Journal of Southeast University (English Edition) Vol. 27, No. 1, pp. 17 – 21 Mar. 2011 ISSN 1003—7985

# Novel technique for cost reduction in mutation testing

Jiang Yuting Li Bixin

(School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

**Abstract:** Aimed at the problem of expensive costs in mutation testing which has hampered its wide use, a technique of introducing a test case selection into the process of mutation testing is proposed. For each mutant, a fixed number of test cases are selected to constrain the maximum allowable executions so as to reduce useless work. Test case selection largely depends on the degree of mutation. The mutation distance is an index describing the semantic difference between the original program and the mutated program. It represents the percentage of effective test cases in a test set, so it can be used to guide the selection of test cases. The bigger the mutation distance is, the easier it is that the mutant will be killed, so the corresponding number of effective test cases for this mutant is greater. Experimental results suggest that the technique can remarkably reduce execution costs without a significant loss of test effectiveness.

**Key words:** mutation testing; mutation distance; sample learning; execution cost; test case selection

**doi:** 10.3969/j.issn.1003 – 7985.2011.01.004

Mutation testing is a fault-based testing technique early proposed by DeMillo et al[1]. It has been proved to be very useful and effective to detect faults in programs in many researches [2-5]. However, high cost has always been the key restriction factor that obstructs its widespread use. Therefore, it is very important to study some techniques to reduce the cost of mutation testing. Mutation testing imposes unacceptable demands on computing and human resources because enormous numbers of mutants need to be compiled and executed against one or more test cases[6]. The number of executions in conventional mutation testing is computed as follows:

Suppose that the test case set is $t = \{c_1, c_2, \ldots, c_n\}$, and $M$ is the mutant set[7].

$$\begin{aligned}\text{exeCnt}(t, M) = {} & \#M + \#M - \#\text{kill}(M, \{c_1\}) + \\ & \#M - \#\text{kill}(M, \{c_1, c_2\}) + \ldots + \\ & \#M - \#\text{kill}(M, \{c_1, c_2, \ldots, c_{n-1}\})\end{aligned}$$

In the computational formula, the symbol # means to obtain the size. For example, $\#M$ represents the size of the mutant set. Function "kill" returns the set of mutants which have been killed by the test cases in the parameter table. Obviously, this repetitive execution upon different test cases will cause considerable costs especially when the mutant turns out to be an equivalent one. If every test case in the set cannot kill the mutant, all the running work that has been done for this mutant is useless.

In this paper, we seek an efficient way to reduce this kind of useless work by executing a mutant fewer times. We assume that if a mutant cannot be killed after a proper number of executions, it seems highly unlikely to be killed by the remaining test cases either. Generally speaking, the proper number depends largely on the implicit relationship between test cases and mutants.

In order to discover the implicit relationship between test cases and mutants, the mutation distance is introduced in this paper. It is defined to describe the semantic difference between the original program and the mutated program. By using the mutation distance, test cases and mutants are able to be linked together. It is obvious that a program with much semantic change will be easily killed because it will act very differently from the original one. The distance between them is long. There is always only one original program in the procedure of mutation testing, so we can utilize the mutation distance to classify mutants. Based on different distances, we cluster mutants into different classes and then we can provide an appropriate number of executions for each class to reduce useless work. Numerical values of the mutation distance in this paper are obtained from sample learning.

## 1 Mutation Distance

In mutation testing, many faulty versions (known as mutants) are generated through introducing faults to the original program[8]. The concrete operation of introducing faults can be viewed as replacing one "action" in a program with another. For example, expression $(a + b)$ being mutated to $(a - b)$ is accomplished by replacing a "+" action with a "–" action. This kind of operation can be reduced to arithmetic operator replacement (AOR) which is called a mutation operator. The implementation of mutation testing cannot operate without mutation operators. Researchers in this field have proposed many kinds of mutation operators. For example, in Tab. 1 we list five typical operators and present some details about them. These five operators are generally considered as the most effective operators in mutation testing[9]. It is worth noticing that even an operator will possibly generate many mutants, because there are not only different feasible replaceable actions but also different locations in a program for change. Take the operator ROR as an example, there are totally six relational operators: >, <, ==, !=, >= and <=, if a program has $N$ different relational operator usages, then the number of mutants produced by ROR is $5N$.

