A Novel Method of Mutation Clustering Based on Domain Analysis

Changbin Ji[1,2], Zhenyu Chen[1,2*], Baowen Xu[1], Zhihong Zhao[1,2]
[1]*National Key Laboratory for Novel Software Technology, Nanjing University, P.R. China*
[2]*Software Institute, Nanjing University, P.R. China*
* zychen@software.nju.edu.cn

Abstract

Mutation testing is an effective but operational expensive technique. There exists much improvement to help mutation testing become a wide-used testing technique. The main objective is to reduce the number of mutants and test cases to be executed. The simplified mutant set can still approximate the adequacy. Many new techniques are put forward continuously. One of them is called mutation clustering. It integrates data clustering and mutation analysis, such that both mutant set and test set can be reduced dramatically.

This paper revisions the extant mutation clustering. The goal is to provide a method which can guide clustering mutants statically before test case generation. And then the simplified mutant set is used to generate a test set. We hypothesize that such a test set can kill the original mutant set approximately. The cost of mutation testing will further decreases, because only a small test set needs to be generated and executed. A case study shows the encouraging result to support our hypothesis.

1. Introduction

Mutation testing is testified as an effective fault detective method [1]. The mutation testing can find most faults in the program with high probability. It begins with changing the original program into many mutants. Testers will run all the mutants against test cases. The test case kills one mutant when the result is different from the initial result. More test cases may be added to kill the live mutants. The test set has a mutation score as a label to illustrate its ability to detect the faults.

Mutation analysis has achieved much advance in fault-based testing [2]. But it is still on the way of becoming a commercial testing technique, because of many industrial reasons [3]. One fatal problem is that mutation testing is computationally expensive. Even a program with 30 lines of codes may generate hundreds of mutants. To execute so many mutants will spent much time and resource. Testing mission with time constrains can only choose giving up mutation testing. Costing too much human resource is anther problem that needs to be tackled. There existed syntactically different but semantically same mutants. They are called equivalent mutants. No test case can kill this kind of mutant so that equivalent mutants must be removed. Distinguishing equivalent mutants is usually done by hand, so that test cases are sometimes needed to be selected manually.

The works to improve the performance of mutation analysis continue to today [3]. Many strategies are introduced to reduce the computational expense. Weak mutation technique which is called “do smarter” pays attention on when to compare the states during mutating. A “do faster” approach reduces the cost of running mutants in the same environment. For example, schema-based mutation analysis built some models for mutation which compiles and executes faster than conventional interpretive method [4].

Obviously, it is easy to recall a “do fewer” method. The methodology means to execute fewer mutants against fewer test cases [3]. The key idea is executing the important mutants to save resource. The number of mutants is determined by the mutation operators. Minimizing the mutation operators will work. A. J. Offutt et al. proved that mutation testing will keep its testing strength with fewer mutation operators [3]. Wong et al. introduce selective mutation criterion which chooses mutation operators according to their performance on detecting faults [5, 6]. The work in [7] went to further step to prove that ABS, AOR, LCR, ROR and UOI are five necessary mutation operators which provides the same coverage as the non-selective mutation. Randomly sampling can also avoid executing large number of mutants. It randomly selects x% mutants from initial mutant set. Ignoring the remaining mutants is testified effective [8]. Chen et al. used fault class hierarchy to skip some mutation operators in Boolean specification-based testing [9]. Moreover, some graph contraction algorithms were provided to merge some mutants from a same mutation operator, such that the test set is reduced further [10].

S. Hussain et al. have introduced mutation clustering which could reduce the number of mutants and then to reduce the number of test cases [11]. At the beginning, they run all mutants against all test cases. The execution information was saved as measure for each mutant. Distance between mutants could be calculated. Then they used some clustering algorithms to assemble similar mutants. Test cases killing most mutants were reserved. The preparation for clustering was still operational expense. In this paper, we present a different method to weight mutants based on domain analysis. The measuring information is obtained statically to reduce the cost of executing every mutant against test cases. So that the cost of mutation testing is reduced and the energy of mutation testing is remained.

This paper is organized as follows. Section 2 provides the relative work of mutation clustering. In Section 3, some strategies are proposed to reduce the domains, because the domains of relative variables can help to distinguish the mutants. Experimental results and analysis is shown in Section 4. Some further improvement in Section 5 will be done in the future.

