An Effective Test Case Selection for Software Testing Improvement

Adtha Lawanna
Department of Information Technology
Vincent Marry School of Science and Technology, Assumption University
Bangkok, Thailand
adtha@scitech.au.edu

Abstract— **One problem of testing software is selecting the suitable test cases from the test suit regarding the size of the programs. If the size of selected test cases is big, then it can affect the whole performance of software development life cycle. Accordingly, it increases testing time and produce many bugs. Therefore, this paper proposes the improvement of software testing for selecting the appropriate and small number of test cases by considering the amounts of the functions modified, lines of code changed, and numbers of bugs produced after modifying programs. The reason of proposing the software testing improvement model is to prepare effective algorithm, while numbers of bugs are lower than the traditional methods. According to the experimental results, the size of the selected test cases by using the proposed model is less than Retest All, Random, and a Safe Test about 98.70%, 87.86%, and 84.67% respectively. Moreover, the ability of STI is higher than the comparative studies about 1-20 times regarding the number of bugs found after modifying a program.**

Keywords—software testing; test case; code; bugs

I. Introduction

Software engineering is applied for several fields such as computer science, system dynamics, system science, and management system [1]. The software development life cycle is a methodology used in this field [2]. However, the major problem of software testing, which is studied in this paper refers to choosing the appropriate test cases, which contain bugs, functions, and any errors [3]. The serious problem is the size of the selected test cases is too big when modifying program each version [4-5]. This causes the testing time and errors increase. Therefore, this paper presents the model to solve this problem. The retest all method, random technique and a safe test are used for the comparisons. The studies show that method of retesting all possible cases is simple but it introduces time consuming during testing the software. While, the random technique is easier than the previous method, when testing some test cases selected from the whole program. Unfortunately, that it cannot guarantee the accuracy of auditing the software [6]. Another is a safe test technique, which gives the better performance in term of reducing many ineffective test cases, while few bugs are produced compared with the old approaches [7-8]. To the survey, some traditional test case selection techniques work effectively regarding the complexity of codes, environments, and user requirements [9]. This paper studies three factors that can affect the test case selection, which are functions, codes, and software versions. The traditional methods mentioned earlier can be applied for these environments. However, the size by chosen test cases and numbers of bugs after using these techniques need the improvement for better performance, especially in the process of software testing. Therefore, the proposed model named, Software Testing Improvement (STI) is developed for improving the ability of choosing the test cases, while the minimum test cases and bugs are reachable.

II. The Concept of Selecting the Test Cases

A. Dataset

The seven subject programs developed by the Siemen Suite are used in this paper as shown in Table I.

The details of each program can be downloaded from http://pleuma.cc.gatech.edu/aristotle/Tools/subjects.

TABLE I. THE SUBJECT PROGRAMS

Name	*F*	*L*	*V*
replace	21	516	32
print_token	18	402	7
print_token2	19	483	10
schedule2	16	297	10
schedule	18	299	9
totinfo	7	346	23
tcas	9	138	41

Definitions;
F is the numbers of function.
L is the lines of code.
V is the numbers of version.

B. Traditional Methods

Retest-All Method: RA

This technique tests all test cases in a test suite before writing the new code by considering functions that required by both users and developers. It suits for the smallest size with low complexities under certain changes. However, it is not appropriate technique, where numbers of function and code are large. This means testing needs long time and high cost of the maintenance phase. Besides, integrating parts of testing is a difficult task.

Algorithm of RA explained as follows;

```
If test cases ($t$) are found then
    Select all test cases
End
```

Accordingly, RA cannot give the small numbers of test cases and produces many bugs after the testing. Therefore, some researchers find the better algorithm for handling this situation.

Random Technique: RD

To improve the performance of the previous methods, the random technique is developed. It reduces testing time, where all cases are tested. This approach can be applied to handle the software that has a big size regarding the requirements and lines of code [10]. The algorithm of RD is to select test cases randomly from the test suite. According to this, it may choose the irrelevant test cases instead of the relevant test cases. This makes the improvement of the new software drops.