As discussed above, since mutants are generated by replacing one action with another, the difference degree between

**Received** 2010-09-06.

**Biographies:** Jiang Yuting (1986—), female, graduate; Li Bixin (corresponding author), male, doctor, professor, bx.li@seu.edu.cn.

**Foundation items:** The National High Technology Research and Development Program of China (863 Program) (No. 2008AA01Z113), the National Natural Science Foundation of China (No. 60773105, 60973149).

**Citation:** Jiang Yuting, Li Bixin. Novel technique for cost reduction in mutation testing[J]. Journal of Southeast University (English Edition), 2011, 27(1): 17 – 21. [doi: 10.3969/j.issn.1003 – 7985.2011.01.004]

**Tab. 1** Five typical mutation operators

| Mutation operator | Description | Feasible action |
|---|---|---|
| ABS | Absolute value insertion | $x->\|x\|$ |
| AOR | Arithmetic operator replacement | +, -, *, /, mod |
| LCR | Logical connector replacement | &&, \|\| |
| ROR | Relational operator replacement | <, <=, >, >=, ==, != |
| UOI | Unary operator insertion | ++, --, !, +, - |

actions can be used to describe the mutation distance between the original program and the mutated program. In this paper we use mdis$(i,j)$ to denote the mutation distance between programs. $i,j$ are different actions of a mutation operator; $i$ is the original action while $j$ is the mutated one.

Although $i,j$ are feasible actions of the same mutation operator, mdis$(i,j)$ may vary differently. Entirely different effects may be produced even by applying the same operator. For example, a relational expression in the original program is $a+b<c$; the mutant with expression $a+b<=c$ is much the same as the original one semantically and it is only under the situation $a+b=c$ that they produce different outputs. However, the mutant with the expression $a+b>c$ will act quite differently in most cases. So mdis$(<,<=)$ will be much smaller than mdis$(<,>)$. The value of mdis$(i,j)$ represents how semantically different the two actions are. The greater the value is, the easier the corresponding mutant will be killed.

In this paper, mdis$(i,j)$ is computed by learning some sample programs. Due to the reason that action is not specific to a single program, the results obtained will be instructive and meaningful for other programs.

## 2 Calculation of mdis$(i,j)$ by Sample Learning

Suppose that an action set for a typical mutation operator is $A=\{a_1,a_2,a_3,\ldots,a_n\}$, a sample set $Y=\{Y^1,Y^2,Y^3,\ldots,Y^m\}$ in which $Y^1$, $Y^2$ and so on are all sample programs. The set of generated mutants is $M=\{Y^1_{12},Y^1_{13},\ldots,Y^1_{1n},Y^2_{12},\ldots,Y^m_{1n}\}$, $|M|=mC^2_n$; the member $Y^k_{ij}$ means replacing action $i$ with action $j$ for program $Y^k$. Because there may be more than one location that can be changed in $Y^k$, $Y^k_{ij}$ itself is a set of mutants. Case $=\{c_1,c_2,c_3,\ldots\}$ is the set of test cases which are automatically generated by the test data generator according to input information about the program. By executing all the test cases for each mutant, the number of effective test cases will be obtained and then the mean number for $Y^k_{ij}$ can be figured out. The mean number is called as mean_Case$(Y^k_{ij})$ and it indicates how many test cases in Case are effective for $Y^k_{ij}$. mdis$(i,j)$ is then computed through comprehensive analysis of all the selected samples. The computational formula is as follows:

$$\text{mdis}(i,j)=\frac{\sum_{k=0}^{m}\text{mean\_Case}(Y^k_{ij})}{m\,|\,\text{Case}\,|}$$

According to the above formula, mdis$(i,j)$ is always between 0 and 1. The minimum value is 0, meaning that a mutant is much the same semantically as the original program and it cannot be killed by any case in the test case set. The maximum 1 means a mutant is quite different from the original one and can be killed by almost every test case. From the formula, we can say that mdis$(i,j)$ is actually a probability value representing what percentage of test cases are effective, so this value can be used to determine the maximum allowable executions later. As long as the sample set is representative and the samples are enough in number, the results will be accurate enough to describe the difference degree between actions.

