Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/080534

Reducing Mutation Testing Endeavor using the Similar Conditions for the same Mutation Operators Occurs at Different Locations

Tannu Singla[1], *Ajay Kumar*[1,*] *and Sunita Garhwal*[2]

[1] Department of Computer Science and Engineering, Thapar University, Patiala, 147004, India
[2] School of Mathematics and Computer Applications, Thapar University, Patiala, 147004, India

Received: 6 Sep. 2013, Revised: 4 Dec. 2013, Accepted: 5 Dec. 2013
Published online: 1 Sep. 2014

Abstract: Mutation testing is a software testing technique that ameliorates the quality and reliability of critical software. This paper presents a mutation testing technique based on the concept of the same mutation operator under similar conditions occuring at different locations in the program. In the proposed technique, we assemble the tantamount behaviour of mutants under a group and a single mutant is selected from the group for performing mutation testing. The benefits of the proposed approach are a reduction in time, cost and effort.

Keywords: Mutation Testing, Mutation Score, Mutation Operators.

1 Introduction

Software testing is an essential activity in the software development life cycle; it assists in ameliorating the level of confidence in the correctness and reliability of the software under testing. Mutation testing is a white box fault-based testing technique. It works in conjunction with the traditional testing techniques. The main goals of mutation testing are to provide a test adequacy criterion and diagnose faults in the program under testing. It promises to ameliorate the quality of software. There are a number of problems associated with mutation testing, such as the large time consumption, the equivalent mutant problem and the large effort consumption. A number of techniques [1,2,3,4,5,6,7,8] have been proposed for reducing effort and finding equivalent mutants. Offutt and Lee [1] proposed an approach using the concept of weak mutation to significantly reduce computations and increase the effectiveness of mutation testing but it is not appropriate for programs having large lines of code. Umar and Harman [9] provided a detailed description of method and class level mutants, outlining information related to the chances of detection for every mutant type.

Offutt and Pan [2] solved the feasible path problem using the mathematical constraint system; the approach helps in detecting the infeasible constraints and recapturing them before applying testing. The mathematical constraints are developed using the method level mutants [2]. The technique [2] is applicable to a program having large lines of code as it works on the mathematical constraints. Offutt and Craft [7] optimize the code by removing dead code, invariant propagation, constant propagations, hoisting and sinking, loop invariant detection and common sub-expressions. Demillo and Offutt [6] also resolved the feasible path problem by using the constraint-based testing (CBT) technique and proposed a CBT algorithm for the detection of equivalent mutants.

The paper is organized as follows: section 1 contains the introduction with some preliminary concepts of mutation testing; section 2 describes the proposed approach and experimental results; section 3 contains comparisons between the proposed approach and the existing approaches, and finally section 4 contain conclusions and future scope.

* Corresponding author e-mail: ajayloura@gmail.com

1.1 Mutation Testing

The speculating faults are introduced in the program code, causing meager changes in the original program. These meager changes will lead to a divergence in the program called mutants. These meager changes are performed with the help of mutation operators introduced in the program under testing. The mutated program, which is obtained by applying only one mutant in the original program, is called a *first-order* mutant. The mutated program, which is obtained by applying more than one mutant operator in the original program, is called a *higher-order* mutant. Examples of mutation operators for imperative languages include statement insertion or deletion and the replacement of each arithmetic operator with another one, e.g. + with *, - or /. The different categories of mutation operators are statement mutations, operator mutations, constant mutation and variable mutation [10]. Fig.1 illustrates the concept of mutation testing in which mutated program contains one mutant / in place of % in the original program.

