Journal of Theoretical and Applied Information Technology
28th February 2013. Vol. 48 No.3
© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

STUDY ON SOFTWARE TESTING SUFFICIENCY BASED ON PROGRAM MUTATION TECHNOLOGY

[1]NING JINGFENG , [2]FAN XIAOLI

[1]College of Computer Science and Engineering, Changchun University of Technology, Changchun 130012, China

[2]China United Netwofk Communications Corporation Limited of Baicheng branch

Email:[1] ningjingfeng@mail.ccut.edu.cn， [2]18643601115@wo.com.cn

ABSTRACT

—Software testing sufficiency means that the software's performance on limited testing data can represent its performance on all input data. Ideally, in software testing, the testing should be carried on till all errors in the program are detected and removed. As a testing strategy to measure the completeness of the test cases set, mutation testing is a defects-oriented unit testing technology, and a feasible software testing method to generate a complete set of test cases. The article systematically simulates the different defects in software by using mutation operators to create mutants, and then construct testing data set to be able to kill these mutants. It analyzes the procedures of mutation testing and the generation of mutation operators with specific examples. Experiment indicates that the program mutation technology has enhanced the test cases, which greatly improved the software testing sufficiency.

Keywords： *Program Mutation, Software Testing, Mutation Operators, Testing Sufficiency*

1. INTRODUCTION

On one hand, the aim of software testing is to detect the errors and defects in the software as many as possible, and give definite opinions on whether the final software products meet the specified requirements or not; on the other hand, is to find maximum underlying problems on the tested software products and their executing process with minimum cost and time [1]. The sufficiency criteria of software testing have qualitative description theoretically, for example: there are limited sufficient test sets for any software; the more the testing is, the less the sufficiency growth for further testing is, etc. However, it is still a research topic that needs continuous practice on quantitative evaluation [2]. To measure the sufficiency of the software testing process [3], the first is to solve the measurement index problem [4]. This article analyzes the evaluation testing sufficiency and enhances test sets through program mutation technology, which is an effective technology to evaluate the testing performance. This technology provides a set of strict criteria for testing evaluation and testing enhancement. Even the test set meets certain testing sufficiency criteria, such as MC/DC covering criteria, most criteria are not sufficient for program mutation.

2. SOFTWARE TESTING SUFFIENCY

The aim of software testing is not to verify its correctness, but to detect errors. To evaluate the degree of the testing process can be measured by testing sufficiency. The followings are the relevant definitions of software testing sufficiency.

Definition1. Software testing sufficiency: Suppose software P is to meet functional requirements set R, recorded as （P， R）. R includes n requirements， recorded as R_1， R_2， …， R_n； suppose test set T includes K testing cases to verify whether P meets all requirements in R, and suppose each testing case in T has executed P, and P runs correctly.

C: If for each requirement r in R, at least one case in test set T proves that P meets r, it is considered that T is sufficient against (P, R).

Definition 2. Measurement of testing sufficiency: Given test set T and covering domain C_e, which has n elements, n≥0. What we call T covers C_e means each element e in C_e is tested by at least one testing case in T. If T covers all elements in C^e, it is considered as T is sufficient against criterion C; if T only covers K elements in

C^e, $k<n$, it is considered as T is insufficient against C. Fraction k/n represents T's sufficiency against C, also called T's coverage rate against C, P and R.

3. MUTATION TESTING

Program mutation testing method can be traced back to the late 1970s, which was originally proposed by DeMillo, Lipton and Sayward. It is based on Competent Programmer Hypothesis (CPH) and Coupling Effect Hypothesis, which is defects-oriented software testing method [5-6]. The key to mutation testing is how to generate the needed test data. At present, there are mainly two automatically generated methods to kill the test data of the mutants: Constraint-Based Test data generation (hereinafter referred to as CBT method) [7], and Dynamic Domain Reduction test data generation (hereinafter referred to as DDR method) [8].

3.1 Basic Definitions Of Mutation Testing

Definition 3. Mutation is a behavior of modifying program even in small ways. P refers to the tested original program, M refers to the program after slight modification of P, then M is called as P's mutant, P is called as M's parent. Suppose the grammar of P is correct, and is able to pass compiling, then M is sure to be grammatically correct. The behavior represented by M is the same as that of P.

