Using Higher Order Mutation for Reducing Equivalent Mutants in Mutation Testing

Aderonke Olusola Akinde*

Department of Computer Science, Babcock University, Nigeria.

ARTICLE INFO

Corresponding Author:
Aderonke Olusola Akinde,
Department of Computer Science,
Babcock University, Nigeria.

KeyWords: Mutation Testing, Mutation Order, Higher Order Mutants, Equivalent Mutants, Equivalence Ratio

ABSTRACT

Customarily, first order mutants are used for mutation testing, but the first order mutants have proved to cost a lot computationally and in terms of human effort over the years. This is mainly as a result of the very large number of mutants that will be generated and the large proportion of equivalent mutants. This has hindered the software industry at large from applying it at the testing phase of Software Development Life Cycle, SDLC, thereby forfeiting the advantages that can be derived when mutation testing is employed. Various techniques have been introduced by researchers in order to curb the setbacks. Higher order mutation testing was later introduced, and it was proposed to be more effective in dealing with the mutation testing challenge. Although higher order mutation generates more numbers of mutants than first order mutation, the number of equivalent mutants produced is largely reduced. This paper reports the result of an empirical study for validating if truly higher order mutation reduces the number of equivalent mutants generated depending on the order of mutation by comparing the degree of equivalent in the second order and random order mutation. The empirical analysis made use of 2 benchmark programs and 1 self-written program and the percentage equivalence of second order and random order mutants were analyzed using the equivalence ratio, ER. A huge reduction in the number of equivalent mutants was realized as the order of mutation increased.

INTRODUCTION

Background

The ordering of mutants technique to mutation testing, promises to contribute three things to mutation testing: increased subtlety, reduced test effort and reduced number of equivalent mutants, compared to when FOMs are used in traditional mutation testing.[1][2] One can say that all three contributions proposed by higher order mutation testing can be reflected from the ratio of equivalent mutants produced from the test. In other words, a valuable reduction in the number of equivalent mutants show us that the faults depicted by the mutants are more subtle, and also that test effort can be minimized since we have a reduced number of equivalent mutants, hence a reduction in the oracle(human) cost for examining equivalent mutants. Hence, there is a need to empirically confirm the degree to which these contributions of higher order mutation are true. This work empirically confirms one of the highlighted advantages of HOMs which is the reduced number of equivalent mutants. It confirms the argument by examining the degree of occurrence of equivalent mutants in second order and random order mutation testing.

Mutation Testing

Mutation testing was initially proposed by DeMillo et al and Hamlet.[2] It is a white box error-based software testing technique. “Error-based” in the sense that it involves the introduction of faults (or errors) into the underlying code, either by manually seeding the faults or automated generation of the faults. It is a deliberate attempt to introduce faults into a program and determining if our set of test cases will identify these faults.[3] Unlike other techniques of software testing where they aim at the correctness of the functionality of the program under test, the aim of mutation testing is to achieve a test effective set of test cases rather than attempting to find all the faults in a program. Mutation testing can also be used in test data generation, reducing size of test set and to compare techniques of verification. Mutation testing has been empirically proven to be the most efficient in detecting faults.[4] It is based on 2 basic assumptions:

1. The Competent Programmer Hypothesis: assumption that programs under test are written by a competent programmer, and as such are nearly correct, could only contain slight faults. [5]
1. The Mutation Testing Process

Mutation testing deliberately creates several slightly different versions of the program under test. These slightly different versions of the program are referred to as mutants. Mutants are created by applying mutation operators on the program under test. Mutant operators are set of rules that govern the generation of mutants from the program under test, they create syntactic changes to the program under test; these syntactic changes represent possible mistakes from a competent programmer. Examples of mutation operators for imperative languages include statement deletion, replacing each Boolean sub-expression with true and false, replacing each arithmetic operation with another one, e.g. + with *, - and /, replacing each Boolean relation with another one, e.g. > with >=, == and <=, replacing each variable with another variable declared in the same scope (variable types should be the same) The different categories of mutation operators are: statement mutations, operator mutations, constant mutation and variable mutation. The program under test and its mutants are executed, passing each test case in the test set as input to the programs. The output generated by each mutant is then compared with the output generated by the original program; this information tells if the test case identifies the fault introduced by the mutant or not. This kick off the mutation testing process, there is more to be gathered from the results. If we have different results, then we have killed the mutant – implying that the test case identified the fault introduced. If we have the same result, then we have a live mutant. In this case, two facts can be deduced:

1. Test case T is not good enough
   i. Perhaps the test case did not even cover the line of code where mutation was made.
   ii. In this case we need to come up with a better test case and re-do the process until we have a test set that can possibly kill almost all the mutants.