Algorithm of RD described as follows;

```
If test cases ($t$) are found then
    Select some test cases randomly
End
```

Using RD algorithm can give two problems, which are uncertain selection and ineffective software testing, especially controlling program errors.

A Safe Test Technique:ST

This technique is proposed by Rothermel and Harrold, the results by selecting the test cases are more accurate than RD including the size of a test suite is smaller than the RA [11]. This technique determines the test cases that can produce programming errors or bugs after modifying the program. Therefore, it gives a good new version of software [12-13]. However, it may not focus some test cases that should be chosen before the modification. This is because after integrating all components, new faults are probably produced. Currently, the new technique is developed by the improvement of a safe test technique [14]. Accordingly, it is used in the part of evaluation together with random technique and the proposed model.

C. *Description of the Proposed Model*

Software testing improvement (STI) is proposed, particularly in the process of developing and modifying the new software version from the previous program. It is developed by considering the changes of the requirements and modified programs. First of all, the previous program will be realized in terms of the functions that can be changed regarding to the user requirements. Second is to find the lines of code that will be modified from the old program. Third is to determine the appropriate numbers of the test cases that will be used for the next version. However, the set of test cases will be tested. If it fails, then it needs to be revised by checking F and L respectively, until it passes.

D. *Concepts of the Model*

The user requirement of each software can influence the value of function. The average x-value will be determined by using all of the old software versions. Accordingly, the functions will be established. After this, the average b-value will be computed regarding to the used of programmers, Lines of code and testing time. The next step is to find the average *c*-value for finding the appropriate test cases. However, the selected test cases are dependent to human judgment (H).

E. *Algorithm of the Proposed Model*

The algorithm of STI is summarized in Fig. 1, which gives four main steps.

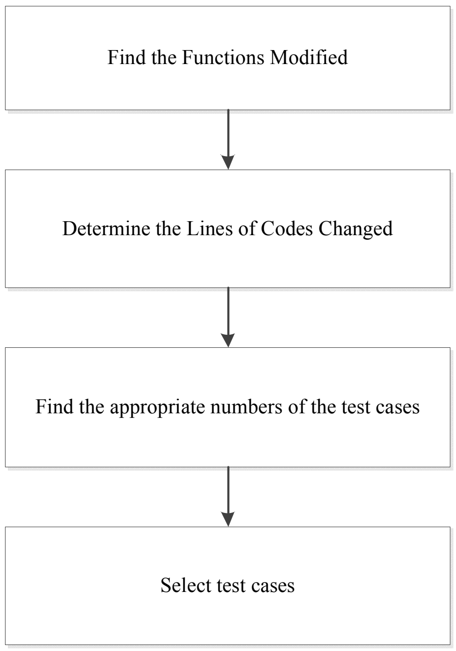
Find the Functions Modified
Determine the Lines of Codes Changed
Find the appropriate numbers of the test cases
Select test cases

Fig. 1. Algorithm of STI

According to Fig. 1, the details are given as follows;

Step 1: Find the Functions Modified

Each program, there are many times of modifying the software. For example, the numbers of modification of the program named replace, print-token, print-token2, schedule2, schedule, totinfo, tcase, space, and player are 32, 7, 10, 10, 9, 23, 41, 33, and 5 respectively. Relevant to these numbers of versions (modification), the original numbers of functions in each program result in the changes as well. So, this step is created in order to find the average on the functions changed for each version of the software.

Algorithm of x-value;

```
If $F \propto R$ then
    $F = xR$ or
    $x = \frac{F}{R}$        (1)
End
```

Whereas;

F is number of function.

R is user requirement.
x is constant value

Finding x is important in order for getting the constant value, which is occurring regarding the relationship of F and R. As we know that, if the software developers cannot control the changes of user requirements. This may the value of the modified functions performing inconsistency and unstable.

