Detecting Equivalent Mutant with Control Flow Graph Coverage and Condition Test for Equivalence Suspected Mutant

Priyanka Gupta[a], Rakesh Kumar[b], Nipur[c]

[a]*Department of Computer Applications, Maharaja Agrasen Institute of Management and Technology Old Saharanpur Road, Agrasen Chowk, Jagadhri, Haryana, India.*

[b]*Department of Computer Science and Applications, Kurukshetra University Kurukshetra, Haryana, India.*

[c]*Department of Computer Science and Applications Kanya Gurukul Mahavidyalaya, Dehradoon, Uttaranchal, India.*

*gupta800@gmail.com

Abstract

Equivalent mutant detection problem is the biggest hurdle in practical usage of mutation analysis in industry. This problem is undecidable, as mutant and its program can't be compared against each possible input exhaustively. This paper proposes a method for detection of equivalent mutant by making use of control flow graph (CFG) coverage of program and its mutant under consideration. The method applies a condition test on path coverage of mutant if it follows a different path as compared to parent program and returns true if equivalence is found. Suggested method is implemented on a case study for evaluation of results.

Keywords - Software Testing, Equivalent Mutant Detection, Control Flow Graph

Introduction

Testing plays a vital role in software quality assurance process. Quality of testing is guided by quality of test cases generated. Exhaustive testing is impractical as resources are always limited. Test adequacy criteria helps in limiting the amount of test cases for the purpose of testing process. Many test adequacy criteria has been found in literature like statement coverage, branch coverage, path coverage, loop count coverage, multiple condition coverage, all definition criteria, all uses criteria etc. (Zhu *et al*., 1997). Each criterion is with some pros and cons. Many statistical and experimental studies have been conducted to compare the effectiveness of different testing strategies (Basili *et al*., 1987; Roper *et al*., 1993; Lott *et al*., 1997 and Kamsties *et al*., 1995). In some situations some criteria overweight others and other times others become better. But no one is 100% effective in process of fault detection and removal. In such situation Mutation adequacy criteria has been suggested as a method that helps in improving the effectiveness and completeness of test cases (Budd *et al*., 1980; Walsh *et al*., 1985; Frankl *et al*., 1997 and Mathur *et al*., 1993).

Mutation-based testing works with a set of operators. Each of the operators modifies the source code as if an error has been injected. The modified program is known as a mutant. Test cases are generated and both mutants and original program are executed against them. If any test case can produce different results then mutant is said to be killed. Otherwise, the mutant is *live*. Either the mutant program is found equivalent to the original program or the test data set is not adequate and needs enhancement. The adequacy of a test data set is measured by a mutation score (MS), that is defined as the ratio of the number

of killed mutants to the total number of non-equivalent mutants (Demillo *et al.*,1978; Budd *et al.*,1980). Although mutation is a promising adequacy criteria in sphere of testing, still it's not without problems (Jia *et al.*,2011).First problem is the very high computational cost of running test set against so many mutants. Even with a very small program plenty of mutants can be generated and all these have to be executed against test set T. Another problem is a common problem with testing practices. And that is of human intervention. A lot of human efforts are required in test oracles. Another major problem is the detection and handling of equivalent mutants. Some methods to overcome all problems of Mutation testing has been found in literature (Jia *et al.*, 2011).

Equivalent mutants and their identification is a big trouble where Human intervention for estimating equivalence is required. It's found that around 8.8% of mutants are equivalent in experiments (Offutt *et al.*, 1993). It adds to cost of mutation testing and has become a hindrance in its practical usage. Unless the detection of all equivalent mutants is done the tester may not have complete confidence in mutation adequacy score, as it's based on non equivalent mutants. Therefore it's mandatory to sort out equivalent mutants. It's found in research that about 10% to 40% of mutants can be equivalent (Offutt *et al.*, 1994; Offutt *et al.*, 1997). Therefore, there is need for the study of detection techniques for equivalent mutants for practical growth of mutation testing and its widespread usage in industry.

A very few methods for its remedy have been found in literature. This paper suggests a new method to tackle this problem with the help of control flow graph notation of software under test.

Equivalent mutants

During mutation analysis process when mutants are generated, they can be categorized into live mutants, dead/ distinguished/extinguished/terminated/killed mutants, and equivalent mutants.