2. P and P' are equivalent mutants: If we keep improving the test set and still no test case can be devised to kill P', then P' is said to be a possible equivalent mutant. An equivalent mutant is one which is syntactically different from the program under test but has the same behavior as the program (semantically equivalent), that is, though their content differs, they generate equivalent output. [2] [6]

Challenges of Mutation Testing

First, is the problem of handling the large number of mutants produced which poses high demand on computational resources and also human resources, even very small programs generate high number of mutants. Second, is the problem of detecting equivalent mutants for manual detection of equivalent mutants is tedious, There are existing solutions to reduce these limitations. It is computationally expensive to run all mutants against all test cases and all mutants are not always relevant. The equivalent mutant problem is encountered in the difficulty of identifying equivalent mutants, for manual detection is very tedious. Thirdly, is the problem of whether mutants do or can simulate real faults, that is, have we truly found a large proportion of any real faults present even if we have killed all the killable mutants? [7] The major difficulty in the issue of the equivalent mutants and its undecidability. [2] As a result of these challenges, implementation of mutation testing in the industry is rare since high computational cost and human effort is required. Several approaches introduced to provide considerable solutions are:

1. The Do Fewer Approach: aims to consider fewer mutants for the mutation test, without losing relevant information. The do-fewer approaches include: selective mutation – where number of mutants produced are reduced by skipping some mutation operators; mutant sampling – here subsets of mutants are selected randomly either by using samples of some fixed size, or until sufficient evidence has been collected indicating that a statistically appropriate sample size has been reached. [7][8]

2. The Do Smarter Approach: used to share execution of mutants over several available machines. The do-smarter approaches include: weak mutation – which compares the state of the mutated portion of mutants with the original program to conclude if the mutant is live or killed[9]; distributed architecture - mutants are individually executed on different machines without any dependence on other mutants [1][2]

3. The Do Faster Approach: which speed up the generation and execution of the mutants. The do-faster approaches include: separate compilation – each mutant is generated individually, compiled and run independently[10]; schema based mutation analysis – a schema of all the mutants is generated and a meta-mutant of the mutants is then created and executed once in one program execution environment, hereby saving the cost involved in interpreting and executing the different mutants[2]

Higher Order Mutation Testing

Higher order mutation testing, also referred to as HOM testing is a new form of mutation testing that makes use of Higher Order Mutants. [2] The order of mutants used may differ; it may be 2nd order mutants (mutation operator applied twice), third order mutants (mutation operator applied three times) and so on. The number of HOMs can be calculated from the number of FOMs since HOMs are gotten by combining FOMs. If we have n numbers of FOMs, then we would have n^n HOMs. To uniquely identify an HOM, two values have to be supplied; the position of mutation and the mutant operator that will be applied. [1][11].

Generating HOMs appears to further complicate the mutation testing problems, but HOMs turn out to have some advantages over using FOMs for mutation testing because any test case that kills an HOM will definitely kill its constituent FOMs. The advantages are increased level of subtlety in mutants,

Table 1: Sample higher order mutant; HOM is a 2nd Order Mutant, it combines two FOMs

<table>
<thead>
<tr>
<th>Program P</th>
<th>FOM 1</th>
<th>FOM 2</th>
<th>HOM (2nd Order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a == 1 &amp;&amp; a + b &gt; c) then</td>
<td>if (a == 1 &amp;&amp; a + b &gt; c) then</td>
<td>if (a == 1 &amp;&amp; a + b &gt; c) then</td>
<td></td>
</tr>
</tbody>
</table>

2. The Coupling Effect Hypothesis: assumption that complex faults are coupled with simple faults, such that if a test set detects the simple faults it will also detect the complex faults. [5]
reduced testing effort and reduction in the number of equivalent mutants. [1]

