ISSN: 2277-9655
Impact Factor: 4.116

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

A NOVEL APPROACH TO DETECT EQUIVALENT MUTANTS USING CONTROL FLOW GRAPH

Rupinder Kaur*, Sanjay Tyagi
* Research scholar,Department of Computer Science and Applications, Kurukshetra University, Kurukshetra, Haryana-136119
Assistant Professor,Department of Computer Science and Applications, Kurukshetra University, Kurukshetra, Haryana-136119

DOI: 10.5281/zenodo.57991

ABSTRACT

Software testing verifies and validates software systems. Mutation testing is a testing procedure where mutants are created by seeding fault to the code. Test cases are exercised on these mutants to measure the adequacy given by mutation score. If the mutants are not killed, then either the test data is not sufficient or there is the presence of equivalent mutant, which is an undecidable problem. Equivalent mutant problem makes testing process more difficult and tedious. A technique has been proposed in this paper to detect equivalent mutants by observing RIP model and control flow graph of original and mutant program. The approach has also been tested on some sample programs.

KEYWORDS: Control Flow Graph, Equivalent Mutants, Mutants, Mutation Testing, Test Suites, RIP model.

INTRODUCTION

Testing refers to the task of checking out a system or its components in order to identify whether it satisfies the defined requirements or not. Testing when validates and verifies a software system is called as software testing. It identifies errors, mistakes or faults in the source code and provides knowledge about the quality of software under test. Mutation testing is a process in software testing to check whether a test set used to test a given software is adequate or not to reveal defects. Richard Lipton suggested mutation testing in 1971 but later on DeMillo, Lipton and Sayward had published it [1]. It is treated as white box testing or code based technique, as all or some portion of source code is required to access. It has also been used as black box testing. In Mutation testing, different versions called mutants of program are created by seeding artificial faults in the program [2].

Hypotheses related to mutation testing are:

Competent programmer hypothesis

This hypothesis was first presented in 1978 by DeMillo *et al*. [2]. According to this, competent programmers generally commit very small faults so the faulty program is very close to the correct program. The faults committed by competent programmers are small, but those faults can have a huge effect on semantics of program. Therefore, the faults introduced to create the mutant should resemble the real mistakes which are likely to be done by programmer during writing code. So, some syntactic changes are made in the program to create the mutant.

Coupling Effect hypothesis

Another is coupling effect hypothesis that was also suggested in 1978 [2]. In this hypothesis, simple and complex errors were coupled. It states that test suites able to distinguish all mutants created by seeding single defects are so subtle that they can also reveal complex defects. Hence, the test suites adequate in killing first order mutant (i.e. mutants created by inserting one simple fault) are good enough to kill higher order mutants (i.e. the mutant generated by inserting more than one faults). So there is no need to generate higher order mutants.

ISSN: 2277-9655
Impact Factor: 4.116

Mutation testing provides mutation adequacy score that estimates the efficiency of test data. Mutation adequacy score [3] is given by-

Mutation Adequacy Score (MS) = $\frac{D}{M-E}$ (1)

Here, D- no. of dead mutants,
M- Total no. of mutants,
E- no. of equivalent mutant.

Mutation adequacy score results between 0.0 and 1.0. More the adequacy score, more adequate is the test case.

PROBLEM FORMULATION

In mutation testing, mutants of program are created. Then a test data from a test set is exercised on main as well as mutant program.

- If the outputs of both the programs are different, then the mutant is distinguished by test input and the test case is adequate.
- If the output of both the programs are same for every test input in a test set then the reason can be (a) The test set is unadequate to kill the mutant and may be killed by improving the test inputs, or (b) The mutant is semantically same to original program i.e. equivalent mutant. Equivalent mutant is a program that is syntactically dissimilar but semantically similar to original program.

In mutation testing, test cases might fail to detect mutants due to following reasons [4]

i. The mutation has created an equivalent mutant and thus remains undetected.
ii. RIP model is not satisfied by the test data.
iii. Test set is not adequate.

Equivalent mutant problem (EMP) is a major obstacle in mutation testing. An equivalent mutant is one which changes the original program in such a way that the semantics remain unchanged. There exists no test data which can identify difference among equivalent and original code.