Live mutants are those which are not differentiated by test set T under consideration from the original program P. Dead mutant is one that is differentiated by some test from its original program P and equivalent mutants are those which even after sufficient rounds of test enhancement are still alive and can't be distinguished. Equivalent mutant is a mutant that is considered equivalent to its parent P in sense that for each test input from input domain of P observed behavior of M is identical to P. Equivalent mutant is syntactically different but semantically same with program P.

```
Sample program A1
……
……
For (int i=0; i<10; i++)
{statements here doesn't changes the value of i}
……
Sample mutant M1 of A1
……
……
For (int i=0; i!=10; i++)
{Statements here doesn't changes the value of i}
……
```

Figure 1 : Example equivalent mutant 1 (Jia *et al.*, 2011).
Sample Program A2

```
……
……

if(x == 2 && y == 2)
{z = x + y;}
……
Mutant M2 of A2
……
……
if(x == 2 && y == 2)

{z = x * y;}
……
```

The value of Z will be equal to 4; any test set will be unable to determine any faults with this program because the value will always be equal to 4.

Figure 2 : Example equivalent mutant 2 (Umar *et al*., 2006)

Literature survey

Adampoulos *et al*., (2004) has suggested detection of equivalent mutants with the help of genetic algorithms in co-evolution of mutants and test cases. This technique makes use of genetic algorithms to attain selective mutation without decreasing the mutation operators. This technique identifies the equivalent mutants and the best test cases for killing mutants with a methodology with three steps. First step is for evolution of mutants. Initially a set of mutants are generated and tested against a fixed set of test cases. Results are calculated and a fitness value is assigned based on performance. Higher the number higher the probability of being equivalent mutant and least probability of getting killed by test set. Next level of mutants is generated by making subsets of this level. Each subset represents the individual for the GA and is evaluated in number of steps and finally gets some mutants which can't be killed and are equivalent mutants. At second step similarly test cases are evaluated for best performance and at third level both mutants and test cases are convoluted (Adampoulos *et al*., 2004).

Voas *et al*., (1997) were the first to suggest the application of program slicing to Mutation Testing. In technique suggested by (Hierons *et al*., 1999) program slicing has been used for equivalent mutant detection. They had made use of weak mutation. Weak mutation is that form of mutation analysis where change in state or variable of a program is analyzed just after its execution is done. In program slicing approach slices with the changed code in mutant is identified and effect is studied for each slice. If mutant killed means no equivalence otherwise equivalence in raised. Use of slicing has reduced the job to be carried up manually. Also they have suggested equivalent mutants will create identical slices for all mutants. (Harman *et al*., 2001) extended the work by using dependence analysis. They have defined a framework of dependence analysis that is better than Program Dependence Graph (PDG) approach, used in slicing and other forms of program analysis. They have defined an augmented testing process which starts and ends with dependence analysis phases. Equivalent mutants are removed in pre-analysis phase, and few mutants which are left are handled by human with much lesser efforts afterwards.

Baldwin *et al*., (1979) were the first to suggest detection of EM with the help of compiler optimizations.

Their approach was based on the idea that optimization procedure of source code will produce an equivalent program, therefore a mutant can be detected as EM by either optimization or de-optimization process. They suggested six rules for compiler optimization and were these were empirically implemented (Offutt *et al*., 1994).

Offutt *et al*., (1994) suggested that mutants which are generated by change in dead or irrelevant code will not make any difference in output and will thus be equivalent (dead code method). Dead code is one which is never executed. Also live mutants can be compared with constant table entries. A mutant, for which entry is found, is equivalent. Invariant propagation technique saves the relationship of two variables (invariants) in invariant table. An equivalent mutant will have same definitions in the table. Common sub expressions method keeps check on all temporary variables generated during compilation and keeps a check on equality relationship amongst the variables. In loop invariant method equivalent mutant makes a change in loop by moving it inside or outside. In hoisting and sinking method equivalent mutants will generate same results irrespective of hoisting or sinking of code.

Offutt *et al*., (1997) have suggested some constraints on test cases and suggested these conditions to be satisfied for detecting equivalence between the mutants. For program P and its mutant M, reach ability condition is one where its mandatory that the test case should reach the changed statement in M otherwise no change in results could be found. Necessity condition says that the state of mutant M and Program P should vary after the execution of mutated statement. Sufficiency condition says that P and M should have different final states for the execution of test case. These suggested conditions are used to specify constraints on the program and then used to check for its equivalence.

Ellims *et al*., (2007) said that mutants with syntactic difference and the same output can be also semantically different in terms of running profile. These mutants often have the same output as the original programs but have different execution time or memory usage. According to them resource-aware can be used to kill the potential mutants.