Definition 4. The killed mutant has a test case t (test cases set T), when one mutant represents a different behavior characteristic from P, the mutant is killed.

$$\exists t \in T | f(m, t) \neq f(P, t)$$

Definition 5. Live mutant for any test case t $\in T$, mutant m and p always represent the same behavior, and then the mutant is live.

$$f(m, t) = f(P, t) \qquad \forall t \in T$$

Definition 6. Mutation score is used to measure the capability of test case set to detect errors, use the following formula to calculate:

$$MS(P, T) = \frac{K}{M-E}$$

Where: K is the number of the killed mutants; M is the total number of mutants; E is the number of mutants equivalent to the original program.

There are two possibilities for mutants to be "live": one is the mutants are equivalent to the original program; second is the modified code is not executed, i.e. The present set of test cases is not sufficient to detect the errors. Hence, through this mutation score (MS), quantitative analysis on the sufficiency of the set of test cases can be made to help us to create new test cases for more sufficient testing on the program.

For program p, M is a mutant generated from the mutation operation of its statement S, if the specified test case t can kill M, then the test case t is effective, if not, t is ineffective. If the following 3 broad conditions, t is sure to kill M:

C_1 Reachability:

Mutant M is generated from mutation of executable statement S in program P, other statements in the mutant are the same as the original program, if test case t fails to execute the mutation statement S in M, the operating result of t on P and M must be consistent, so it cannot kill M.

C_2 Necessity:

If test case t intends to kill M, t must create a different state at the same point from the original program after executing mutation statement S. Mutant M after statement S has the same code with the original program P, if after executing the mutation statement, it has the consistent state with the original program after executing the statement at the same point, the operating state of the two programs are inevitably consistent, thus failing to kill M. Necessary condition doesn't contain reachable condition, which is described by the predicate expression of mutation statement.

C_3 Sufficiency:

The inconsistency of final operating state of test case t for Mutant M and original program P is the sufficient condition for t to kill M.

3.2 Mutation Operators

The key to mutation testing is how to generate mutants, while the mutation operator is the basis of generating mutants. Mutation operator is a production device or a program transformational rule. It transforms one grammatical structure into another grammatical structure, and ensures the correct grammar of the program after conversion, but not to keep the conformity of the semanteme. Mutation operators can be done aiming at different grammatical items, for example, relational operators, predicate, arithmetic expression and other sections to design mutation operators. The mutation operators mentioned in the article use the mature mutation operators as reference from the 22 kinds of mutation operators in the traditional program mutation, the details are as follows:

TABLE I. Mutation Operators

Name of mutation operators	English description
VRP	Value Replacement
COR	Comparator Operator Replacement
LCR	Logical Connector Replacement
AOR	Arithmetic Operator Replacement

3.3 Basic Principle Of Mutation Testing

The basic principle of mutation testing is to define a set of mutation operators, simulate the errors in the program through the large amount of mutants generated from mutation operators' applying to the source program, apply mutation operators to the source program to create a set of mutants, mutation operator is a small grammatical change on the source program. Operate the source program and mutants on the designed test cases, if the results are different, the mutants are said to be killed. Generally the effectiveness of the test cases is evaluated by the ratio of the killed non equivalent mutants. Mutation on one section of a program at a time is called single mutation, while mutation on k sections of a program is called K mutations. Multiple mutations on the same-location are called same-location multiple mutations.

4. EVALUATION PROCEDURES OF TESTING SUFFICIENCY BASED ON MUTATION TECHNOLOGY

P is the program to be tested, T is P's test set, and R is the requirement that P must meet. Given program P and test set T, quantitative evaluation on the excellent degree of T by calculating P's mutation value can be obtained. Mutation value is a numerical value between 0 and 1. If the mutation value is 1, it indicates test set T is sufficient against mutation criteria, while the mutation value is less than 1, it indicates T is insufficient. An insufficient test set can be enhanced through adding test cases to make its mutation value increase.

