Effect of Multiple Test Case Sets and Reduced Test Data of a Test Case Set on Mutation Testing

Updesh Kumar Jaiswal[a*], Suveg Moudgil[b], Fazal Ahmed Siddiqui[c]
[a]*Department of Information Technology*
IMS Engineering College, Ghaziabad, U.P., India
[b]*Department of Computer Science & Engineering*
IMS Engineering College, Ghaziabad, U.P., India
[c]*Department of Computer Science & Engineering*
Integral University, Lucknow, U.P., India
*[a]updesh.jaiswal@imsec.ac.in

Received: *30.04.2019,* ***Accepted:*** *24.05.2019*

Abstract

Mutation Testing is a white-box, fault-based software testing technique that measures the effectiveness of the test cases. In mutation testing to kill a mutant at least one test data is required, so to kill all mutants that can be generated by different types of mutation operators a large number of test data is required. In general practice to perform testing, tester generates multiple sets of test cases to satisfy the same criterion or according to the same procedure, and then computes their average performance. In this paper we raise some questions: Does tester need to generate multiple sets of test cases in mutation testing? If a single test case set is sufficient to find and kill all the mutants present in a program then how can we reduced the size of this test case set? To give the answers of these questions we performed experiments on different programs in a controlled environment. We found that the practice of using multiple sets of test cases is not necessary in mutation testing and proposed a new approach that can reduce the size and cost of a selected test case set.

Keywords - Software testing; mutant; mutation testing; test case set; test data

Introduction

Software testing is a complex activity which must be done to find the errors and quality assessment. The key challenge in the process of software testing is to reduce costs, time and maximize benefits. Software testing involves test planning, design, execution, evaluation and reporting activities (Y.S, Ma, 2010; Jia and Harman, 2015). Test design is a very important activity of software testing, it includes reviewing the test basis, identifying test conditions, designing tests, evaluating them and then designing the test environment setup. One of the critical tasks in testing is the generation of test data (Y.S, Ma, 2010; Deng *et al.,* 2013; Untch, 2011; Just *et al.,* 2012).

Mutation testing involves the creation of different mutants and mutant programs of the program being tested. A mutant program contains at least one syntactic change that is made to an original program and a mutant is a simple modification in the original program (Denag *et al.,* 2013; Papadakis and Malevris, 2018). Thus each mutant contains at least one single fault. If one test data is generated according to one mutant, then the total number of test data required to kill all mutants in a program will be equal to the number of all the possible mutants that can be generated (Untch, 2011; Just *et al.,* 2012; Jia and Harman., 2015; Kaminski *et al.,* 2013). In practice, this may become a big burden and required high cost as well as labor.

Example of the Mutants

Let us take an example of a program that can find out the largest integer among the three integers. This program can be called as "LI program" and has 20 LOC, is shown in Figure 1.

```
1 if(p>q)
2   {if(p>r)
3    {
4  printf("value of integer p is largest in all :%d",p);
5    }
6 else
7   {
8  printf("value of integer r is largest in all :%d",r);
9    }
10    }
11 else
12  {if(r>q)
13   {
14  printf("value of integer r is largest in all :%d",r);
15   }
16   else
17   {
18 printf("value of integer q is largest in all :%d",q);
19   }
20    }
```

Figure 1: LI program

In the Figure1, on the mutated location "p>r" in line no. 2 of LI Program we can create 5 mutants (p<r, p<=r, p==r, p! =r, p>=r) by using different relational operators used in C programming language. Since these 5 mutants are created or mutated at the same location so they are called as same-location mutants (Untch, 2011; Just *et al.,* 2012).

Example of a Test Case Set to Test the LI Program

To test LI program of Figure 1, we can create a test case set which contains 5 test cases, which are given below:

T1: All the three integers are equal (for e.g. (10, 10, 10))

T2: Only two integers are equal (for e.g. (10, 10, 8))

T3: T2 with all permutations (for e.g. (10, 8, 10), (10, 10, 8), (8, 10, 10))

T4: All the three integers have different values (for e.g. (6, 4, 8))

T5: T4 with all permutations (for e.g. (6, 4, 8), (6, 8, 4), (8, 6, 4), (8, 4, 6), (4, 6, 8), (4, 8, 6))

Thus this test case set of LI program has 5 test cases with minimum 10 different test data inputs. We can create some other different test case sets also to test the similar LI program which depends upon the knowledge and experience upon the different testers (Y.S, Ma, 2010; Deng *et al.,* 2013; Kaminski *et al.,* 2013).