Original Program

```
Public static int gcd( p, q)
{
 while(q!=0)
   {
    int temp;
    temp=q;
    q=p%q;
    p=temp;
  }
return p;
}
```

Mutated Program

```
Public static int gcd( p, q)
{
  while(q!=0)
   {
     int temp;
     temp=q;
     *  q=p/q;
      p=temp;
   }
  return p;
  }
```

Fig. 1: Function gcd and its Mutated Program

The underlying concept behind mutation testing is to detect mutants using the available test suite. For this purpose, the mutants are executed with the available test suite; On execution, the mutant is required to produce a different result from the original one for at least one of the test cases. Such a mutant is called a killed or detected mutant. If the mutant is not detected, it is called a live or alive mutant. If the mutant is semantically equivalent to the original program, the mutant is called an equivalent mutant. Mutation testing [11] works on the assumptions of competent programmer and coupling effect hypothesis. Competent programmer Hypothesis [12] states that the programmer develops program that is close to the correct version. Coupling effect [12] states that "Test cases that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors".

To measure the dexterity of test suites, a mutation score is calculated. The mutation score is calculated using the ratio of the number of killed mutants over the number of non-equivalent mutants. The mutation score lies between 0 and 1. A mutation score of 1 implies that all the mutants were successfully detected. The aim of this paper is to reduce the effort and time consumption involved in the execution of each and every mutant.

2 Mutation Testing based on Similar Behavior

In the proposed technique, groups of mutants having mutation operators at different locations are created under similar conditions in a program code. Mutants under similar conditions having same mutant operators are grouped in one group; for example, a variable in a print statement and a return statement behaves similarly but behaves differently in an arithmetic statement and a conditional statement.

Input: *Original Program and Test Suite*
Output: Mutation Score
initialization;

1. Remove the infeasible path and dead code from the code
2. Generate a set of mutants using all constraints and necessary conditions then
   - Analyze mutant domain
   - Create G_i groups where i=1 to i=n. Each group is having N_j mutants where j=1 to j=m
   - Killed_Mutant$\leftarrow$ Live_Mutant $\leftarrow 0$, j $\leftarrow 1$
3. **for** $(p = 1$ *to* $p = n)$ **do**
   for *each test case from test suite* **do**
   Apply current test case on Mutant N_j of G_p
   if *(result=killed)* **then**
   ▷ Killed Mutant
   Killed_Mutant$\leftarrow$ Killed_Mutant+N_m
   else
   ▷ Live Mutant
   Live_Mutant$\leftarrow$ Live_Mutant+N_m
   end
   end
   end
4. **if** *results are not satisfactory* **then**
   if $j < m$ **then**
   $j \leftarrow j+1$ Go to step 3
   end
   else
   Killed_Mutant $\leftarrow$ Killed_mutant/j
   Live_Mutant $\leftarrow$ Live_Mutant/j
   end
5. Mutation_score$\leftarrow$Killed_Mutant/Live_Mutant

Algorithm 1: Similar Behaviour Mutation Testing (SBMT) Algorithm

One group may contain all the mutants formed using a mutation operator on the variables which are present in

Table 1: Mutation Testing Results using SBMT Algorithm

Program	Executed Mutants	Killed Mutants	Approx. Killed Mutants	Total % of executed	Total % of Killed
Quad	192	152	277	53.185	100
	297	227	275	77.285	99.27
	361	277	277	100	100
Insert	173	138	277	47.658	98.16
	281	218	271	77.41	99.63
	344	262	272	94.76	100
	363	272	272	100	100
Warshall	94	74	252	30.61	93.33
	152	120	251	49.61	93.64
	210	159	234	68.40	99.15
	268	205	236	87.29	100
	307	236	236	100	100
Bsearch	139	95	217	46.33	96.875
	212	151	227	70.66	98.66
	262	193	227	86.66	98.66
	300	224	224	100	100
Bub	151	124	244	51.01	98.75
	219	179	242	73.98	99.58
	269	219	242	90.87	99.58
	296	243	241	100	100
Trishmall	195	127	241	49.74	98.36
	355	231	245	90.56	100
	382	245	245	97.44	100
	386	245	245	98.46	100
	390	245	245	99.48	100
	392	245	245	100	100
Mid	64	51	140	35.35	98.55
	128	100	138	70.71	100
	154	120	138	85.08	100
	163	126	138	90.05	100
	172	132	138	95.02	100
	181	138	138	100	100
Euclid	99	69	110	62.65	98.214
	132	92	111	83.54	99.107
	158	112	112	100	100
Pat	68	54	106	53.125	100
	128	106	106	100	100

print and return statements. Similarly, another group may contain all the mutants formed using a mutation operator on the variables which are present in all conditional statements. We can execute only one mutant from each group.