Evaluation procedures of test set sufficiency using mutation technology are as shown in Fig. 4-1. Among them, solid lines direct to the next dealing procedure, and dotted lines represent the flow between database and dealing procedure. L represents live mutants, D represents distinguished mutants, and E represents equivalent mutants. P (t) represents the behavior when program executes test case t, and M (t) represents the behavior when mutants execute test case t.

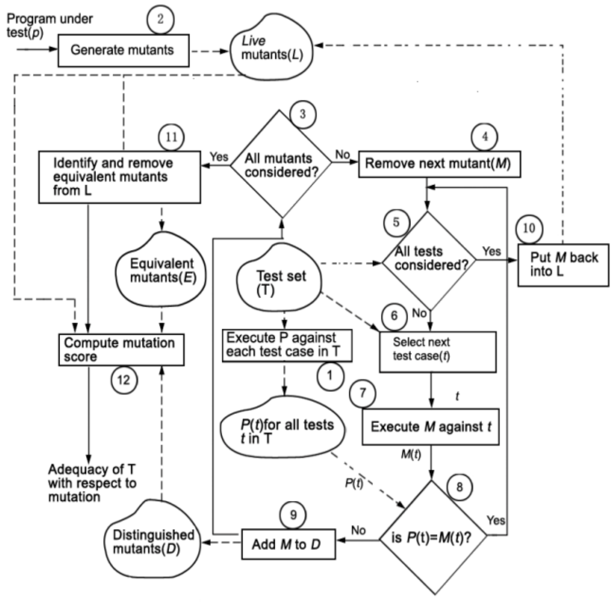

Fig. 4-1 Evaluation Procedures Of Test Set Sufficiency Based On Mutation Technology

1. Execution program. The first step to evaluate test set T against the sufficiency of (P, R) is to execute P against each test case in T. P (t) represents the behavior being observed when P executes t. Generally, the behavior of program P is indicated as the set of output variable in P. Of course, the behavior observed might also be related to the performance of program P. If program P has executed each test case in test set T, and P (t) has recorded in database, step 1 is not necessary. Anyhow, the final results of step 1 are a P (t) database for all t ∈T. Now, suppose f
(t) meets their requirement R. If P (t) is found to be incorrect, program P must be modified, then re-execute step 1. It must be pointed out, after finding program P against test set T is completely correct, the evaluation procedures of testing sufficiency using mutation technology are really begin.

2. Generate mutants. The second step to evaluate the test set T against the sufficiency of (P, R) is to generate mutants.

3. Select next mutant. In step 3 and step 4, select the next mutant to be considered, which must be the mutant not been considered before. Note from now on, mutants in L will be cyclically selected, till each mutant is selected. Obviously, when only one mutant has not been selected in L, only this one can be selected, which has finished in step 3. If several live mutants have not been selected in L, just selecting any one of them, and removing the selected mutant from L.

4. Select next test case. After selecting mutants M, now we need try our best to find a test case from test set T to distinguish M from their parent program. Therefore, we need to execute mutants M against test case in T, thus entering into another cycle, i.e. execute mutants M against each selected test case. When the cycle finishes, either all test cases are executed, or mutants M are found different from parent program by a certain test case, no matter what cases they are, the cycle finishes.

5. Execution and classification of mutants. Up to now, mutants M have been selected to execute test case t1. In step 7, use test case t to execute mutants M; in step 8, check if the results are the same from executing M and executing P against test case t.

6. Live mutants. When no test cases in test set T can distinguish mutants M from their parent program P, M are put back into live mutants set L. Any mutant that has been put back into live mutants set L will not be selected again, for it has been selected for one time.

7. Equivalent mutants. After executing all mutants, check shall be made whether there are live mutants or not, i.e. check whether L set is non-empty or not. If there are still live mutants, their equivalency with their parent program shall be checked. If for each test input against program P input domain, mutants M's behavior are in conformity with P, it is called mutants M are equivalent to their parent program P.

8. Calculation of mutation value. This is the last step to evaluate the test set T against (P, R) sufficiency. Given set L, D and E, using MS (T) represents the mutation value of test set T, the calculation is as follows:

$$\text{MS (T)} = \frac{|D|}{|L|+|D|}$$

It shall be noted that set L only includes live mutants, and these mutants are non equivalent to their parent program. Just as the above formula shown, the mutation value is always between 0 and 1.