## 3 Test Case Selection

In mutation testing, of course, the most interesting mutants are those that are small semantically changed leading to subtle mistakes. However, these mutants are very hard to kill and always cause a lot of useless work. If a mutant cannot be killed in restrictive times, it should be marked as a suspected mutant. Then the tester should be told to resort to other techniques, either to generate a specific test case or to determine whether it is an equivalent mutant instead of a continuous execution which may lead with a high probability to all having been in vain. In this section we will discuss how to make use of mdis$(i,j)$ to make mutation testing more efficient by restricting the maximum executions for each mutant.

Semantically alike mutants should be handled carefully and mdis$(i,j)$ can be used to distinguish among the mutants. So mutants are first clustered into different classes according to different mdis$(i,j)$. For each class of mutants, the difficulty level of being killed varies differently. So the number of test cases selected for them varies. Two test case selection methods are proposed in this paper. In order to evaluate the performance of our technique, two performance indices are proposed.

1) exeCnt: execution times, the workload of mutation testing, to check the efficiency.

2) test_quality: to verify how many mutants have been killed, and to show the effectiveness of our technique, namely

$$\text{test\_quality}=\frac{\#\text{killed\_mutant}}{\#\text{all\_mutant}}$$

### 3.1 Mutant clustering based on different mdis$(i,j)$

Mutants are first clustered into different classes for better organization. mdis$(i,j)$ reveals a kind of implicit relationship between test cases and mutants. As discussed above, semantically alike mutants should be handled carefully. So in our technique, mutants are first clustered into two classes:

$$\begin{aligned}&\text{class}(P)\ |\ \text{mdis}(i,j)\leqslant R\\&\text{class}(Q)\ |\ \text{mdis}(i,j)>R\end{aligned}$$

where $R$ is the threshold value of mdis$(i,j)$.

With the clustering, mutants are divided into two parts. We propose that for mutants with short distance, selected test cases should be limited and thus execution costs can be saved. For mutants with long distance, it is likely that the

first few test cases will kill them, so execution times do not increase no matter how many test cases are selected. The mutants in class($Q$) can be further divided into different classes according to mdis($i, j$). Our technique deals mainly with the mutants of short distance.

The critical part in our testing technique that is different from conventional mutation testing is the using of the test case selection method. We propose two methods for test case selection in this paper called CRS and HRS, respectively.

### 3.2 Complete-random-select (CRS)

In this method, we randomly select a proper proportion of test cases from the original set of cases for each mutant according to the mutation distance. The number of test cases selected for class($Q$) is determined by mdis($i, j$). When mdis($i, j$) is very short, if we still use the same way for class($Q$) to determine the number; only very few test cases or even none cases will be selected. Obviously, this is not appropriate because it may miss some effective test cases. So for class($P$), we use a fixed value in this method. In this way, we can not only reduce execution times, but also improve the chance of finding some effective test cases. All the test cases are selected randomly and we assume that if none of the selected test cases can kill the mutant, which also means that the mutant has reached its maximum allowable executions, the rest may act in much the same way. The number of test cases selected for each class of mutants is computed as follows:

$$\text{mean\_Case}(P) = \lceil \text{Case} \rceil R$$
$$\text{mean\_Case}(Q) = \lceil \text{Case} \rceil \text{mdis}(i, j)$$

### 3.3 Heuristic-random-select (HRS)

In CRS, we do not consider the sustained influence of effective test cases. If one case can kill a mutant, it is more likely to kill other mutants than other test cases. So by this method, we take this kind of probability into consideration and then the selection is not completely random.