The result of single mutant whether killed or live is assumed to be same for all other mutants in its group. These results are used to calculate the overall results for the entire program code. The calculation is done by multiplying the number of mutants present in the group and then adding together all group results. Using the proposed technique, similar results to the existing approaches are obtained by executing lesser mutants. Additionally, the proposed approach requires less effort and time consumption.

Executed on a single mutant from each group, the SBMT algorithm will produce results similar to the actual results. This algorithm works in conjunction with the traditional algorithms and reduces the effort required for performing mutation testing.

2.1 Experimental Results

The proposed algorithm is applied on nine Java programs [1] that cover distinct types of applications. The code length of these Java programs varies from a small number of statements to a large number of statements. All possible method level and class level type mutants for Java are created in these nine programs. Method level operators are arithmetic operators, conditional operators, shift operators, logic operators, relational operators and assignment operators, and class level operators are applied on object-oriented features such as encapsulation,

polymorphism, access modifiers, inheritance and Java-specific features [9]. All mutants of the nine programs are executed manually using the SBMT algorithm to determine the true results. Thus, the probability of mistakes is quite rare.

In table 1, details of the experiment results are reported. The approximate true results of the programs are calculated for the execution of three mutants from each group. The number of mutant groups for a program varies from program to program depending upon many factors. The large number of mutants in a group has no adverse effect on the results, rather it helps in reducing the effort of calculation.

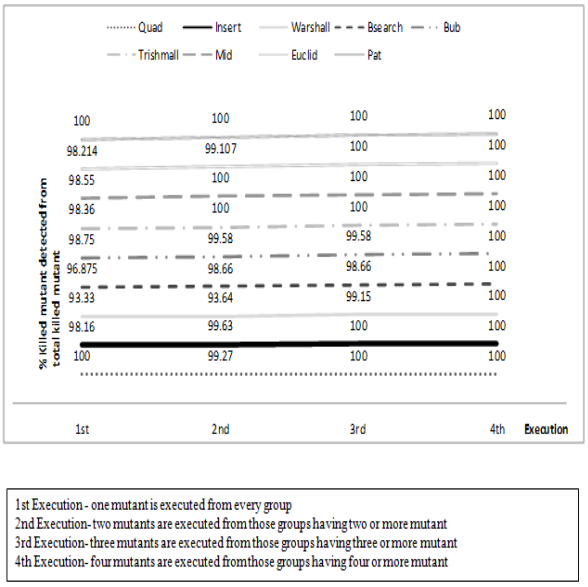

Fig. 2: Execution of number of mutants verses mutant detected out from the total killed mutants

In fig.2, every program delineates the execution of the number of mutants from their groups and the corresponding percentage of detected killed mutants from total killed mutants for the entire program code. Fig.2 illustrates that the expected results are procured by considering only four mutants from each group of every program. Hence, if any group has a mutant count of more than four, there is no need to execute those excess mutants. With the execution of a single mutant from every group, the results are more than 93 percent close to required results and the process requires less than 60 percent of effort and time.

3 Comparison with Existing Approaches

Table 2, illustrate the comparisons of the existing techniques [1, 2] with the proposed (SBMT) technique.

Table 2: Comparison between SBMT and Existing Techniques

Technique	Class Level Mutants	Method Level Mutants	Applicable for lengthy code
Offutt and Pan Proposed Technique[2]	No	Yes	Yes
Offutt and Lee Proposed Technique[1]	Yes	Yes	No
SBMT	Yes	Yes	Yes

The proposed approach is applicable to both method and class level mutants. This approach is also applicable to programs having large lines of code. The major benefits of the proposed approach are the reduction in effort, time and cost consumption.