**RELATED WORKS**

**Constructing Subtle Faults Using Higher Order Mutation Testing:** This paper was published by Y. Jia and M. Harman in 2009. It introduces higher order mutation testing, classification of the HOMs into their different kinds in terms of their manner of subsumption and coupling. It also introduces a search based optimization approach for overcoming the destructively huge number of HOMs that can be produced from FOM (n^n). It reports the empirical study of subsuming HOMs using six (6) programs. The algorithms used in the search based identification of the subsuming HOMs from the population are greedy algorithm, genetic algorithm and hill climbing algorithm. The six categories of HOMs presented are: strongly subsuming and coupled, weakly subsuming and coupled, weakly subsuming and de-coupled, non-subsuming and de-coupled, non-subsuming and de-coupled(equivalent), and non-subsuming and coupled(useless). [2]

![HOMs Classification Diagram](image)

Using the coupling effect hypothesis, if a test set that kill the FOMs contain a test case that kill the HOM, then we have a ‘coupled HOM’, if the test set does not contain a test case that kills the HOM then we have a ‘de-coupled HOM’. A Subsuming HOM is more difficult to kill than its constituent FOMs. If a test case kills a strongly subsuming HOM, the test case is guaranteed to kill its FOMs.

**Decreasing the cost of Mutation Testing with Second order mutants:** This paper was published by M. Polo M. Piattini and I. Garcia-Rodriguez in 2009. It presented the concept of decreasing the cost of mutation testing by combining first-order mutants. It reports that the number of mutants can be reduced to half the original suite in one of the experiments. Three combination strategies were implemented as part of the testooj tool: LastToFirst, DifferentOperators and RandomMix, and all 3 algorithms were found to significantly reduce the number of equivalent mutants. LastToFirst and RandomMix reduced the number of mutants by 50%, while Different-Operators reduce the number of mutants to 40%. Additionally, an analysis of kth-order mutation was carried out based on the data from the experiments. kth-order mutation is cost-effective, although it implies a risk level that should be adequately assessed. [12]

**Test Case Effectiveness of Higher Order Mutation Testing:** This paper was published by S. Kapoor in 2011. It examined the capability of Higher Order Mutation Testing by creating FOMs and resultant subsuming HOMs from a combination of the FOMs. It concludes that “though HOM’s are harder to kill but if we are able to find a subsuming HOM than the number of test cases reduces as these test cases will automatically kill all the FOM’s from which it is constructed thereby leading to reduction in test efforts without loss of effectiveness”. [13]

**METHODOLOGY**

**Empirical Questions**

The questions raised for the execution of this empirical study were centered on the extent to which equivalent mutants’ generation is reduced. Could there be some difference in the performance of mutants with respect to the order of the mutants implemented? Hence the following questions:

1. Are 2nd order mutants likely to result in less equivalence than FOMs?
2. Are random order HOM likely to result in less equivalence than FOMs?

**Subject Programs**

Three (3) programs were selected for the purpose of this study. All programs are coded in C programming language and do not exceed 400 LOCs Two of the three programs were downloaded from SIR (Software-artifact Infrastructure Repository) [15], the third program was self-written in manner that it is susceptible to generating equivalent mutants. The details of the programs are given in Table 2 below:
Method

The tool used in generating all the mutants used for this empirical study is MiLu. A brief description of the MiLu tool is given below:

1. **The MiLu Tool**: A new mutation testing infrastructure developed by Yue Jia, a PhD student at King's College London. MiLu is specially designed for the study of HOMs in C programs, and supports general purpose of mutation testing as well as higher order study. MiLu adopts an approach of efficiency and flexibility; it offers a source code analysis and program testing environment to support full mutation testing with either FOMs, HOMs or both. It also provides room for customized mutation operator. The MiLu tool aims to allow users concentrate on either effective algorithms for generating FOMs and HOMs or to focus on analyzing the results from the experiment. [14] MiLu was used because the subject programs are in C programming language. Also, MiLu can be used to generate first order, 2nd order and random order mutants, giving room to focus on the analysis of the performance of the various categories of mutants. The tool is available on www.dcs.kcl.ac.uk/pg/jiayue/milu/

2. Procedure Implemented

   The method used in this study is stated at this juncture, the procedure is given below:

   1. First order mutants of the three programs were generated using the MiLu tool.
   2. The FOMs for the different subject programs were sampled using a randomly fixed size of 250 (mutant sampling). These mutants were randomly selected without a prior knowledge of their contents.
   3. Each FOM (sampled FOM) was executed manually using a C compiler, with the aim of concluding if the FOM is equivalent or not. This was done by extensively finding a test case that kills the FOM.
   4. Second order and random order mutants were also generated using the MiLu tool. 200 second order and 200 random order HOMs were randomly selected from a pool of 400 mutants for each subject program in two divisions
   5. Each 2nd order mutant and r-order mutant was manually executed using a C compiler, with the aim of concluding if the FOM is equivalent or not. This was also done by extensively finding a test case that kills the mutants.
   6. Test cases for the TCAS program were downloaded from the software repository, but test cases for the TRIANGLE and the TAXLEVELCALC were manually derived. In situations where the mutants proved difficult to kill, the codes were manually looked into to check if a test case can be worked out to kill the mutant or probably the mutant is truly equivalent.
   7. The result of the test was then evaluated for the different categories of mutants.