Step 2: Determine the Lines of Codes Changed

The code comprised with many lines of the instruction. The code becomes complexity, when the environment changes affect the whole software. They are relevant to testing times and programmers. Accordingly, the relationship will be defined to know the mentioned factors affect the capability of the entire program. Algorithm of finding b-value;

```
If $L \propto \frac{1}{T}$ then
    $L = s\frac{1}{T}$        (2)
    EsleIf $L \propto \frac{1}{P}$ then
        $L = p\frac{1}{P}$        (3)
    ElseIf $L \propto \frac{1}{PT}$ then
        $L = b\frac{1}{PT}$        (4)
        $b = LPT$        (5)
    EndIf
    EndIf
End
```

Whereas;
T is testing time
P is number of programmer.
b, p, s are constant value.

This algorithm can give the possibility of the lines of codes that will be changed because of adapting the previous code.

Step 3: Find the appropriate numbers of the test cases

The numbers of test cases are chosen for testing as the new software. This is the most essential process for improving the ability of testing the program. One of the objectives for the proposed model is to select the minimum of the test cases in order to control the size of the software. This means that if the size is too big, then the whole software will be complicated and hard to test each lies of codes, including time consuming may cause another problem. Algorithm of finding c-value;

```
If $t \propto F$ then
    $t = qF$        (6)
    EsleIf $t \propto L$ then
        $t = mL$        (7)
    ElseIf $t \propto FL$ then
        $t = cFL$        (8)
        $c = \frac{t}{FL}$        (9)
    EndIf
    EndIf
End
```

Whereas;
t is number of test case.
F is number of function.
L is line of codes.
q, m and c are constant value.

Step 4: The test case selection

Accordingly, the values of H (human judgments) need to be identified regarding to the related people. Each test case will be evaluated for the acceptance's level, which are described by the algorithm below;

```
Algorithm of test case selection
If $H(tn)$ = max then
        Select $tn$  // $tn$ is test cast that has maximum of $H$.
        ElseIf $H(tm)$ = less than max then
        Select $tm$ // $tm$ is all ways less than $tn$.
        ElseIf numbers of the selected test
        cases = $t$ then
        Stop selection
        EndIf
        EndIf
End
```

Assume that if we want two appropriate test cases then the value H-value will be provided. However, we can get the H-value from user's satisfaction, which is not explained in this research regarding to the complicated methods. As shown in Table II, t8 and t1 are selected regarding to using the selection algorithm.

TABLE II. TEST CASES SELECTION

t	$H(t)$	Acceptance
1	84	2nd selection
2	34	
3	5	
4	54	
5	41	
6	12	
7	32	
8	91(max)	1st selection
9	15	
10	33	
11	0	
12	5	
13	22	
14	41	
15	27	

In worst case, if the H-values are very low, all test cases need to be revised again. In fact, the assumption of setting the acceptance needs to be in consideration before the modification. According to this, the acceptance of H-values will be higher than 80%. This means that if its value is less than the acceptance level will be rejected. Suppose that, if t is equivalent to 5, then it is not appropriate to continue selecting the rest three test cases. This is because the *H*-values of the rest are lower than 80%. Therefore, the development team must define the new appropriate test cases by using the conceptual model proposed again.

Therefore, the whole algorithms by using STI may need the feedback from selecting the appropriate test cases within the development team and users.

However, the benefits of STI are satisfied, when handling the complexities and changes of the modified functions and fixing bugs are successful, which make the process of testing program gets more effective.

III. Results and Dicussion

A. Finding F and L

The results of finding *R*, *x*, and *F* for each program are running relevant to numbers of versions as reported in Table III-IX. Due to the proposed model, we can estimate *F* by using the average *x*-value the next generation of the subject program. According to this, the values of *F* of the subject programs; replace, print-token, print-token2, schedule2, schedule, totinfo, and tcas will be 9, 2, 4, 7, 6, 4, and 4 respectively. Besides, the results of finding *L* of the programs are 244, 210, 266, 102, 109, 152, and 60 respectively.