Categories of Mutation Operators

Mutation Operator is a rule that is applied to a program for producing the mutants. It is mainly categorized into three categories (Y.S, Ma, 2010; Deng *et al.,* 2013; Untch, 2011).

1) Operand Replacement Operators: Replace a single operand with another operand or constant.
For example:

```
if (a > b)
if (8 > b)      Replacing a by constant 8.
if (a > c)      Replacing b by operand c.
```

2) Operator Modification Operators: Replace an operator or insert new operators.
For example:

```
if (a == b)
if (a >= b)      Replacing == by >=
if (a == ++b)    Inserting ++ after ==
```

3) Statement Modification Operators: Delete or replace some line of codes of the program.
For example:

Statement Deletion (SSDL): delete the else part of the if-else statement.

Statement Replace: replace line no. 3 by a return statement.

Some Conditions for a Test Data to Kill a Mutant

If P denotes a program, M denotes a mutant of P on statement S, and T denotes a test data for P, then following three conditions are satisfied by T to kill M (Abraham, 2017; Souza, 2014; Silva, 2017):

1) Reachability Condition: T must be capable to reach S, if T cannot reach S, it is guaranteed that T will not kill M.

2) Necessity Condition: It is necessary that T must be able to cause M to have a different state from P on S.

3) Sufficiency Condition: The final state of M must be different from that of P.

Effect of multiple test case sets on mutation testing

In the introduction part of this paper, we have developed the LI program which has 20 LOC and produced a test case set containing 5 test cases.

Experimental Setup

In similar manner and controlled environment we constructed total 10 test case sets of different size of LI program to perform mutation testing experiment. The each test case set developed by us was adequate to kill all the mutants generated by mutation operator and SSDL. Out of 10 adequate test case sets the test case set which was the smallest in size contained 3 test cases and the test case set which was the largest in size contained 5 test cases. The values of Median, Mean and Standard Deviation of these 10 adequate test case sets are given in Table 1.

Table1: Sizes of Adequate Test Case Sets of LI Program

Program Name	Min. No. of Test Cases	Median	Mean	Max. No. of Test Cases	Standard Deviation
LI	3	3.0	3.5	5	0.7265

Table 1 shows that the value of Standard Deviation of the size of these 10 test case sets was 0.7265 which was very less.

Result

We performed similar type of experiments on another 30 different programs and calculated the average value of Standard Deviations of all 31 programs equal to 0.2125 which was also very less.

Conclusion and Recommendation

From our experiments, we found that there was very little difference between the minimum and maximum number of test cases when averaged over all 31 programs. As well as the average value of standard deviations of all 31 programs was 0.2125 only. So we can say that differences in test case set sizes are not significantly correlated with program size. These experiments indicated that the practice of using multiple test case sets is not necessary to perform the mutation testing so the use of only one adequate test case set is sufficient.

We recommend that use multiple test case sets for mutation testing if only a few subjects are selected, but if many subjects are selected, then the use of multiple test case sets may not increase the accuracy of our experimental works and does not reduces the cost and time of mutation testing.

Effect of Reduced Test Data of a Test Case Set on Mutation Testing

We observed that it is uncontrollable to generate the test data which can definitely meet the sufficiency condition, so we used the reachability conditions and the necessity condition to generate test data which can kill multiple mutants created or mutated at the same location. To show the effect of reduced test data of a test case on mutation testing in a very systematic way we propose a new approach.

Proposed Approach

According to our approach firstly obtain the reachability conditions and necessity conditions of each mutant that can be generated by Relational Operator Replacement (ROR) of C programming language. Then some necessity conditions of same-location mutants are combined into one condition. Finally the reduced test data is generated by combining and forming a program path according to the original program, reachability conditions of these mutants, and the combined necessity conditions.

Example of the Combined Necessity Conditions

Let "a" and "b" are two integers. By applying ROR, relational expression "a>b" can be mutated to a>=b, a==b, a<b, a<=b and a! =b. The necessity conditions of these five same-location mutants will be:

1) (a>b)! = (a>=b)

2) (a>b)! = (a==b)

3) (a>b)! = (a<b)

4) (a>b)! = (a<=b)

5) (a>b)! = (a! =b)

These five conditions can be reduced to:

1') a==b

2') a>=b

3') a! =b

4') True

5') a<b

After more reduction, and combining these conditions, we will get the combined necessity conditions as (a==b, a<b).