It had been empirically found that 15 minutes were taken to find an equivalent mutant [4]. Non detection of equivalent mutant will never lead to 100% mutation score. Manual detection of equivalent mutant is time consuming, thus making the cost of mutation testing high [5]. In the following example, the program P and its faulty version M are equivalent as they produce same output for all possible inputs.

Original Program(P)	Mutant(M)

midvalue=c;
If(a>b)
If(b>c)
midvalue=b;
Else if(c>a)
midvalue=a;
Endif
Else
If(b<c)
midvalue=b;
Else if(c<a)
midvalue=a;
Endif
Endif
Return(midvalue);
End
	midvalue=c;
	If(a>=b)
	If(b>c)
	midvalue=b;
	Else if(c>a)
	midvalue=a;
	Endif
	Else
	If(b<c)
	midvalue=b;
	Else if(c<a)
	midvalue=a;
	Endif
	Endif
	Return(midvalue);
	End

Figure1.Equivalent mutant

Equivalent mutant is an undecidable problem. The relationship between equivalence and test data generation had been examined by Budd and Angluin [6]. They proved that if equivalence between two programs is computed by a process, then adequate test data is also generated by a process and vice- versa. It is showed that no such assessable procedures exist. Hence, a fully automated result to equivalence problem is not possible. So, equivalence detection either between two programs or two mutants is not decidable. Some of mutants always remain hidden by an automated process, due to its undecidability [6].

RELATED WORK

Many efforts had been done before to handle equivalent mutants. Various techniques and heuristics have been introduced by many scholars. Baldwin and Sayward suggested the idea of using optimization methods of compilers to identify equivalent versions of a code [7]. The notion behind is that optimization and de-optimization leads to the creation of semantically similar mutants. These compiler optimization techniques were implemented and were able to detect approximately 10% of equivalent mutants [8].

In 1996, it was observed that detection of equivalent mutant is an instance of the feasible path problem and mathematical based constraints were introduced [5]. If the input constraint is unsatisfied, only then the mutant is equivalent, otherwise mutants are killable as they satisfied the constraint. On average, this technique is able to find 45% of identical mutants. A methodology based on program slicing helps in the manual examination of equivalence of mutants is proposed by Hierons *et al.* [9]. Program slicing extracts knowledge corresponding to a specific calculation that can automatically find likeness between mutant and its correct program. Harman *et al.* also recommended a related technique based upon program dependence to detect equivalent mutant [10].

To avoid the creation of identical mutants, co-evolutionary technique was proposed [11]. The aim of this technique is to produce featured test cases and mutants by simultaneously evolving test cases and mutants. Kintis and Malevris suggested the use of nine arrangements of data flow to find semantically alike mutants [12]. Their evaluation showed that 69% of functionally equal mutants created by AIOS operator were detected.

Schuler *et al.* proposed the use of dynamic invariants to measure the impact of a mutant [13]. The idea behind this approach was then used by Grun *et al.* to assess mutation based on coverage impact [14] and then Schuler and Zeller encompassed it [4]. Schuler evaluated that coverage impact is more realistic for categorizing dead mutants. Another approach towards this direction, suggested the use of method I-EQM to segregate identical mutants [15]. The classification scheme used in this technique employed second order mutants for categorizing first order mutants as dead or live.

PROPOSED WORK

A technique using the RIP model and observing the control flow graph of original and mutant program is used to reveal functionally similar equivalent mutants. In the following paragraphs, Control Flow Graph (CFG) and RIP model have been discussed in brief.

Control Flow Graph

Control flow testing is a method where test data are used to run certain sequence of events. Control flow graph describes the flow of control by creating a graph of source code. CFG is a graphical representation of all paths that might be traversed during execution of a code. In CFG, basic block is represented by a node i.e. series of continuous code with no jumps or branches, and an edge denotes transfer of control from a node to another. There is exactly one entry and one exit in CFG. The CFG can be supposed of as a program counter traversing through the code. Many compilers use CFG to model a program. Reachability is the main property of graph used in optimization. If a sub-graph is not connected from the sub-graph having entry block, then that sub-graph is unreachable during execution and so is an unreachable code [16].

RIP model

The Reachability, Infection and Propagation (RIP) model explains the effect of defects on program execution. The 'RIP Framework' describes three essential conditions for considering a mutant dead. It also states that at least one of the essential conditions must be unsatisfied to treat a mutant equal to its correct program.