B. Finding Test Case (t) by using STI

The results in Table X refer to the number of the selected test cases for the seven subject programs. As described in this table, the value of *t* in replace, print-token, print-token2, schedule2, schedule, totinfo, and tcas are 27, 105, 67, 15, 18, 38, and 15 respectively. Therefore, in the next generation of updating software for each program, we can use this set of selected test cases to be modified and to run the modified software.

C. Comparative Studies on Size of Program

The results in Table XI, the size of each program by using four methods are shown. The results provide by the retest all methods are higher than other technique. This method is the easiest one that can be used to test the software. Time consuming is the main problem, while a safe test and STI can avoid it. However, this method will be a most powerful when numbers of the cases are very small. Accordingly, the values of test cases prepared by the STI are the lowest. This is alternative technique for testing software overnight because the sizes of the test cases have no effect to the whole processes.

D. Comparative Studies on Bugs of Program

The outputs of this section are demonstrated in Table XII-XIV and Fig. 2 shows the compared of numbers of bugs that could be produced after modifying the new software regarding using *RA*, *RD*, *ST* and *STI*. The algorithm of finding the bugs is described as;

If $B \propto t$ then $B = dt$ or

$$d = \frac{B}{t} \tag{10}$$

End

Whereas;

B is number of bug.

t is number of test case.

d is constant value.

The *d*-values are the average score of each faulty version regarding to the seven subject program. According to this, bugs found in the modified software version are considered. Accordingly, the results of finding the numbers of the bugs for each program by using RA are higher than the comparative studies. This is because they are varied by the numbers of the test cases, which are huge. On the other hand, using STI can reduce numbers of the bugs. Therefore, one of the benefits of considering STI is to avoid the bugs that can be occurred, including avoiding time consuming of testing the test cases.

E. Discussion

There are some interested points that should be discussed such determining the relationship between *F* and *R*. One of the possible results may get negative value, which means finding the relationship uses ineffective assumption, e.g., many *R* may not affect *F* much as it should be. Besides, finding the line of code changed may result the opposite from what we expect.

Moreover, another relationship of *L* and *F* can be found in term of positivism, which refers to when *F* increases can make *L* gets longer. However, this situation can be happed depending upon the knowledge and skills of the developers.

Even improving the ability of testing software depends on several factors, but the proposed model can work well, if *R* and *F* are in control. However, the most important factor is *H*, if it is ill defined, this can make the failure of selecting the relevant test cases.

TABLE III. Finding *F* and *L*of the replace, version 33

V	*R*	*x*	*F*	*b*	*P*	*T*	*L*
33	57	0.2	9	185450	16	49	244

TABLE IV. Finding *F* and *L* of the print-token, version 8

V	*R*	*x*	*F*	*b*	*P*	*T*	*L*
8	73	0.06	2	171483	16	52	210

TABLE V. Finding *F* and *L* of the print-token 2, version 11

V	*R*	*x*	*F*	*b*	*P*	*T*	*L*
11	91	0.11	4	220600	16	52	266

TABLE VI. Finding *F* and *L* of the schedule 2, version 11

V	*R*	*x*	*F*	*b*	*P*	*T*	*L*
11	66	0.15	7	87611	17	50	102

TABLE VII. Finding *F* and *L* of the schedule, version 10

V	*R*	*x*	*F*	*b*	*P*	*T*	*L*
10	63	0.13	6	97038	16	55	109

TABLE VIII. FINDING *F* AND *L* OF THE TOTINFO, VERSION 8

V	*R*	*x*	*F*	*b*	*P*	*T*	*L*
8	57	0.08	4	107591	15	43	152

TABLE IX. FINDING *F* AND *L* OF THE TCAS, VERSION 10

V	*R*	*x*	*F*	*b*	*P*	*T*	*L*
10	71	0.07	4	40707	15	41	60

TABLE X. FINDING TEST CASES

Name	*F*	*L*	*c*	*t*
replace	9	244	0.01	27
print_token	2	210	0.25	105
print_token2	4	266	0.06	67
schedule2	7	102	0.02	15
schedule	6	109	0.03	18
totinfo	4	152	0.06	38
tcas	4	60	0.06	15