We first choose a mutant with the largest distance in class ($P$) and then execute it against all the test cases, record effective test cases and put them into a test case set, and finally extend the set to satisfy its required size. The size of candidate test cases set is the same as that of CRS. This method reserves certain particular test cases for use, to keep the test _quality as high as possible. More details are described in the algorithm.

**Algorithm 1** Heuristic random select

if $m \in$ class($P$)

execute Case on mutant with

$\text{mdis}_{\max}(i, j)$

put every effective test case into a set

extend this set to $\lceil \text{Case} \rceil R$

else $m \in$ class($Q$)

cluster class($Q$) again and select the corresponding number of test cases.

## 4 Empirical Study

This section conducts an empirical study to evaluate the performance based on two indices. All the relevant contents involved in the experiment are organized in Fig. 1.

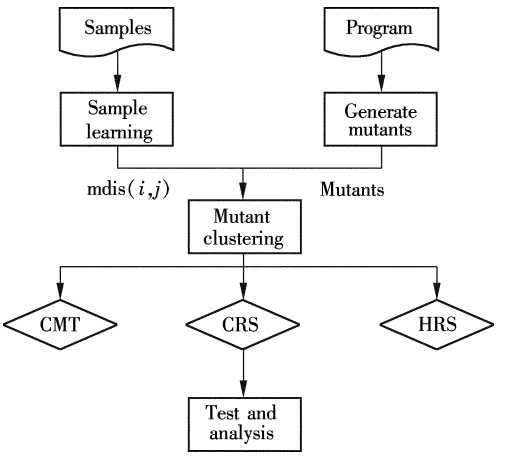


**Fig. 1** Experimental procedure

### 4.1 Calculation of mdis($i, j$)

The premise of our experiment is to calculate the distance between actions. Here, we choose ROR as our research objective. The action set is $A = \{ <, >, <=, >=, ==, != \}$. The sample programs[10] are listed in Tab. 2.

**Tab. 2** Sample programs

| Function prototype | Description |
|---|---|
| bubble(int $A[\,]$, int $n$) | Bubble sort on an integer array |
| mid(int $i$, int $j$, int $k$) | Median of three integers |
| min(int $a$, int $b$) | Return the minimal value of two integers |

After calculation, fifteen different kinds of mutation distance learned from samples are presented in Tab. 3.

**Tab. 3** Values of mdis($i, j$)

| $i$ | $j$ | mdis($i, j$) | $i$ | $j$ | mdis($i, j$) | $i$ | $j$ | mdis($i, j$) |
|---|---|---|---|---|---|---|---|---|
| < | <= | 0.001 3 | <= | > | 0.777 8 | > | == | 0.553 2 |
| < | > | 0.777 8 | <= | >= | 0.777 8 | > | != | 0.552 5 |
| < | >= | 0.777 8 | <= | == | 0.552 5 | >= | == | 0.553 2 |
| < | == | 0.552 5 | <= | != | 0.553 2 | >= | != | 0.552 5 |
| < | != | 0.553 2 | > | >= | 0.001 3 | == | != | 0.777 8 |

### 4.2 Experimental results

We choose tritype[11] as the program under the test. The function of tritype is to determine the type of a triangle based on the length of three triangle sides. By applying the operator ROR, 85 mutants are generated. Then we cluster the mutants by mdis($i, j$) and compare three methods of test case selection to find the most optimal one. The first of them is conventional mutation testing (CMT) which does not involve a test case selection strategy, and the remainders are CRS and HRS, respectively.

