Control Oriented Mutation Testing for Detection of Potential Software Bugs

Muhammad Bilal Bashir
Center for Software Dependability
Mohammad Ali Jinnah University
Islamabad, Pakistan
bilalbezar@gmail.com

Aamer Nadeem
Center for Software Dependability
Mohammad Ali Jinnah University
Islamabad, Pakistan
anadeem@jinnah.edu.pk

Abstract— **An effective test case suite is one that has the potential to reveal real faults in the code. Mutation testing is a fault-based technique that helps measure the strength of test case set and to generate effective test cases. Faults are injected through pre-defined mutation operators and mutation score is a mean to calculate usefulness of test case set. Mutation testing approach suffers from two of its inherent issues; it is computationally expensive, second generation of equivalent mutants. Along with original program, every mutant has to be executed against all the test cases. An equivalent mutant always produces same output on original and mutant program for every possible test case. Program's behavior comprises of two kinds of information; control flow and data flow. Mutation testing techniques that have been proposed so far consider only data flow information and they ignore the importance of flow of control within the program. In this paper we propose a new approach that encourages using control flow information along with data flow information in mutation testing for detection of bugs. Bugs can hide themselves within equivalent mutants and in these scenarios control flow information can be quite useful to uncover them. Experiments show that using control flow information can help revealing potential bugs in the program.**

Keywords- mutation testing; equivalent mutant; control flow; mutation score

I. INTRODUCTION

Testing is an important phase of software development life-cycle that allows the tester to test the software under development by providing inputs and comparing the outputs to the expected results. This phase is important because it checks that if the software is working according to the desires of client and it verifies that developers have developed the software right. Testing an object oriented paradigm is a challenging task due to its salient features and traditional testing methods are not competent enough to test these features. We either need to introduce extensions in the existing testing methods or we have to come up with new testing methodologies that can incorporate new object oriented features.

The concept of mutation testing was first coined in 1978 by DeMillo, Lipton, and Sayward [1]. Software testers and researchers have paid a lot of attention and shown their interest in this type of testing due to its nature. Mutation testing provides assistance in measuring the effectiveness and later generation of test case suite. The main idea behind mutation testing is to introduce plausible faults in the original program and run the test cases to see if they can help to identify these faults. This process leads the tester to generate an effective test case set that may help finding real faults in the program. Figure 1 shows an iteration of mutation testing process.

The mutation testing begins with the process of *mutant* generation. A fault, which is syntactically correct, is injected in the original program to produce a mutant. Mutation testing approaches offer a set of *mutation operators* that are used to introduce faults in the original programs. A mutation operator covers a set of similar changes that can be made in the original program. All the test cases are exercised on original program and mutants generated from it and outputs of both the versions are compared. If a single test case can distinguish output of an original program with the mutant's output, we declare such a mutant as *killed*, otherwise it is said to be *alive*. If no test case can distinguish the output of a mutant with the original program, that particular mutant is called *equivalent mutant*. It is not possible to kill an equivalent mutant by any mean because it is semantically similar to the original program. The effectiveness of test case set is measured by calculating *mutation score*, which is ratio of the total number of killed mutants and the total number of non-equivalent mutants. The process of mutation testing can be rerun by a tester in another iteration to kill more alive mutants and to increase mutation score.

Mutation testing is computationally expensive, which has been the main hindrance of getting large scale acceptance by software industry. We have to generate mutants against all applicable mutation operators and then need to execute all those mutants against all the test cases. This process makes mutation testing computationally expensive and time consuming task. Another problem with mutation testing is generation of equivalent mutants that increase the computational cost and also decrease the mutation score. Detection of equivalent mutant is an undecideable problem [3] but heuristics can be used to judge the nature of mutant and to prevent the generation of equivalent mutants.

All the object oriented mutation testing techniques [8, 9, 11, 12, 13, 15] use output (data flow information) of the program to either declare a mutant as killed or mark it as equivalent. A program's behavior is defined by two different elements i.e. control flow and data flow. In that case considering only data flow information may result in ignoring useful information that can help us identifying change in mutant's behavior as