4 Conclusions and Future Work

The proposed approach is inexpensive for mutation testing. The SBMT algorithm works for mutants having the same mutation operator at different locations in a program under similar conditions, by accumulating them in a group. The proposed algorithm covers all types of Java mutants and can be applied to programs of any length. The SBMT algorithm helps in reducing the time, effort and cost consumption to 60 percent. This technique has a limitation in that it is not able to work on the subgroup of a group. The work can be extended on subgroups and an automated system for the formation of groups can be created and integrated with SBMT.

Acknowledgement

The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

[1] A. J. Offutt and S. D. Lee, An empirical evaluation of weak Mutation, *IEEE Transactions on Software Engineering*, **20**, 337-344 (1994).

[2] A. J. Offutt and J. Pan, Detecting Equivalent Mutants and the Feasible Paths, *Software Testing, Verification, and Reliability*, **7**, 165-192 (1997).

[3] B. J. M. Grun, D. Schuler and A. Zeller, The Impact of Equivalent Mutants, *IEEE International Conference on Software Testing, Verification and Validation Workshops*, 192-199 (2009).

[4] Y. Jia and M. Harman, Constructing subtle faults using higher order mutation testing, *IEEE International Working Conference on Source Code Analysis and Manipulation*, 249-258 (2008).

[5] P. G. Frankl, S. N. Weiss and C. Hu, All-Uses vs Mutation Testing: An experimental comparison of effectiveness, *The Journal of Systems and Software*, **38**, 235-253 (1997).
[6] R. A. DeMillo and A. J. Offutt, Constraint-based automatic test data generation, *IEEE Transactions on Software Engineering*, **17**, 900-910 (1991).
[7] A. J. Offutt and W. M. Craft, Using Compiler Optimization Techniques to Detect Equivalent Mutants, *The Journal of Software Testing, Verification and Reliability*, **4**, 131-154 (1996).
[8] T. Singla and A. Kumar, Mutation Operators corresponding Conditions Contributing in Deporting them Equivalently, *International Journal of Computer Science and Technology*, **4**, 656-658 (2013).
[9] M. Umar, An Evaluation of Mutation Operators for Equivalent Mutants, *M.S. Thesis, King's College, London*, (2006).
[10] K. Kapoor and J. P Bowen, Ordering Mutants to Minimise Test Effort in Mutation Testing, *Proceedings of the 4th International conference on Formal Approaches to Software Testing (FATES) Lecture Notes in Computer Science*, **3395**, 195-209 (2005).
[11] Y. Jia and M. Harman, Higher Order Mutation Testing, *Information and Software Technology*, **51**, 1379-1393 (2009).
[12] R. A. DeMillo, R. J. Lipton and F. G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *Computer*, **11**, 34-41 (1978).

Tannu Singla received her B.Tech. degree in Computer Science and Engineering from RIMT-IET college, PTU University, Jalandhar, India in 2011. She is pursuing her M.Tech. in Software Engineering from Thapar University, Patiala, Punjab, India. Her research interests include Software Testing and Software Engineering.

Ajay Kumar is an Assistant Professor at Computer Science and Engineering Department, Thapar University, Patiala, Punjab, India. He obtained his M. Tech. Computer Science and Engineering from the Kurukshetra University, India in 2004. He received his PhD degree in Theory of Computation from the Computer Science and Engineering Department, Thapar University in 2013. He has ten years of teaching experience in the area of Theory of Computation, Software Testing and Programming Languages. His research interests are Theoretical Computer Science and Software Testing.

Sunita Garhwal is an Assistant Professor at School of Mathematics and Computer Applications, Thapar University, Patiala, Punjab, India. She obtained her M. Tech. in Software Engineering from the Kurukshetra University, India in 2007. She has six years of teaching experience in the area of Automata Theory, Compiler design and Graph theory. Her research interests includes Automata Theory and Software Engineering.