RESULTS AND DISCUSSION

The result of the test is illustrated here with different statistical charts; pie charts, bar charts, for easier comprehension of what transpired during the mutation testing of the different order of mutants.

1. Overall result of the mutation testing of the subject programs is given in Table 3 below.

<table>
<thead>
<tr>
<th>Mutant Order</th>
<th>Subject Program</th>
<th>No of Mutants Analyzed</th>
<th>No of Equivalent Mutants</th>
<th>Percentage Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Order</td>
<td>Triangle</td>
<td>250</td>
<td>19</td>
<td>7.6%</td>
</tr>
<tr>
<td></td>
<td>TCAS</td>
<td>250</td>
<td>72</td>
<td>28.8%</td>
</tr>
<tr>
<td></td>
<td>TaxLevelCalc</td>
<td>250</td>
<td>53</td>
<td>21.2%</td>
</tr>
<tr>
<td>2nd Order</td>
<td>Triangle</td>
<td>200</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>TCAS</td>
<td>200</td>
<td>7</td>
<td>3.5%</td>
</tr>
<tr>
<td></td>
<td>TaxLevelCalc</td>
<td>200</td>
<td>9</td>
<td>4.5%</td>
</tr>
<tr>
<td>r-Order</td>
<td>Triangle</td>
<td>200</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>TCAS</td>
<td>200</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>TaxLevelCalc</td>
<td>200</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

2. Result from First Order Mutants

   **Figure 2: 1st order mutants’ results for all subject programs**

   ![Figure 2](image)

3. Result from Second Order Mutants

   **Figure 3: 2nd order mutants’ results for all subject programs**

   ![Figure 3](image)
4. Result from Random Order Mutants

\[ \text{Equivalence Ratio, ER (n)} = \frac{\text{Percentage equivalence of the FOM}}{\text{Percentage equivalence of the HOM}} \]

Where \( n = \text{mutation order} \)

1 Answer to 3.1.1 (Are 2\textsuperscript{nd} order mutants likely to be result in less equivalence than FOMs?)

Figure 5: Percentage equivalence in the difference subject program based on the order of mutation

It was found that FOMs are more likely to generate equivalent mutants than 2\textsuperscript{nd} Order mutants, judging from the difference in the percentage equivalence derived from the first and second order mutants of all subject programs. The TRIANGLE program gave an equivalence ratio of 7.6 to 0, TCAS gave a ratio of 28.8 to 3.5 while the TAXLEVELCALC gave a ratio of 21.2 to 4.5 in terms of the percentage equivalence of their First Order Mutants to their 2\textsuperscript{nd} Order mutants. Figure 5 reflects a significant drop in the number of equivalent mutants derived from using second order mutants.

2 Answer to 3.1.2 (Are random order HOMs likely to result in less equivalence than FOMs?)

Similarly, the r-HOMs also proved to be less likely to generate equivalent mutants with a zero percentage equivalence result from all three subject programs. Figure 5 gives a picture of the striking performance from using randomly ordered HOMs for mutation testing. All the three line graphs witnessed a drop in altitude as the mutant order increased.

\[ \text{CONCLUSION} \]

This study reveals and supports large reduction in the number of equivalent mutants generated by higher order mutation. The ordering of mutants approach to reducing mutation testing cost of dealing with numerous equivalent mutants, proved to be true judging from the results obtained. Hence, this study concludes that HOM testing is truly capable of offering the advantage of reduced number of equivalent mutants as proposed in the previous work that presents the higher order mutation testing paradigm.[1][7]

For future work it is recommended that further empirical study to identify the Effect of the choice of mutation operator for C Programs. This involves empirically studying the contribution made by the choice of mutation operator on the percentage equivalence of the different order of mutants considered.

\[ \text{REFERENCES} \]