The most recent work on the equivalent mutants was conducted (Grun *et al*., 2009). They investigated the impact of mutants. The impact of a mutant was defined as the different program behavior between the original program and the mutant and it was measured through the code coverage in their experiment. The empirical results suggested that there was a strong correlation between mutant kill ability and its impact on execution, which indicates that if a mutant has higher impact, it is less likely to be equivalent. They said that more a mutation alters the execution; the higher are the chances of it being non-equivalent.

Control flow graphs

Control flow graphs are the most common form of graphs used in software engineering. Given a program P written in imperative language, CFG is a directed graph where nodes represent computations (either statements or fragments of statements) and edges represent flow of control between statements. There is an edge from node i to node j if the statements that represents node j can be executed immediately after statements that represents node i (Jorgensen *et al*., 2012). Each node represents a basic block with a single entry and single exit point.CFG models all program executions. Possible executions are represented by paths in the graph (Pressman *et al*., 2005).

RIP Model: Conditions for killing a mutant

Ammann *et al*., (2013) have proposed three conditions that need to be satisfied for a failure to be observed

for some particular test.

a) Reachability condition says that for a failure to be true the location or locations in the program that contain the fault must be reached only then the fault can get activated to become a failure. In mutation analysis this is the mutated statement.
b) Infection says that the state of the program must become incorrect on execution of faulty code. Test causes the faulty code to result in incorrect state.
c) Propagation says that the infected state must cause output or final state of the program to become incorrect and the infected state leads to incorrect output.

For a failure to be true (a&b&c) needs to be satisfied for a test. When this model is used in mutation analysis this can be used to justify the conditions for killing of a mutant. A mutant whose execution is satisfying (a&b&c) is supposed to be killed otherwise alive. This killing can further be classified as: weak killing and strong killing. Strong killing satisfies all three conditions and generates the final output as erroneous. Weak killing relaxes the propagation condition and in spite of considering final output as point of observation any point after the infected state can be considered as the point of observation.

Proposed Method for Equivalent Mutant Detection

Equivalent mutant detection problem is an undesirable problem. When a program P and its mutant P1 produce same output against the same input, mutant is suspected to be equivalent. But this equivalence is not for sure as it's difficult for a tester to compare a program and all its mutants for equivalence against each possible input exhaustively. In this study author suggests a method based on path coverage of control flow graph for a program (P) and its mutant (P1). When any program is executed with some input data it activates some edges of the CFG. Its execution starts from the start node and is observed at some node which can be called as point of observation.

Mutant P1 of program P is generated by making some minor change in program P motivated by competent programmer hypothesis theory of mutation testing. This small change represents a small probable error which a competent programmer may commit while programming program P. when this mutant is run with the same input I as its Program P and produces the same output O, it's suspected to be equivalent to P, but can't be surely declared as equivalent as not verified against all possible inputs. This equivalence represents that they both differ in their syntax but still semantically are alike. Examining the path coverage of their CFG's can help in decision making for their equivalence. Let path followed by P is represented by p1,p2,p3,p4,p5…..pn, where p1 is the start node and pn is the exit node. Let us suppose that both P and P1 follow the same sub-path p1, p2, p3. But at p3 now they follow different branches. P follows p4, p5, and comes to an end point p6. On other hand P1 follows p4', p5' and joins back at p6. Considering conditions suggested by RIP model, decision for their equivalence can be made. If path followed by mutant satisfies all three conditions successfully it represents failure and hence can be distinguished from its parent hence is not equivalent. Otherwise if conditions are not satisfied by mutant and it follows a different path than its parent in CGF, it represents the equivalence between the program and its mutant.

Node p 3 in Figure 3 represents mutated statement in the mutant which if executed means the control has reached there satisfying first condition of reachability, also the effect of this infection is transparent as node p4'& p5' are activated and state of the system has changed, therefore second condition of infection has also been satisfied and at node p6 if results are considered different, satisfying condition of propagation and it means failure occurs and this is not an equivalent mutant. But if any of three conditions

are not satisfied this is considered to be as equivalent mutant. This suggests a very simple and effective method of detecting equivalent mutants.

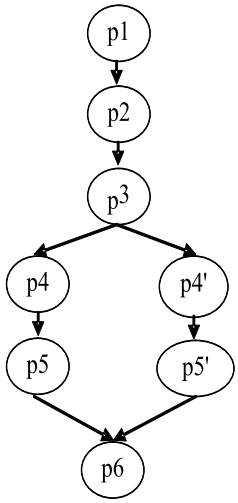

Figure 3 : Control flow graph paths for program and its mutant

Algorithm for the proposed Equivalent Mutant Detection Methodology

This algorithm takes as input a program P and its equivalence suspected mutant P1 and returns true if suspicion is confirmed.