Similarly relational expressions a<=b, a<b, a==b, a>=b, and a! =b all has the similar types of results. In

Table 2, the combined necessity conditions of each relational expression are given.

Table 2: Combined Necessity Conditions of Relational Expression

Relational Expression	**Combined Necessity Conditions of Relational Expression**
a<b	a==b, a>b
a<=b	a<b, a>b
a==b	a<b, a>b
a>=b	a<b, a>b
a>b	a<b, a==b
a!=b	a<b, a>b

Experimental Setup

To perform this experiment we selected the same LI program and *at location "p>r", we produced 5 mutants (p<r, p<=r, p==r, p! =r, p>=r) by applying ROR.* As (p>q) represents the branch predicate then *reachability condition of "p>r" will be:*

p>q

In case of relational expression *"p>r" the combined necessity conditions will be (p<r, p==r).* Finally the desire condition of test data to kill multiple mutants that are mutated at the location "p>r" is obtained as:

(p>q) && {(p<r) && (p==r)}

= }(p>q) && (p<r)} && {(p>q) && (p==r)}

=]}(r>p>q)} && {(p==r)>q}].

In this experiment, we took the test case set of LI program which is described in the introduction part of this paper. So on applying condition [{(r>p>q)}&& {(p==r)>q}] on the test data inputs of this test case set, we found the reduced test data as [(6, 4, 8) && (10, 8, 10)]. Table 3 shows, how only these two test data [(6, 4, 8) && (10, 8, 10)] can kill all five same-location multiple mutants that are mutated at location "p>r" of LI program.

Results

In Table 3, "0" represents that the particular mutant has generated but output of the LI program is same so mutant is not killed by corresponding test data and result "1" represents that the particular mutant has generated and output of the LI program is different so mutant is killed by corresponding test data. In fourth column "Yes" shows that particular mutant is killed by either first input data or second input data or by both.

Table 3: Killing of mutation by reduced test data of a test case set

Mutant of "p>r"	**First input data (6, 4, 8)**	**Second input data (10, 8, 10)**	**Mutant killed**
p<r	1	0	Yes
p<=r	1	1	Yes
p==r	0	1	Yes
p!=r	1	0	Yes
p>=r	0	1	Yes

We performed similar type of experiments on another 30 different programs and found that it is possible to reduce the size of test data used of every program.

Conclusion and Recommendation

By applying our approach, lesser number of test data is required to kill all the mutants and we can reduce the size of test data of any given test case set. If reduced size of test data of a test case set is used then cost and required labor of mutation testing will be very less. So we recommend that in mutation testing, we should not have the practice of producing and applying multiple test case sets to test a given program but we should have the practice of reducing the size of test data input of any test case set of the program.

References

Abraham, R., Erwig, M. 2017. Mutation Operators for Spreadsheets. *IEEE Transactions on Software Engineering*, 35(1), 94– 108.

Deng, L., Offutt., J., Li, N. 2013. Empirical evaluation of the statement deletion mutation operator. *6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), Luxembourg*.

Just, R., Kapfhammer, G. M., Schweiggert, F. 2012. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? *Eighth Workshop on Mutation Analysis (IEEE Mutation 2012)*, Montreal, Canada.

Kaminski, G., Ammann, P., Jeff, Offutt. 2013. Improving logic based testing. *Journal of Systems and Software*, 86:2002–2012.

Ma, Y.S., Offutt, J., Kwon, Y.R. 2010. MuJava : An automated class mutation system. *Software Testing, Verification, and Reliability*, Wiley, 15 (2), 97–133.

Papadakis, M., Malevris, N. 2018. Searching and Generating Test Inputs for Mutation Testing. *Springer Plus*.

Silva, R. A., do Rocio Senger de Souza, S., de Souza, P.S.L. 2017. A systematic review on search based mutation testing. *Information and Software Technology.81:19-35.*

Souza, F., Papadakis, M., Durelli, V.H., Delamaro, M.E. 2014. Test data generation techniques for mutation testing: A systematic mapping, *Proceedings of the 11th ESELAW*, 1–14.

Untch, R. 2011. On reduced neighborhood mutation analysis using a single mutagenic operator. *ACM Southeast Regional Conference*, Clemson SC, 19–21.

Yue, J., Mark, Harman. 2015. Constructing subtle faults using higher order mutation testing. *In 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 249–258, Beijing.