978-0-7695-4927-9/12 $26.00 © 2012 IEEE
DOI 10.1109/FIT.2012.15

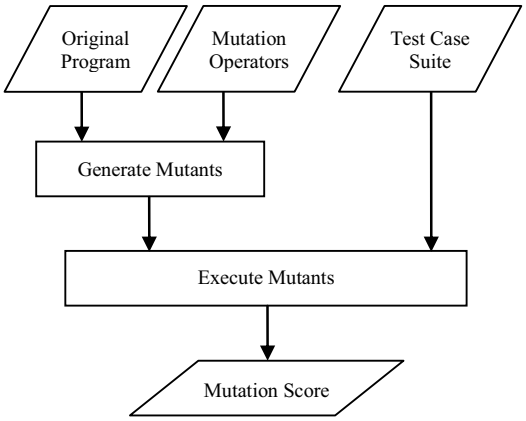


Figure 1: Mutation Testing Process Flow

compared to the original program when the same test case is executed on them. In this paper we propose a new approach that not only considers data flow information but also uses control flow information of the program to check if the mutant's behavior is changed or not. With the addition of control flow information we can identify suspicious equivalent mutants and their analysis can help uncovering potential bugs in the original program. In some cases when output of original and modified program stays unchanged but the control flow changes in modified program then analyzing equivalent mutants can be helpful identifying suspicious piece of code that may contain a bug.

Rest of the paper organized as follows; section II illustrates mutation testing approach whereas section III contains details about our new proposed approach. Section IV shows an example on how our new approach works. Section V concludes the discussion and section VI lists some future directions.

II. MUTATION TESTING

If we compare conventional mutation testing with object oriented mutation testing, not much difference is noticeable. In both cases we generate mutants using original program and mutation operators and later test case set is excursed against both the versions of the program. Mutation score is calculated considering total number of killed and non-equivalent mutants in both paradigms. The only difference we can find in object oriented paradigm is in the test case structure, which consists of an object creation, call to some methods to gain interested state of the object and then execution of method under test with all required input values that need to be passed on as arguments. The structure of test case is beyond the scope of this research, so we will not discuss it in detail. Figure 2 shows mutation operators set for object oriented paradigm in general.

In figure 2 we can see three ovals each representing a set of features and mutation operators. The 'structured paradigm features' oval represents features like primitive data types, arithmetic operators, and so on, whereas 'language specific features' oval covers set of language specific features like input/output streams, exception handling etc. The oval in the middle captures all the object oriented features. It borrows features from other two ovals and hence it may end up having large mutation operators set than the structure paradigm has.

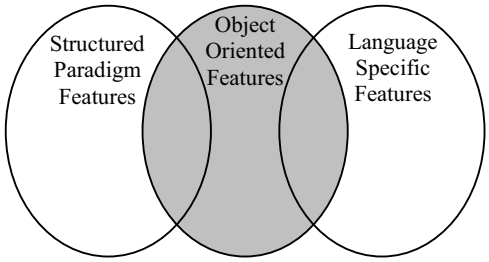


Figure 2: Mutation Operators Set

A. *Equivalent mutant*

An equivalent mutant always produces same output as of the original program against all possible the test cases for that program. Despite the syntactic differences, equivalent mutant is semantically similar to the original program. Detecting an equivalent mutant is undecideable problem [3] but some research [2, 4, 6, 7, 12, 14, 15, 16] has been done to devise heuristics to detect equivalent mutant. It is important that either we prevent generation of equivalent mutants or correctly detect them during mutation testing to reduce the computational cost.

We have shown below an example of equivalent mutant (written in Java programming language) that always produces the same output although a mutation has been performed. Figure 3(a) shows an arbitrary example of a class `Math` that declares a public data member `num` at line 2. It also declares a public method `calSquare()` at line 2-4 that returns an integer square of public data member `num` at line 3;

```
public class Math {
1.  public int num;
2.  public int calSquare() {
3.     return num * num;
4.  }
}
```

Figure 3(a): Original Program

Figure 3(b) shows mutated version of the Math class shown in figure 3(a). It applies *Access Modifier Change* mutation operator that has resulted in changing a public data member `num` to become private but that does not change the semantics of the program. For every test case both original and modified program produces the same output.

```
public class Math {
1.    private int num;
2.  public int calSquare() {
3.     return num * num;
4.  }
}
```

Figure 3(b): Mutated Program

III. CONTROL ORIENTED MUTATION TESTING

In this section we present our proposed technique for object oriented mutation testing. We propose using control flow information in mutation testing to detect suspicious equivalent mutant that can help identifying potential bug.