2. Relative Work

Data clustering is usually used in pattern-analysis, decision making, knowledge discovery and machine learning situations [12]. It is called non-supervised method to collect similar data elements. Data clustering conducts the distance between data items which will be classified. Using one suitable clustering algorithm [13], the data items which satisfy the similarity requirement will be clustered into the same category. The clusters act as the data set but have smaller scale. Clustered data set reserves the statistical feature and saves cost for following analysis. S. Hussain et al. apply clustering technique on minimizing the quantity of mutants [11]. Mutants and associated test cases are generated. All mutants are executed against the test cases. The result is recorded to represent which test cases can kill the one mutant. The mutants are represented by the test cases that kill it and the test cases that do not kill it. The discrimination between mutants is measured by Hamming distance. Additionally a way to determine the centroid of candidate clusters is provided. Then they apply K-means and Agglomerative clustering algorithm by these values of distance [14]. Large numbers of mutants are turned into fewer clusters. Then these clusters reduce the test cases under greedy algorithm. It is testified that mutation clustering do lower the computational cost and keep the energy of test cases.

Before clusters replace the original mutants set, all mutants must be executed against all test cases. It spends significant resource. We suppose there is a more effective way to divide the mutants directly but not to design and operate all test cases. We adopt another standard which can be gotten statically to mark every mutant.

CBT and DDR are two methods that can generate test cases automatically in mutation testing. Both of them focus on the domains of variables. CBT takes the algebraic expressions as constraints [15]. Constraints are reduces dimension by dimension. When all the regions of variables are determined, the test data will be generated to approximate relative adequacy. DDR takes initial set for each input [16]. When moving through the control flow of the program, the sets of values are modified dynamically. DDR resolves several shortcomings that CBT suffers. That means there are some inner relationship between values of variables and test cases. We hope the scales of variables can guide parting mutant set.

The application of mutation operators firstly generates some mutants for the program. According to domain analysis, one value set corresponds to one mutant. After that we calculate the distance from one mutant to another. Clustering methodology is employed here to cluster the mutants. Then we select one mutant from one cluster randomly. These mutants compose the next generation. Section 3 will introduce the domain analysis by which we reduce the domain of variables. Some kinds of statements are selected to decrease the values of domains. Replacing executing all test cases, we use static analysis to measure mutants.

3. Mutation Clustering

Some researchers make significant contribution to mutation clustering. Before mutants are clustering, total test cases and mutants will be executed. Then the results are used to evaluate the mutants. We intend to apply a static method to measure all mutants. The static method does not need to execute mutants and provides improvement on mutation clustering.

All variables in one program are picked up. For every mutant, we use some symbolic execution rules to reduce the domain of all variables [17]. The variable set which has the reduced domain is taken as a mutant representation. This method saves execution cost before mutation clustering. Basically we apply the static control flow analysis which has practical cost [18]. This section provides a few of specifically tactics to handle the execution path conditions in the analysis procedure.

3.1. Optional module and path condition

If one mutant works, the execution must satisfy the basic condition that the program would not exit before the mutation point.

Figure 1 depicts a simple program. The program has three possible ends under three varying conditions. We take the clauses which contain return statement as one program module and those normal clauses as some optional modules. The four modules are treated as four control dependence nodes of the entry node. For example, one mutant is generated from one statement in module 4. Obviously, the execution must avoid running module 1 and module 2. Optional module 3 can be not decided. Then the all the variables domain will be against the conjunction of condition 1 and condition 2. Condition 3 can be dismissed. The necessary condition consists of the inner module conditions and exit-against conditions. The following conditions are not under consideration.

Identifying the modules in the program is prepared for the further analysis. The thinking of reused software information accelerates determining the location of every mutant [19].