1 Consider a program P and its mutant P1 for consideration of equivalence between them. Let the mutant be called as equivalent suspected mutant.
2 If P1 produces different results for test case as compared with P this represents distinguished/ killed mutant and return false.
3 Else compare CFG coverage for test case execution for both P and P1 for assuring their equivalence.
4 If same path is not followed then mutant qualify for condition check. In condition check evaluation for condition (a&b&c) for mutant P1, where a represents reach ability condition, b represents infection condition and c represents propagation condition has to be done.
5 If condition is satisfied this signifies mutant is distinguished otherwise mutant is declared as equivalent mutant and return true

Implementation of proposed method on case study

Let us consider program P which displays largest of three numbers as a case study program for implementation of proposed methodology for equivalent mutant detection, as represented in Figure 4

```
1  read a, b, c
2  m=a
3  if (b>=m) then
6  m=b
7  if (c>=m) then
8  m=6
```

```
9   write m
10  end
```

Figure 4 : Case Study Program P

A first order mutant P1of program P has been generated by introducing a small error at line no3., represented in Figure 5

```
1   read a, b, c
2   m=a
3   if (b>m) then
4   m=b
5   if (c>=m) then
6   m=6
7   write m
8   end
```

Figure 5 : Mutant P1 of program P

For test caseT {a=10, b=10, c=5}program P produces output=10

For same test case T {a=10, b=10, c=5}mutant P1 produces output=10

Both produces same output for same input. Mutant P1 is suspected for equivalence with program P. control flow graphs for both P and P1 are represented in Figure 6.

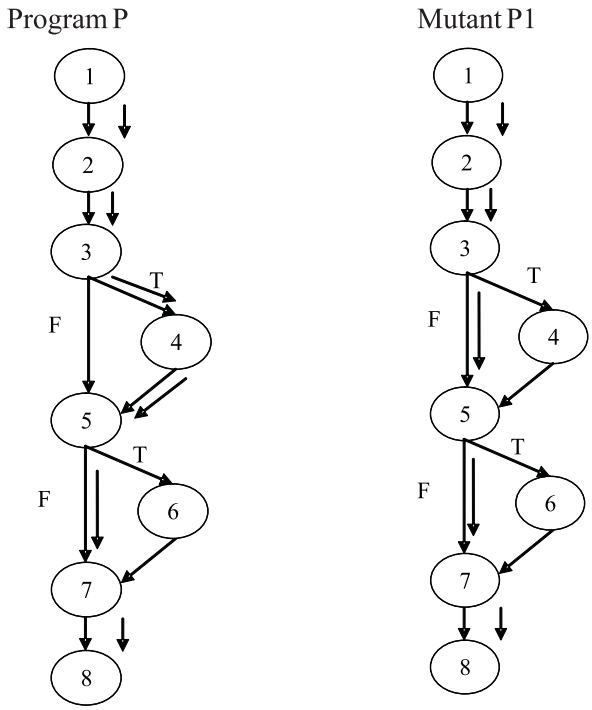

Figure 6 : CFG for Program P and Mutant P1for test case T

For T path followed by P is 1,2,3,4,5,7,8 (1)
For T path followed by P1 is 1,2,3,5,7,8 (2)
Both P and P1 follow up different path which means mutant P1 qualify for condition test.

Mutant was mutated at statement 3 in program P. path coverage of mutant P1 (1,2,3,5,7,8) as shown in equation 2 represents node 3 included which signifies the execution of statement 3 in mutant P1 and hence reach ability condition is satisfied. Under execution of T, P and P1 changes states at node 3 as visible from Figure 6. Program P follows 3, 4, 5 sub path whereas Mutant P1 follows 3, 5 sub-path, which signifies infection. They follow up same sub path thereafter i.e 5, 7, 8 and ends at 8. For evaluation of third condition point of observation could be either node 7 or 8. If node 7 is considered as observation point then it will be treated as weak killing as discussed in section otherwise if node 8 has been considered for result evaluation it is strong killing. In this study strong killing has been done. Results for both P and P1 at node 8 are found to be same, which signifies that infection couldn't be propagated and hence propagation condition failed finally failing condition (a&b&c), which signifies that P and P1 are equivalent and P1 is declared as equivalent mutant of P.

Conclusions and Future work

Suggested method has been successfully implemented on a sample case study. For a test case, equivalence between the parent program and its equivalence suspected mutant has been identified successfully. For automatic implementation of the proposed method adjacency matrix representation of the CFG can be used for detection of equivalence with the parent program.

