assignment (=), Addition assignment (+=), Subtraction assignment (- =), Multiplication assignment (*=), Division assignment (/=).

IBO4 replaces each of these operators with each other distinct operators.

2. IUO: Unary Operators Insertion Unary operator includes: „!“, „&“, „~“, „*“, „+“, „++“, „-„, „- -„.

IUO replaces each unary operator with each other distinct unary operator.

1.3 Arithmetic Operator Replacement Operators:
There is one arithmetic operator replacement operator starting with a included here:
AOR: Arithmetic operator replacement

Each occurrence of one of the operators +, -, *, and / is replaced by each of the other operators.

IV. CONCLUSION AND FUTURE WORK

Developing a means of assessing how good generated test sets are is an important testing subject. Test cases are used to detect possible errors and bugs in software applications. In this paper, mutation based testing is used to test applications user interfaces and test if they can differentiate invalid from valid test cases. An automatic tool is developed to automatically generate test cases from applications user interfaces. Later on, and based on the generated test cases, an aspect of one component in each test case is changed to create test case mutations. An automatic execution and verification process is developed to evaluate the validity of the proposed mutations. The automatic execution and verification processes verify each control individually regardless of its test case. In mutation original test cases and their results are stored. Those are considered as the baseline for mutation based testing. After generating mutation, to test those mutations, a mutation is said to be killed if its test case result is different from that of the original. The validation of the results considers killing mutants by rejecting them. This makes the automatic verification process difficult due to the difficulty of defining the GUI correct and incorrect states.

REFERENCES

[1] Choi, B.J., DeMillo, R.A., Krauser, E.W., Martin, R.J., Mathur, A.P., Offutt, A.J., Pan, H., Spafford, E.H. The Mothra Tool Set. Proceedings of the 22nd annual Hawaii international conference on system sciences (HICSS'22), Kailua-Kona, HI , USA, (pp: 275- 284), vol. 2 (1989).

[2] Mateo, P.R. Usaola, M.P. Offutt, J. Mutation at System and Functional Levels. Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 6-10, Paris, France, (pp: 110), (2010)

[3] Offutt, A.J. Jie Pan. Detecting Equivalent Mutants and the Feasible Path Problem. Proceedings of the 11th annual conference on computer assurance COMPASS 96). 17-21 June, Gaithersburg, MD, USA, (pp: 224), (1996).

[4] Praphamontripong, U. Offutt, J. Applying Mutation Testing to Web Applications. Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 6-10 April, Paris, France, (pp: 132), (2010).

[6] DeMillo, R.A. Offutt, A.J. Constraint-Based Automatic Test Data Generation. IEEE Transactions On Software Engineering (TOSEM), VOL. 17, Issue 9, (pp: 900), Sep. (1991).

[7] Nan, Li Praphamontripong, U. Offutt, J. An Experimental Com-parison of Four Unit Test Criteria: Mutation,

Edge-Pair, All uses and Prime Path Coverage. IProceedings of the International Conference on Software Testing Verification and Validation Workshops, 1-4 April, Denver, CO, USA, (pp: 220), (2009).

[8] Offutt, A.J. Rothermel, G. Zapf, C. An Experimental Evaluation of Selective Mutation. Proceedings of the 15th International Conference on Software Engineering, 17-21 May, Baltimore, MD, USA, (pp: 100), (1993).

[9] Suet Chun Lee Offutt, J. Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis. Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), 27-30 Nov., Hong Kong, (pp: 200-209), (2001).

[10] [Agrawal89] Agrawal H. et al., Design of Mutant Operators for the C Programming Language, SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.

[11] [Delamaro97] Delamaro M. et al., Integration Testing Using Interface Mutations, 1997.