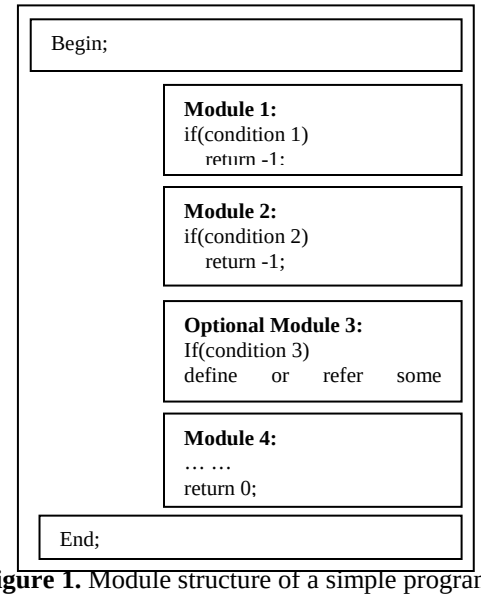

Figure 1. Module structure of a simple program

Practically, the state of the program includes input domains, path condition and a path counter which indicates the next module to execute. Take the example in Figure 1, we initial all input domains according to the software specification. When passing through the execution path, the path condition integrates the conditions from different modules. For instance, the path condition starts with True and Module 1 is the beginning value of path counter. The mutant is generated in Module 4. Firstly, the program encounters the Module 1 which presents a potential end. The path condition is ¬condition1. Path counter is Module 2. In like manner, after traversing Module 2, the path condition is (¬condition1∧¬condition2), and the path counter is Module 3. Module 3 is an optional module which can define or refer some variables. Path condition ignores this kind of module and path counter moves to the next module. Finally, the path condition turns into (¬condition1∧¬condition2). Path counter changes with Module END.

3.2. Reduction conditions for mutant

To distinguish every mutant, we use domain of relative variables to measure these mutants. The domain must satisfy the combination condition.

There are three types of conditions which should be combined.

1. Exit-against conditions: as it is described in subsection 3.2., choosing mutant path would avoid program exiting.

2. Nested conditions: in one module, there may be some nested conditions. All correlative conditions up the mutation point will be satisfied.

3. Mutated conditions: Mutation operators make syntactical differences on conditions. We take the mutated conditions into account. Mutants that affect the same statement have the equivalent exit-against conditions and nested conditions. The mutated statement is treated as mutated conditions so that we can tell differences.

Original:	**Mutant:**
if(A > 0 || B > 0) return 1;	if(A > 0 && B > 0) return 1;

Figure 2. Mutated condition

Path condition combines type 1 and type 2 conditions. In the third state, when some conditional operators are encountered, we take the mutated condition as true. In Figure 2, the original operator is || which means one true part causes the whole statement true. Mutation operator COR, i.e. conditional operator replacement, changes || into && which represent intersection result of several conditions. Then the following statement in the same module will be considered under execution. It supposes that the initial input of variable A is [-10, 10] and B is [-20, 20]. The default return value is 0. The domains of A changes to [0, 10] and B stays the same when we analyze the program statically because the whole module is an optional module. When the mutated conditional statement is under observation, the domain of A

changes to [0, 10] and B is reduced to [0, 20]. The reason is that the mutated condition is an affirmative one. And then the return value should be 1.

The intersection of path condition and reduction condition forms one constraint which the variable must satisfy. The values of domains are bounded after all the correspondent conditions are synthesized.

3.3. Defining mutant distance and centroid of cluster

The three types of conditions above are grasped out to determine the domains of all relative variables in program to be tested. The combination of the domains corresponds a mutant and forms a measure to distinguish mutants from each other. It needs to define a distance calculation method to apply clustering regulations.

Hamming distance provides a solution. Table 1 shows how Hamming distance works in our analysis. A, B, C and return value represent the variable measure for M1 and M2. If the domains differ, a 1 is brought in. Otherwise, the distance is 0. Then the sum of 1 is counted as the distance of the different mutants. In Table 1, the distance between M1 and M2 is 3.

Table 1. Calculate the distance between two mutants

	A	**B**	**C**	**Return Value**
M1	(-30,30)	(-30,15)	(-15,0)	1
M2	(-30,30)	(-30,30)	(-30,30)	-1
Dis.	0	1	1	1

Through the distance value, we apply a procedure like K-means algorithm to agglomerate mutants. Firstly we randomly choose k clusters. K is usually the half of total number. Then we evaluate the distance from the remaining mutants to these k clusters. One mutant joins the cluster whose distance is less than the specific value. Then halving k, the procedure will be repeated.