Results in Fig. 2 and Fig. 3 show that both exeCnt and test_quality increase as well with the growing amount of test cases. Fig. 2 shows that the HRS is always superior to the CRS and the CMT in reducing workload, and that with the

growing amount of test cases, this superiority is more outstanding. Moreover, test_quality of the three methods are very similar as shown in Fig. 3, and the CMT is best. The CRS and the HRS are very close to each other. This is caused by the reduction of execution times. In the CRS and the HRS, as both of test case sets are reduced a lot, this will definitely lead to a loss in test_quality. The number of killed mutants is decreased and more mutants are marked as suspected equivalent mutants for human intervention. But the gap is very small, especially when the test case set is huge, and the actual test quality of these three methods are almost the same. So this kind of loss is affordable. It can be concluded that the CRS and the HRS can reduce execution cost remarkably and do not deteriorate the effectiveness of mutation testing.

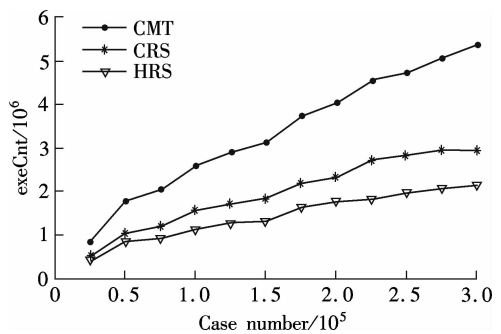


**Fig. 2** exeCnt of three methods

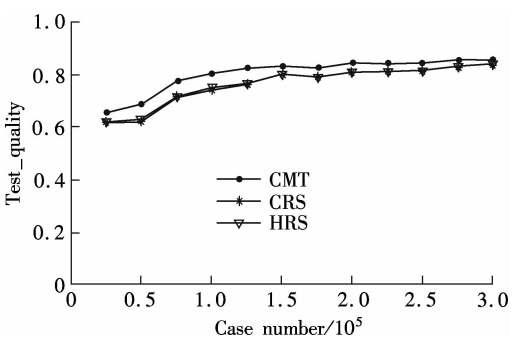


**Fig. 3** Test_quality of three methods

When the amount of test cases is 25 000, exeCnt in the conventional method is 872 547, test_quality is 65.88%, and the number of mutants that are not killed is 29. So useless work in the CMT is at least 72 500 execution times. The effective work is 147 547 executions, with a very low work efficiency of the whole workload which is about 16.9%. The total workload in the HRS is 427 179 executions and the test_quality is 62.4%. With the number of surviving mutants increasing to 32, test_quality decreases about 3%, but workload decreases 51%, so testing efficiency has been greatly improved.

In actual mutation testing, the amount of mutants is always very huge, far more than 85. In this situation, mutants can simulate more mistakes and the workload of testing will be heavy. If using the CMT, the efficiency will be very low; while using the HRS, the efficiency of the system will be enhanced greatly, and the test quality is maintained at the same time. The research findings in this paper will be especially helpful when the program under the test is of a large size because each execution of the program will cost a lot of time and resources.

Furthermore, the results in the above experiment show that the HRS is the best of the three methods in both exeCnt and test_quality. So we study the HRS further.

Fig. 4 and Fig. 5 show that exeCnt and test_quality are rising synchronously with the increase of $R$. exeCnt increases greatly when $R$ is in the range from 0.5 to 0.6. Also it can be seen that test_quality is locally optimal when $R$ is about 0.5. So in this experiment, it is better to set $R$ as 0.5. That is because at this point, test_quality reaches a higher level and exeCnt is not very large.