ISSN: 2277-9655
Impact Factor: 4.116

1. ***Reachability*:** Reachability requires that control should reach the mutated statement by test case. If no possible test suite can execute the mutated statement, then it is unreachable.
2. ***Infection*:** Infection constraint requires that execution of mutated statement results in a state difference i.e. after running the mutated statement, the state of mutant and its "right" program must differs. Infection estimates the influence of mutants on state of a program.
3. ***Propagation*:** The infection caused by mutated statement i.e. infected state must transmit to certain observable point in the program. The observable points can be an output or return statement and also the expression that depends on the mutated statement must also compute a different value.

Proposed model is based on the assumption that equivalent mutants are syntactically different and semantically same i.e. functionally equivalent (for every possible test case they will produce the same output). Its meaning is that they are performing the same job using two different logics and if using a test case the difference in the logic can be realized then mutants can be determined as semantically similar i.e. equivalent mutants. So if using a test case the flow of control in P (i.e. Correct program) and P'(i.e. Equivalent mutant) is different then, following inferences can be drawn: (i) The reachability condition of RIP has been satisfied, (ii) The infection is not caused, (iii) The infection is caused but third condition of RIP i.e. Propagation to some observable point is not satisfied. As two programs which are semantically same may be logically different and if a test case has shown the difference between the two by the traversal of different paths in the CFG. It means the test case is adequate enough to realize the logical difference as the results are still same. So P and P' are logically different but functionally equivalent. (For example, programs for bubble sort and selection sort are functionally equivalent and logically different). So P' is an equivalent mutant.

Implementation

In a direction to test the proposed model, two cases are used. In first the equivalent mutant is created by making the change in an operator, and the second by making a change in an operand. Then the flow of control is observed.

Sample program 1

Original program(X)	Mutant(X')
1. Read a, b, c;	1. Read a, b, c;
2. m=a;	2. m=a;
3. If(b>=m)	3. If(b>m)
4. m=b;	4. m=b;
5. If(c>=m)	5. If(c>=m)
6. m=c;	6. m=c;
7. Write m;	7. Write m;
8. End	8. End

Figure 2.The mutant X' is created by replacing the operator >= in X in line no. 3 above by > in X'.

The program X in Fig.2 finds the largest from three numbers a, b, c and X' is mutant of X. The input given is a=8, b=8, c=4. Fig.3 and Fig.4 shows the control flow graph for X and X' respectively. In X', the mutated statement no. 3 is reached and executed. So, reachability constraint is satisfied. But the execution of mutated statement does not cause infection in terms of any change in variable's value or memory states. Hence, infection constraint is not satisfied. At observable point i.e. statement no. 7, there is no sign of infection, which results in same output i.e. 8 in both programs. Also no other statement after mutated statement modifies the value of m, which can alter the infection done by mutated statement.

ISSN: 2277-9655
Impact Factor: 4.116

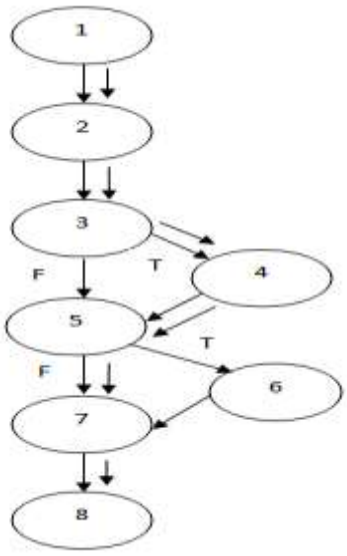

Figure 3.Control Flow Graph of X

When we compare the flow of control for both X and X' in Fig.5 and Fig.6 respectively, it shows the execution of mutated statement no. 3. It is found that logic is changed at statement no. 3, which cause different path to be traversed after the execution of statement no. 3 in X' and X.