Before number of clusters can be further reduced, one delegate mutant must be chosen to represent the whole cluster. K-means algorithm uses the means of clusters. Due to the specificity of the values of domain, it is hard to directly calculate the means of two or several domains. We elect the delegate arbitrarily. The k mutants take part in next round clustering. The total mutant number will be reduced to k/2.

4. Experimental Result and Analysis

The aim of this section is to evaluate the hypothesis that the values of domains can also distinguish the mutants as execution results of test cases. Test set generated from clusters can show almost the same ability to kill the mutants as the original test set.

We divide the experiment into several steps below:

1. At first, we apply muJava[1] to generate mutants M_F for a program P. Some mutation operators in class are dismissed according to the feature of the case program [20].
2. Then a test set T_F is designed for M_F and evaluated with mutation score MS_F.
3. We find all variables in P including return values.
4. The domains of the variables are recorded according to the characteristic of every mutant. We take the domain as a measure of every mutant.
5. Now we calculate the distance between every two mutants.
6. A *k* which is half of total mutants and a threshold value that is half of max distance value are picked. Clustering selects *k* mutants randomly as the initial clusters. We cluster the mutants which have the distance less than the threshold value. One mutant is selected randomly to present one cluster and these mutants form the new mutant collection M_N.
7. Test cases which can not kill one mutant M_N are removed from T_F. Remaining test cases compose new test set T_N.
8. T_N is evaluated with MS_N to kill the M_F. The efficiency of T_F and T_N is compared by different mutation scores.
9. The quality of mutants is also measured to show our method can effectively reduce the execution cost for mutation testing. If necessary, M_N can be clustered more times and T_N will be also reduced.

In the actual case study, we take the triangle program as the object. Table 2 provides the results of mutation clustering. We manually write 21 test cases for M1 which is generated by 12 mutation operators. M1 are divided into 73 clusters. Then 73 mutants are selected from the clusters. These mutants are called M2. In the same manner, M3 derives from M2 and M4 derives from M3. We remove the test cases that can not kill one mutant in M2. The test cases are reduced from 21 to 16. We try to kill M1 using the 16 test cases. Significantly, the number of killed mutants is reduced only by 4. The total number of mutants is saved by 50%. The computation cost of mutation testing falls successfully. One more point worth mentioning is that after the mutants are clustered two times the test cases turn into low quality. In this case M3 gains the best performance. M3 has only 25% mutants of M1 and 13 test cases filtered by M3 kill 94% mutants that 21 test cases killed.

1 http://cs.gmu.edu/~offutt/mujava/

Table 2. Results of mutation clustering

1 Number of test cases:21			
	Number of mutants	**Number of killed mutants**	**Mutation Score**
M1	**147**	**137**	**93%**
M2	**73**	**68**	**93%**
M3	**36**	**35**	**97%**
M4	**18**	**18**	**100%**

2 Number of test cases:16 filtered by M2			
	Number of mutants	**Number of killed mutants**	**Mutation Score**
M1	**147**	**133**	**90%**
M2	**73**	**68**	**93%**

3 Number of test cases:13 filtered by M3			
	Number of mutants	**Number of killed mutants**	**Mutation Score**
M1	**147**	**130**	**88%**
M2	**73**	**66**	**90%**
M3	**36**	**35**	**97%**

4 Number of test case:9 filtered by M4			
	Number of mutants	**Number of killed mutants**	**Mutation Score**
M1	**147**	**117**	**79%**
M2	**73**	**60**	**82%**
M3	**36**	**31**	**86%**
M4	**18**	**18**	**100%**

Test cases are minimized with the mutant decreasing. Whereas, the test cases are still strong as the original test set. Actually the structure of triangle program affects the strength of test set. It is difficult to design a test set which can achieve 100% mutation score for the triangle program, because there are two inputs and one output in the program. We can not handle mutants generated from logical mutation operators in this case. So the descending of M2 and M3 is acceptable. The positive results testify our hypothesis that values of domains can keep the strength of mutants as the test cases. Clustering mutants provides an optimal solution, and it makes mutation testing to execute fewer mutants. Our method moves a further successful step.