References

Adamopoulos K., Harman M. and Hierons R. 2004. How To Overcome The Equivalent Mutant Problem And Achieve Tailored Selective Mutation Using Co-Evolution. *AAAI Genetic And Evolutionary Computation Conference 2004 (GECCO 2004), Seattle, Washington, USA. LNCS 3103,* 1338-1349.

Ammann, P., Offutt J. 2013. Introduction to software testing. Second Edition, chapter two, chapter 5.2, *www.cs.gmu.edu/~offutt/softwaretest/.*

Baldwin D., and Sayward G. 1979 Heuristics for determining equivalence of program mutations. *Department of Computer Science, Yale University, New Haven, Connecticut, Tech Report. 276.*

Basili V.R., Selby R.W. 1987. Comparing The Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering,* SE-13(12), 1278-1296.

Budd T., Lipton R., Demillo R. Sayward F. 1980. Theoretical And Empirical Studies on Using Program Mutation to Test The Functional Correctness of Programs. *Proceedings of The Seventh Conference on Principles Of Programming Languages. Las Vegas, ACM Press, 220-233.*

Budd T.A. 1980. Mutation Analysis of Program Test Data. *Ph.D, Yale University, New Haven, CT.*

Demillo R., Lipton R.,Sayward F.1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer Magazine,* 11(4), 34-41.

Ellims M., Ince D.C., Petre M. 2007. The Csaw C Mutation Tool: Initial Results. *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07), published with Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07), Windsor, UK, IEEE Computer Society,* 185-192.

Frankl P.G., Weiss S.N., Hu C. 1997. All -Uses Versus Mutation Testing: An Experimental Comparison of Effectiveness. *Journal of Systems and Software,* 38(1), 235 -253.

Grun B.J.M., Schuler D., Zeller A. 2009. The Impact of Equivalent Mutants. *Proceedings of 4th International Workshop on Mutation Analysis (MUTATION'09), published with Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops, Denver, Colorado, IEEE Computer Society,* 192-199.

Hierons R., Harman M., Danicic S. 1999. Using Program Slicing to Assist in The Detection of Equivalent Mutants. *Journal of Software Testing, Verification and Reliability,* 9(4), 233-262.

Jia Y., Harman M. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering,* 37(5), 649-678.

Jorgensen P.C. 2012. Software Testing: A Craftsman's Approach. *3rd Edition, Auerbach Publications, ISBN-13: 9780849374753.*

Kamsties E., Lott C.M. 1995. An Empirical Evaluation of Three Defect-Detection Techniques. *Proceedings of 5th European Software Engineering Conference,* 362-383.

Lott C.M. and Rombach H.D. 1997. Repeatable Software Engineering Experiments For Comparing Defect-Detection Techniques. *Journal Of Empirical Software Engineering,* 1(3),241-277.

Mathur A.P., and Wong W.E. 1993. Comparing the Fault Detection Effectiveness of Mutation and Data Flow Testing: An Empirical Study. *Technical Report Serc-Tr-146-P, Software Engineering Research Center, Purdue University.*

Morell L. 1990. A Theory of Fault-Based Testing. *IEEE Transactions On Software Engineering,* 16(8), 844-857.

Offutt A.J., Rothermel G., and Zapf C. 1993. An Experimental Evaluation of Selective Mutation. *15th International Conference on Software Engineering, Baltimore Maryland,* USA,100-107.

Offutt, A.J., and Craft, W.M. 1994. Using Compiler Optimization Techniques to Detect Equivalent Mutants Software Testing, Verification and Reliability, 4(3), 131-154.

Offutt, A.J., and Pan, J. 1997. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability,* 7(3), 165-192.

Roper, M., Miller, J., Brooks, A., and Wood M. 1993. Towards The Experimental Evaluation Of *Software Testing Techniques. Technical Report, No. 1.0.*

Umar. M. 2006. An Evaluation of Mutation Operators for Equivalent Mutants. *Master's Thesis, Department of Computer Science, King's College, London.*

Voas, J., and McGraw G. 1997. Software Fault Injection: Inoculating Programs Against Errors. *John Wiley & Sons, New York,* 47-48.

Walsh P.J. 1985. A Measure of Test Case Completeness. *Ph.D, The State University of New York, Binghamton, NY.*

Zhu, H., Hall, P.A.V., and May, J.H.R. 1997. Software Unit Test Coverage And Adequacy. *ACM Computing Surveys,* 29(4), 366-427.