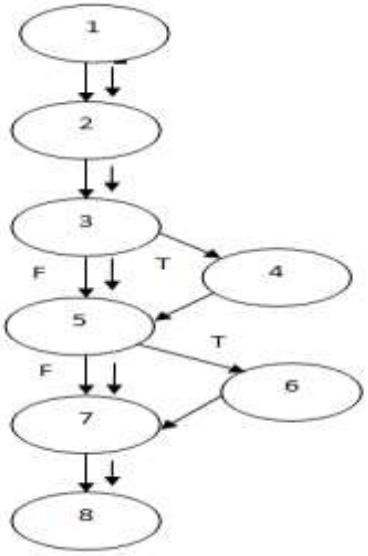

Figure 4.Control Flow Graph of X'

As in this sample1, infection only in terms of flow of control is observed, no state infection is present. Due to this, RIP model is not satisfied as whole, so the mutant is not considered as dead.

ISSN: 2277-9655
Impact Factor: 4.116

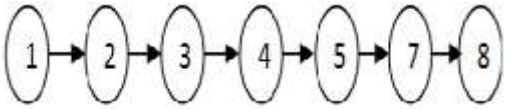

Figure 5.Flow of control of X

Same output for the input (i.e. same input provided to both X and X') but through different path shows that both programs are not functionally different and the difference in the logic due to the mutation has been realized which causes the change in flow of control of X'. Therefore, there is no further need to kill this mutant being equivalent mutant.

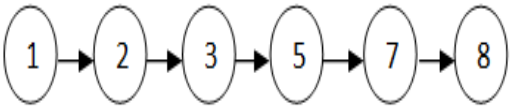

Figure 6.Flow of control of X'

Sample program 2

Original program(M)	Mutant(M')
1. Read A[], n;	1. Read A[],n;
2. For(k=0;k<n-1;k++)	2. For(k=0;k<n;k++)
3. For(i=0;i<n-k-1;i++)	3. For(i=0;i<n-k-1;i++)
4. If(A[i]>A[i+1])	4. If(A[i]>A[i+1])
5. temp=A[i]	5. temp=A[i]
A[i]=A[i+1]	A[i]=A[i+1]
A[i+1]=temp	A[i+1]=temp
6. Write A[];	6. Write A[];
7. End	7. End

Figure 7.Original program M and mutant M' created by changing the operand in M i.e. n-1 is replaced by n.

Fig.7 is program of sorting an array A of size n where M is original and M' is mutant program. Fig.8 shows control flow graph for both M and M'. The input (array) given to both M and M' is 7, 4, 5, 2. In M', the mutated statement no. 2 is reached and executed, satisfying the reachability constraint. The loop at statement no. 2, runs for k=0 to k<n in M'. This causes infection in the value of k. But the infection is not propagated to observable point. Hence, results in same output. RIP model is not satisfied as whole as only reachability and infection constraint is satisfied. This shows that the mutant is not killed.

ISSN: 2277-9655
Impact Factor: 4.116

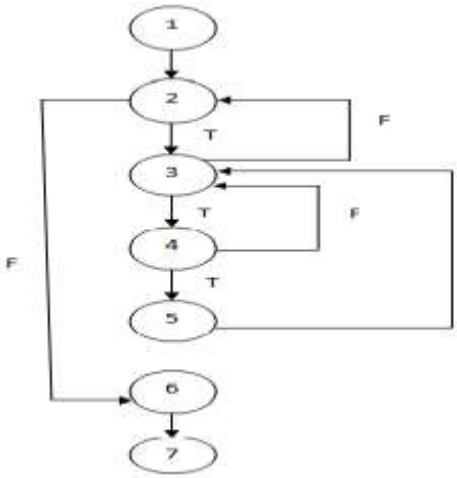

Figure 8.Control Flow Graph for M and M'

When we look at Fig.9 and Fig.10, there is some difference in flow states of M and M'. The infection at statement no. 2 is reflected from the flow of control. The loop at statement no. 2 is changed in M' which causes k to run one more time than the original program. So, this infection causes difference in flow of control for both original and mutant program but the output for both the programs remains same.

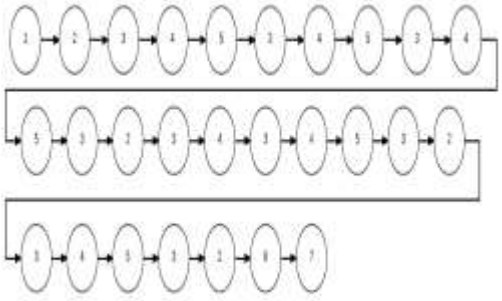

Figure 9.Flow of control of M

The same output for both program shows that both are semantically same. But different flow of control reflects the infection due to some syntactic difference in both programs. Hence, the mutant is treated as equivalent and we can stop the testing here.