Suppose ︱M︱represents the total number of mutants generated in step 2, the following formula can also be used to calculate the mutation value:

$$\text{MS (T)} = \frac{|D|}{|M|-|E|}$$

If test set T is able to distinguish all mutants besides equivalent mutants, ︱L︱=0 and mutation value MS (T) is 1. If T is unable to distinguish any mutant, ︱D︱=0 and mutation value MS (T) is 0.

5. EXPERIMENTAL ANALYSIS

Consider the following program P.

```
1 Begin
2  Int x， y,
3  Input (x，  y),
4  If (x < y)
5  Then
6  Output(x+y),
7  Else
8  Output(x*y),
9  End
```

P is used to compute function (x， y).

$$f(x,y) = \begin{cases} x+y & \text{if } x<y \\ x*y & else \end{cases}$$

Suppose the following test set is used to test P:

$$T_p = \begin{Bmatrix} t_1 :< x=0, y=0 > \\ t_2 :< x=0, y=1 > \\ t_3 :< x=1, y=0 > \\ t_4 : x=-1, y=-2 > \end{Bmatrix}$$

For all t∈T_p, their P(t) database list is shown as Table Ⅱ:

Table Ⅱ P (T) Database List

Test case(t)	Expected output f(x， y)	Observed output P(t)
t1	0	0
t2	1	1
t3	0	0
t4	2	2

Suppose the following mutants are changed from program P through the following procedures: (a) change arithmetic operators, replace all addition operators （+） with subtraction operators (-), replace all multiplication operators （*） with division operators （/）；（b）Change integer variable, replace integer variable v with v+1. Using the method, totally 8 mutants of program P are obtained, which are marked as M1 to M8 as shown in the following table.

Totally 8 mutants are obtained in the above table, which are called live mutants. Then we get a set

L ={M1, M2, M3, M4, M5, M6, M7, M8}

Select M1 mutant，remove M1 from L, we get

L = {M2, M3, M4, M5, M6, M7, M8}

Input test case in Tp, till it is distinguished from parent program. Select t1 :< x=0，y=0>.

Table Ⅲ Mutants Of Program P

Source program line No.	Original statement	Mutants identifier	Mutants statement
1 2 3 4 5 6 7 8 9	Begin Int x，y Input (x，y) If (x < y) then Output(x+y) Else Output(x*y) End	 M1 M2 M3 M4 M5 M6 M7 M8	None None None If (x+1 < y) If (x < y+1) None Output(x-y) Output(x+1+y) Output(x+y+1) None Output(x/y) Output((x+1)*y) Output(x*(y+1)) None

Table Ⅳ Statistical Table Of Execution Results For Test Cases

		t1 t2 t3 t4	D
Parent program	P(t)	0 1 0 2	{ }

Mutation program	M1(t)
	M2(t)
	M3(t)
	M4(t)
	M5(t)
	M6(t)
	M7(t)
	M8(t)
		0 0*
		NE NE
		0 1 0
		2
		0 2*
		NE NE
		0 2*
		NE NE
		0 -1*
		NE NE
		0 1 0
		0*
		0 1
		1* NE
		U* NE
		NE NE
			{M1}
			{M1}
			{M1，M3 }
			{M1，M3，M4 }
			{M1，M3，M4，M5 }
			{M1，M3，M4，M5，M6 }
			{M1，M3，M4，M5，M6，M7 }
			{M1，M3，M4，M5，M6 ,M7,M8}

Execute M1 against t1, for given input x=0，y=0， the output result is 0 for condition x+1<y is false, hence P (t1) =M1 (t1) =0, which means test case t1 is unable to distinguish M from P. When t= t1, condition P (t) =M (t) is true. Continue to select the next test case t2, execute M1 against t2, and find P(t2)=1， M1(t2)=0， P(t2)≠M1(t2)， add mutant M1 to the distinguished or killed mutants set D. then select mutant M2， execute M2 against test case in test set Tp, till M2 is distinguished or all test cases are executed in Tp. The summary of execution results is shown in table 4. The column D in the table represents the distinguished mutants set. Except for M2 in the table, test set Tp distinguished all mutants. While initially all mutants are live. NE represents the mutant in the line hasn't executed the corresponding test case in the column. M8 is distinguished by test case t1; its output is no definition for it is divided by 0, which is marked by "U". This means that P (t1) ≠M8 (t1), meanwhile the first test case that distinguished mutants is marked with asterisk (*).