5. Conclusion and Future Work

We took the triangle program as the object program. This program has 6 relative variable references. Comparing with the method of S. Hussain, our method depends on fewer dimensions; we take one variable as a dimension. It will improve the definition when we revise the strategy to measure the distance. In case of one domain involves the other one, we record the distance with 1 and the totally different ones with 2. In such way, we can extend the range of distance value. The mutants will be classified with higher accuracy.

The second problem which needs to be tackled is that how the centroid of a cluster should be determined. All the mutants are represented by the domain of variables. It lacks a method to define a rational means of several values of domains.

The prophase of the case study is done manually. We will envision a full automatic system to simulate the whole procedure of mutation clustering. Then we can apply our analysis on other programs with bigger scopes. We hope it can help reduce the operational cost and makes the mutation testing practically.

6. References

[1] A.P. Mathur and W.E. Wong, "An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria", *Software Testing, Verification & Reliability.* 1994, 4(1): 9-31.

[2] J. H. Andrews, L. C. Brand and Y. Labiche, "Is mutation an appropriate tool for testing experiments?", *in Proceedings of the 27th International Conference on Software Engineering.* 2005, pp. 402-411.

[3] A.J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal", *Kluwer International Series on Advances in Database Systems, Mutation testing for the new century.* 2001, pp.34-44.

[4] R. Untch, A. J. Offutt and M. J. Harrold, "Mutation analysis using program schemata", *in Proceedings of the 1993 International Symposium on Software Testing, and analysis*, 1993, pp.139-148.

[5] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Constrained mutation in C programs", *in Proceedings of VIII Symposium on Software Engineering.* 1994, pp.439-452.

[6] A. J. Offutt, G. Rothermel and C. Zapf. "An experimental evaluation of selective mutation", *in Proceedings of the Fifteenth International Conference on Software Engineering.* 1993, pp.100-107.

[7] A. J. Offutt, G. Rothermel and C. Zapf, "An experimental determination of sufficient mutation operators", *ACM Transactions on Software Engineering and Methodology.* 1996, 5 (2):99-118.

[8] A. P. Mathur and W. E. Wong, "Reducing the cost of mutation testing: an empirical study", *Journal of Systems and Software*, 1995, 31(3):185-196.

[9] Z.Y. Chen, B.W. Xu, X.F. Zhang and C.H. Nie, "A novel approach for test suite reduction based on requirement

relation contraction", *in Proceedings of the ACM symposium on Applied Computing.* 2008, pp.390-394.
[10] Z.Y. Chen, B.W. Xu and C.H. Nie, "A detectability analysis of fault classes for Boolean specifications", *in Proceedings of the ACM symposium on Applied Computing.* 2008, pp.826-830.
[11] S. Hussain, M. Harman, "Mutation Clustering", Master Thesis, King's College London, UK, 2008.
[12] A. K. Jain, M. N. Murty, P. J. Flynn, "Data Clustering: A Review", *ACM Computing Surveys.* 1999, 31(3): 264-323.
[13] G. Fung. "A Comprehensive Overview of Basic Clustering Algorithms", 2001.
[14] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman and A. Y. Wu, "An Efficient k-Means Clustering Algorithm: Analysis and Implementation", *IEEE Transactions on Pattern Analysis and Machine Intelligence.* 2002, 24(7):881-892.
[15] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation", *IEEE Transaction on Software Engineering*. 1991, 17(9):900 -910.
[16] A. J. Offutt, Jin Z and Pan J., "The dynamic domain reduction procedure for test data generation", *Software: Practice and Experience*. 1999, 29(2): 167 -193.
[17] Corina S. Pasareanu and Willem Visser, "Verification of Java Programs Using Symbolic Execution and Invariant Generation", 2004, LNCS 2989:0302-9743.
[18] A. Rountev, O. Volgin and M. Reddoch, "Static control-flow analysis for reverse engineering of UML sequence diagrams", i*n Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering table of contents*. 2005, pp.96–102.
[19] R. Prieto-Díazd, "Domain analysis: an introduction", *ACM SIGSOFT Software Engineering Notes archive*. 1990, 15(2):47-54.
[20] Y. S. Ma, J. Offutt and Y. R. Kwon, "MuJava: a mutation system for java", *in Proceedings of the 28th International Conference on Software Engineering.* 2006, pp. 827-830.