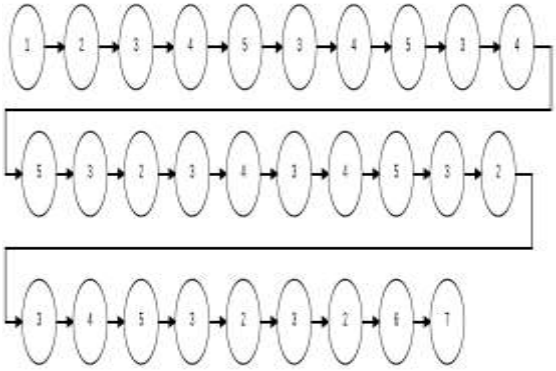

Figure 10.Flow of control of M'

CONCLUSION

Software testing evaluates software system in order to identify whether the software is doing what it is expected to do. Mutation testing uses the concept of fault seeding. The performance of test sets are measured in terms of Mutation Adequacy Score (MS=D/(M-E), the value of which lies between 0 and 1 depending upon the capability of test sets to reveal the mutants. In best case, it should be one. If there is no equivalent mutant, the formula will be MS=D/M. But due to the presence of equivalent mutant, the number of E is very difficult to decide. Due to an undecidability, it is not possible to conclude why the value of MS is not 1 i.e. whether it is due to live mutant or equivalent mutant. In this research paper, it is concluded that if a mutation causes a change in the logic and if that change is realized (by comparing control flow graph) by certain test input and still the mutant is not killed, then there is no further need to enhance the test cases and the change may be considered as equivalent mutant. In future, the research work can be extended to automate this process so that time and efforts can be saved.

REFERENCES

[1] Jefferson Offutt and Roland H. Untch, "Mutation 2000:Uniting the orthogonal," 2000.
[2] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, "Hints on Test Data Selection:Help for the Practicing Programmer," IEEE, pp. 34-41, 1978.
[3] R. G. Hamlet, "Testing programs with the aid of a compiler," IEEE Transactions on Software Engineering, pp. 279-290.
[4] David Schuler and Andreas Zeller, "(Un-)Covering Equivalent Mutants," in IEEE Third International Conference on Software Testing,Verification and Validation, pp. 45-54.
[5] A. Jefferson Offutt and Jie Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," Software Testing,Verification and Reliability, pp. 165-192, 1997.
[6] Timothy A. Budd and Dana Angluin, "Two notions of correctness and their relation to testing," Acta Informatica, pp. 31-45, 1982.
[7] Douglas Baldwin and Frederick Sayward, "Heuristics for determining equivalence of program mutations," 1979.
[8] A. Jefferson Offutt and W. Michael Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," 1996.
[9] Rob Hierons, Mark Harman, and Sebastian Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," Software Testing,Verification and Reliability, pp. 233-262, 1999.
[10] Mark Harman, Rob Hierons, and Sebastian Danicic, "The Relationship between Program Dependence and Mutation Analysis," in Mutation Testing for the new century.: Kluwer Academic Publishers.
[11] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation using Co-evolution,".
[12] Marinos Kintis and Nicos Malevris, "Using Data Flow Patterns for Equivalent Mutant Detection," in IEEE International Conference on Software Testing,Verification and Validation Workshops, 2014.
[13] David Schuler, Valentin Dallmeier, and Andreas Zeller, "Efficient Mutation Testing by Checking Invariant Violations," in 18th International Symposium on Software Testing and Analysis, 2009, pp. 69-80.
[14] Bernhard J.M. Grun, David Schuler, and Andreas Zeller, "The Impact of Equivalent Mutants," in IEEE International Conference on Software Testing,Verification and Validation Workshops.
[15] Marinos Kintis, Mike Papadakis, and Nicos Malevris, "Isolating First Order Equivalent Mutants via Second Order Mutation," in IEEE Fifth International Conference on Software Testing ,Verification and Validation, 2012.
[16] Wikipedia.[Online].https://en.wikipedia.org/wiki/Control_flow_graph