When a test case executes on the original and mutant program we note down the execution traces of both the programs. At the end of execution we compare the output of the program and if we find the test case produces same output on both the programs, we compare their traces. If the modified program experiences different trace we note down the mutant as well as the test case until and unless another test case kills that mutant. Once the number of mutation testing iterations exhaust and some mutants remain alive then we analyze all those mutants one by one. If a mutant does not have a test case that experiences a different execution trace, we call it an *equivalent* mutant and the rest those have at least one test case that experiences different execution trace are declared as *suspicious* mutants. This process is done only for those mutation operators those have the tendency to alter the control flow during the execution. Some mutation operators like 'Relational Operator Replacement' or 'Overriding Method Rename' can change the control flow during program execution and when control flow goes through different branches in the code, usually it produces different output. Therefore, considering these mutation operators if a mutant experiences different execution trace but produces same result then there is a chance that the piece of code in original program may have a bug in it. We now present a list of mutation operators taken from [5, 10, 15] by filtering out those operators that cannot change control flow of a program directly. There are total 16 such operators; 2 from conventional and 14 from object oriented mutation testing.

Conventional Mutation Operators [5, 10]

- Logical connector replacement (LCR)
- Relational operator replacement (ROR)

Object-Oriented Mutation Operators [15]

- Overriding method deletion (IOD)
- Overriding method calling position change (IOP)
- Overriding method rename (IOR)
- `super` keyword insertion (ISI)
- `super` keyword deletion (ISD)
- Explicit call of a parent's constructor deletion (IPC)
- `new` method call with child class type (PNC)
- Overloading method contents replace (OMR)
- Overloading method deletion (OMD)
- Member variable initialization deletion (JID)
- Java-supported default constructor creation (JDC)
- Reference comparison and content comparison replacement (EOC)
- Accessor method change (EAM)
- Modifier method change (EMM)

In object oriented paradigm, testing is performed method wise. After object's instantiation, tester performs certain method calls to attain interesting state of an object before executing the method under test. Before we go in the details of how the control flow of a program should be used in conjunction with program's output, it is important to define 'Program Output' in the context of mutation testing.

A. *Program's Output*

Generally a program output is the result of the program that it produces after performing certain operations on the data. In mutation testing and especially in object oriented mutation testing, it is important we clearly state what a program output is. In case of structured paradigm where code is written in form of functions, a program output can be considered as the value returned by a function. On the other hand in object oriented paradigm it is not sufficient just to consider the value returned by a function because object has states (data members) as well. So a function returning a value may also be defining object's state. In existing mutation testing studies, none has clearly defined what a program output is in the context of mutation testing? Is it just the value returned by a function or is it object's state? Due to the nature of object oriented paradigm, both of them should be taken into account. So we are using both of them for our research and when we state that a method or a function is producing an output that means either it is changing object's state or it is returning a value or both.

It is not necessary for a method in a class to either return a value or change object's state. It can still be a valid method and such methods exist in the code. So we have explained both of these scenarios in detail in next section. Later we discuss how our approach works in both scenarios i.e. "method produces an output" or when "method does not produce an output" one by one.

B. *Method Producing an Output*

If a mutated method produces an output then along with comparing output with the original program we propose to compare the execution traces (control flow) of both original and the mutated methods. We declare a mutant as *equivalent* if none of the executed test cases produces different output and experiences same execution trace. On the other hand if a mutant produces same output but experiences different execution trace for at least one test case or produces different output but experiences same execution trace, we declare that mutant as *suspicious* mutant. We encourage checking such suspicious mutants to ensure they do not contain a bug. Figure 4 explains the scenario in detail. A mutant is said to be *killed* if both output and execution trace differs for at least one test case.

The control flow of original program is shown in figure 4(a) where doted arrows show execution flow against a given test case T and it produces output 40. The other three control flows in figure 4(b), 4(c) and 4(d) represent three different cases against original program in 4(a). Figure 4(b) shows the test case produces same output but execution flow is different,

in figure 4(c) mutant produces different output with same execution flow, whereas figure 4(d) represents mutant with different execution flow as well as different output. With the application of control-oriented mutation testing we declare mutants in figure 4(b) and 4(c) as *suspicious* mutants whereas mutant in figure 4(d) is said to be *killed*. A mutant is *alive* if its execution trace as well as output remains the same and later it is declared *equivalent* if none of the test cases produces different output and execution trace.