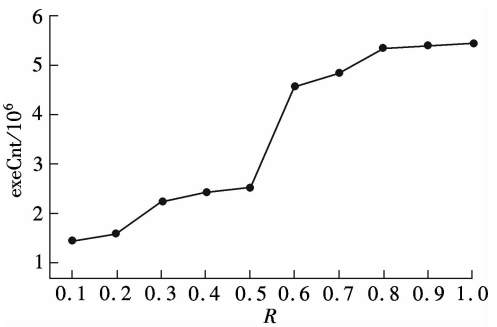


**Fig. 4** Relationship between $R$ and exeCnt

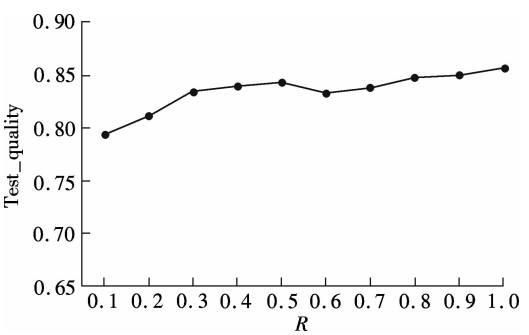


**Fig. 5** Relationship between $R$ and test_quality

## 5 Conclusion

This paper proposes to apply test case selection in the procedure of mutation testing to reduce testing cost. We define the mutation distance to represent the semantic difference between the original program and the mutated program, and then use it to guide in the selection of test cases. Experiments show that our technique is useful in cost reduction. However, there still exist some problems to be solved. The mutation distance is largely determined by the sample set, so the procedure of sample learning is very important. Future work may be conducted on making sample learning smarter and being able to deal with feedback information.

## References

[1] DeMillo R A, Lipton R J, Sayward F G. Hints on test data selection: help for the practicing programmer [J]. *IEEE Computer Society*, 1978, **11**(4): 34 - 41.

[2] Frankl P G, Weiss S N, Hu C. All-uses versus mutation: an experimental comparison of effectiveness[J]. *Journal of Systems and Software*, 1997, **38**(3): 235 - 253.

[3] Wong W E, Mathur A P. Fault detection effectiveness of

mutation and data-flow testing[J]. *Software Quality Journal*, 1995, **4**(1): 69 - 83.
[4] Andrews J H, Brand L C, Labiche Y. Is mutation an appropriate tool for testing experiments? [C]//*Proceedings of the* 27*th International Conference on Software Engineering*. Los Alamitos, CA, USA, 2005: 402 - 411.
[5] Yue J, Harman M. An analysis and survey of the development of mutation testing[R]. London: Center for Research on Evolution, Search and Testing(CREST), 2009.
[6] Wong W E, Mathur A P. Reducing the cost of mutation testing: an empirical study[J]. *The Journal of Systems and Software*, 1995, **31**(3): 185 - 196.
[7] Bottaci L, Mresa E S. Efficiency of mutation operators and selective mutation strategies: an empirical study[J]. *Software Testing, Verification and Reliability*, 1999, **9**(4): 205 - 232.
[8] Zhang L, Hou S S, Hu J J, et al. Is operator-based mutant selection superior to random mutant selection? [C]//*Proceedings of the* 32*nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2010: 435 - 444.
[9] Offutt A J, Lee A, Rothermel G, et al. An experimental determination of sufficient mutant operators[J]. *ACM Transactions on Software Engineering and Methodology*, 1996, **5**(2): 99 - 118.
[10] Offutt A J, Rothermel G, Zapf C. An experimental evaluation of selective mutation[C]//*Proceedings of the* 15*th International Conference on Software Engineering*. Baltimore, MD, USA, 1993: 100 - 107.
[11] DeMillo R A, Offutt A. Constraint-based automatic test data generation[J]. *IEEE Transactions on Software Engineering*, 1991, **17**(9): 900 - 910.

# 一种用于降低变异测试代价的新技术

蒋玉婷　　李必信

(东南大学计算机科学与工程学院,南京 210096)

**摘要**:针对变异测试代价大、无法广泛应用的问题,提出了一种在变异测试过程中引进测试用例选择以降低测试代价的方法.通过为每个变异体选择一定数量的测试用例,约束变异体允许执行的最大次数,从而减少无用功.测试用例的选择与变异体的变异程度相关,变异距离是描述源程序和变异体之间差异程度的指标,能够衡量测试集中有效测试用例的比例,进而指导测试用例的选择.距离越大,意味着变异体越容易被杀死,对应的有效测试用例则越多.实验结果表明该方法在不影响测试效果的情况下,可以明显降低变异测试的执行成本.

**关键词**:变异测试;变异距离;样本学习;执行代价;测试用例选择
**中图分类号**:TP311