In this application, only M_2 hasn't been distinguished by the test case in the test set, and becomes live mutant. In this case, only one live mutant, seven distinguished mutants, and no equivalent mutants, then |D|=7，|L|=1，|E|=0，calculate MS (TP) =7/ (7+1) =0.875.

Judge whether M_2 and P are equivalent or not. Analyze M_2 and P, suppose f_p (x，y) represents the function calculated by P, g_{m2} (x，y) represents the function computed by M2, as shown in the followings:

$$\text{Fp (x，y)} = \begin{cases} x+y & if \quad x<y \\ x*y & else \end{cases}$$

$$g_{m2}\text{ (x，y)} = \begin{cases} x+y & \text{if} \quad x<y+1 \\ x*y & else \end{cases}$$

To find if M_2 and P are equivalent then change to find the condition x=x1 and y=y1 for f_p (x1, y1) ≠g_{m2} (x1, y1). To make f_p (x1, y1) ≠gm2 (x1, y1), the following two conditions (identified as C_1 and C_2) must establish.

C_1: (x1＜y1)≠(x1＜y1+1)

C_2: x1*y1 ≠x1+y1

Design a test case t :< x=1, y=1> meet C_1 and C_2 simultaneously, it can be prove P (t) =1, M_2 (t) =2 through calculation, which shows that M_2 can be distinguished by at least one test case.

Therefore, M_2 is not equivalent to its parent program P. Add t into T_p, the improved T_p is obtained, T_p includes 5 test cases, MS(TP′)=1. So the test set T_p is enhanced through adding test case t.

6. CONCLUSION

The key problem of testing sufficiency is the capacity to check faults. To introduce mutation testing technology that can be used in unit and integration testing stages into software testing enables the effectiveness of evaluation of the existing testing sufficiency. The article analyzes the procedures of software testing sufficiency based on program mutation technology through test cases, which greatly improves the accuracy and reliability of judgment and decision-making for the software testing sufficiency. At present, the article only makes experiments on several traditional mutation operators, so the experimental data obtained have certain limitation, and the article only makes mutation testing against source code, to some extent, it is not comprehensive, in the future work, we will make further research on these problems, in hope of improving the efficiency of mutation testing.

ACKNOWLEDGMENT

The authors wish to thank Ningjingfeng and Huming.

REFERENCES

[1] BEIZER B. Software testing and quality assurance [M]. New York: International Thomson Computer Press, 1996.J. Clerk Maxwell, a Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] BURNSTEIN I, SUWANASSART T, CARSON R. Developing a testing maturity model for software test process evaluation and improvement [C] //International test conference, 1996.K. Elissa, “Title of paper if known,” unpublished.

[3] BLACK R. Software Testing Process Management [M]. Beijing：China Machine Press，2003 - 10.

[4] FENTON N, PFL EEGER S L. Software metrics-a rigorous and practical approach [M]. 2nd ed.1997.

[5] Zheng Ren-Jie. Computer Software Testing Technologies. Beijing: Tsinghua University Press，1992(in Chinese).

[6] Zhu Hong， Jin Ling-Zi. Quality Assurance and Testing of Software. Beijing: Science Press，1997(in Chinese).

[7] DeMillo R A， Offutt A J. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering， 1991，17(9): 900-910.

[8] Offutt A J， Jin Z， Pan J. The dynamic domain reduction procedure for test data generation. Software: Practice and Experience，1999，29(2): 167-193.

[9] Shan Jinhui， Gao Youfeng， Liu Minghao， et al. A new approach to automated test data generation in mutation testing [J]. Chinese Journal of Computers， 2008， 31(6):1025-1034 (in Chinese).

[10] Offutt a J， Pan J. automatically detecting equivalent mutants and infeasible paths [J]. Software Testing，Verification and Reliability，1997，7(3): 165-192.