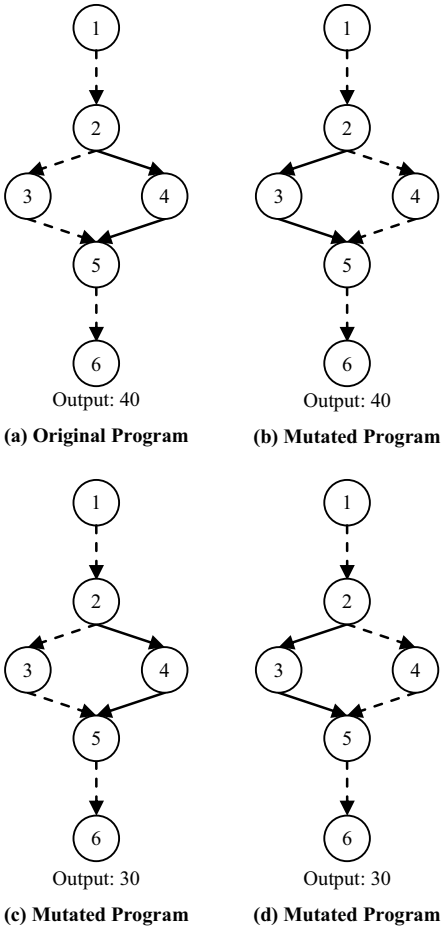


Figure 4: Control Flow Mutation Testing: Method Producing an Output

Control-oriented mutation testing can help in finding potential logical bugs in the program. We explain this with the help of an example. Consider the code snippet given in figure 5(a) that declares and defines a `arbitrary()` method with two integer parameters `a` and `b`. The method returns a value of type integer as well.

This method defines an instance level data member `i` which is declared in the class containing this `arbitrary()` method. The data member `i` is defined twice in this method. Both the definitions of `i` at line 2 and line 4 assigns value 10 making the branch structure useless. The value of data member `i` is eventually returned at line 6. Now consider mutated versions of `arbitrary()` method in figure 5(b).

```
public int arbitrary(int a, int b) {
1  if(a < b) {
2     i = 10;
3  } else {
4     i = 10;
5  }
6  return i;
}
```

Figure 5(a): Original Program

```
public int arbitrary(int a, int b) {
1  if(a > b) {
2     i = 10;
3  } else {
4     i = 10;
5  }
6  return i;
}
```

Figure 5(b): Mutated Program

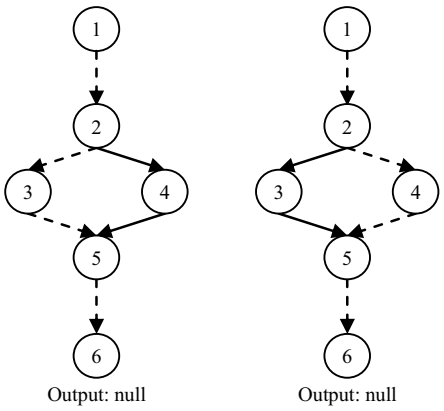
The mutated method declared in figure 5(b) has a mutation on line 1. The 'Relational Operator Replacement' mutation operator applies here and the original condition `if(a < b)` is replaced with mutated statement `if(a > b)`. The problem here is that either body of `if` statement executes or body of `else` executes, the program will always return 10 because expressions of defining data member `i` are same. The conventional mutation testing methods declares this modification as *equivalent* mutant because no test case can distinguish the output between them. On the other hand control-oriented mutation testing declares it as *suspicious* because although the output is same but control flow of both original and mutated programs is different.

C. Method Not Producing an Output

If a mutated method does not produce an output then we propose to use only execution traces (control flow) of original and mutated method to check if a mutant is killed or alive. In this case if we only consider output of the program (object's state), it will always be the same and this mutant will eventually be marked as equivalent mutant. Logically if the method does not produce an output, the execution trace should remains the same and mutation (from the list we present in section III) should not cause it to alter during execution. Figure 6 explains the scenario in detail;

The control flow of original program is shown in figure 6(a) where doted arrows show execution flow against a given test case T and it does not produce any output. Figure 6(b) shows the test case T produces different execution flow. With the application of control-oriented mutation testing we declare mutant 6(b) as *suspicious*. A mutant is said to be *alive* if its execution trace remains the same.

Later we present an example to illustrate the working of our control-oriented mutation testing technique.



(a) Original Program **(b) Mutated Program**

Figure 6: Control Flow Mutation Testing: Method Not Producing an Output

IV. AN EXAMPLE

In this section we present our idea with the help of an example. We apply "`new` method call with child class type (PNC)". Consider the Java language code snippet in figure 7;

```
public class Person {
1   private int age;
2   public Person( int a ) {
3     age = a;
4   }
5   public String findAgeGroup() {
6     if( age < 20 ) {
7       return "Teenage";
8     } else {
9       return "Adult";
10    }
11 }
}
```

(a) Person Class

```
public class Student extends Person {
1  public Student( int a ) {
2    super( a );
3  }
}
```

(b) Student Class

Figure 7: Illustration of Control-Oriented Mutation Testing

Figure 7(a) declares a Person class with a private data member age at line 1. It declares a parameterized constructor (line 2 to 4) accepting an integer parameter and assigns that integer to data member age. The Person class declares a method with name findAgeGroup() (line 5 to 10) that returns a String value depending upon the value of age data member. Figure 7(b) declares a class Student that inherits from Person class and just has a parameterized constructor with some other methods too. It inherits findAgeGroup() method from its parent class Parent. so Parent.findAgeGroup() executes either Student class reference calls it or Person class references.

Now consider code snippet shown in figure 8 that shows original and modified versions of a method declares in `University` class;

```
public class University {
1  public String stdAgeGroup(int age) {
2    Person per = new Person( age );
3    return per.findAgeGroup();
4  }
}
```

(a) University Class – Original Program

```
public class University {
1  public String stdAgeGroup(int age) {
2    Person per = new Student( age );
3    return per.findAgeGroup();
4  }
}
```

(b) University Class – Modified Program

Figure 8: University Class with Method `stdAgeGroup()` to illustrate control-oriented mutation Testing Technique

Figure 8(a) declares a `University` class with a public method `stdAgeGroup()` (line 1 to 4) that returns a `String` value. Line 2 creates a `Person` class instance and line 3 calls the method `findAgeGroup()` on instance created in line 2 and returns the value calculated by this method. The modified code in figure 8(b) is almost identical to the code in figure 8(b) except one change at line 2. The modified `University.stdAgeGroup()` method applies "`new` method call with child class type (PNC)" mutation operator and changes the call to `Person()` constructor with its child class' `Student()` constructor.

When the original and mutated code executes on a test case the method `Person.stdAgeGroup()` executes because `Student` class inherits it and does not override it. So applying "`new` method call with child class type (PNC)" mutation operator has not made any change to program's semantics hence the output remains same for all the test cases. With conventional mutation testing the mutant in figure 8(b) marks as *equivalent* mutant but with our new approach mutant in figure 8(b) declared as *suspicious* because call to Student class constructor changes the execution flow for mutated program.

V. CONCLUSION

Mutation testing is fault-based testing approach to measure the effectiveness of test cases and generate test cases that have potential to detect real faults in the program. Mutation testing is computationally expensive and suffers the problem of

equivalent mutants. Mutation testing techniques use output (data flow information) of the program to check if a mutant is killed or alive, whereas control flow information of the programs is ignored. Control flow information (execution trace) can be helpful especially when those mutation operators are applied that can directly change the control flow of a program. We propose a new technique named 'Control-Oriented Mutation Testing' that uses control flow information along with data flow to detect suspicious mutants that may reveal bugs in the program. We explicitly define program's output in this study and explain in detail how control flow information can be used in mutation testing. Experiments on small scale show promising results and practicality of this approach though more experiments need to be done on real world programs to check the impact of using control flow information.

VI. FUTURE WORK

We are planning to extend the scope of our experiments and apply the control oriented mutation testing on large scale on real world programs. This will help checking the impact of our approach and its practicality. We will use programs written in Java programming language for our experiments.

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer 11, pp 34–41, April 1978, J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] D. Baldwin, F. Sayward., "Heuristics for determining equivalence of program mutations", Research report 276, Department of Computer Science, Yale University, 1979

[3] T. A. Budd, D. Angluin, "Two notions of correctness and their relation to testing", Acta Informatica, 18:31{45, November 1982

[4] A.J. Offutt and W.M. Craft. Using compiler optimization techniques to detect equivalent mutants. Journal of Software Testing, Verification, and Reliability, 4:131–154, 1994

[5] 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

[6] A.J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification and Reliability, 7(3):165–192, 1997

[7] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. Software Focus, 9(4):233–262, 1999

[8] S. Kim, J. Clark, and J. McDermid, "The Rigorous Generation of Java Mutation Operators Using HAZOP," In 12th International Conference Software & Systems Engineering and their Applications, December 1999, unpublished

[9] S. Kim, J. Clark, and J. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs," In Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems, Erfurt, Germany, Oct. 2000

[10] E. F. Barbosa, J. C. Maldonado, A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," Software Test Verification Reliab 11, pp 113–136, 2001

[11] P. Chevalley, "Applying Mutation Analysis for Object Oriented Programs Using a Reflective Approach," In Proceedings of the 8th Asia-Pacific Software Engineering Conference, Macau SAR, China, December 2001

[12] 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

[13] 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

[14] K. Adamopoulos, M. Harman and R. M. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution", Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'04)Seattle, Washington, USA, 26th-30th, June 2004

[15] 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, New York, NY, USA, 2006. ACM

[16] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants", Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10)Paris, France, 6 April 2010.