TABLE XI. COMPARING SIZES

Name	*RA*	*RD*	*ST*	*STI*
replace	5,542	554	398	27
print_token	4,130	413	318	103
print_token2	4,115	412	389	67
schedule2	2,710	271	234	15
schedule	2,650	265	225	18
totinfo	1,052	214	199	38
tcas	1,608	203	83	15

TABLE XII. THE NUMBERS OF BUGS BY USING *RA*

Name	*B*	*d*	*t*
replace	15	0.0027	5,542
print_token	18	0.0044	4,130
print_token2	20	0.0049	4,115
schedule2	17	0.0063	2,710
schedule	17	0.0064	2,650
totinfo	16	0.0152	1,052
tcas	18	0.0112	1,608

TABLE XIII. THE NUMBERS OF BUGS BY USING *RD*

Name	*B*	*d*	*t*
replace	11	0.0199	554
print_token	13	0.0315	413
print_token2	13	0.0316	412
schedule2	13	0.0480	271
schedule	13	0.0491	265
totinfo	10	0.0467	214
tcas	13	0.0640	203

TABLE XIV. THE NUMBERS OF BUGS BY USING *STI*

Name	*B*	*d*	*t*
replace	0	0.0000	27
print_token	1	0.0097	103
print_token2	0	0.0000	67
schedule2	0	0.0000	15
schedule	0	0.0000	18
totinfo	0	0.0000	38
tcas	0	0.0000	15

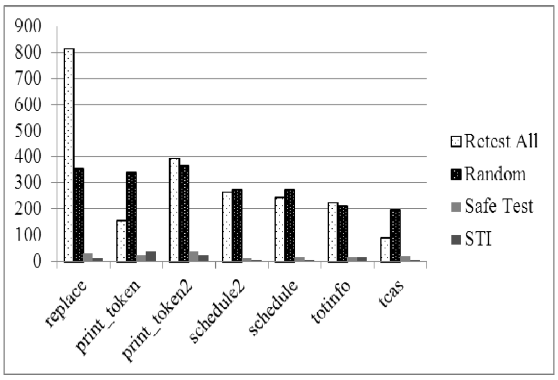

Fig. 2. Comparing number of bugs

IV. CONCLUSION

The software testing improvement (STI) is proposed for increasing the ability of test case selection regarding to the concept of regression test selection, which are RA, RD and ST techniques. Besides this, it can be used for predicting the numbers of functions modified and lines of code changed by using four algorithms demonstrated in the paper. These algorithms give three benefits listed as follows; the results of predicting the functions modified and lines of code changed for the new software version, the smallest numbers of the test cases, including bugs that are produced after modifying software are small when compared to the traditional methods. However, the STI is applied for seven subject programs and compared the performance with only three techniques. Therefore, it may not cover other situations or some important factors such as the limitation of testing software, knowledge and skill of testers, and the environment by using software.

REFERENCES

[1] A. Abran, J.W. Moore, P. Bourque, R. Dupuis, and L.L. Tripp, "Guild to the Software Engineering Body Knowledge," IEEE, 2004.

[2] J. Feller and B. Fitzgerald, "A framework analysis of the open source software development paradigm", Proceedings of the twenty first international conference on Information systems', International Conference on Information Systems, Brisbane, Queens-land, Australia, 2002, pp. 58-69.

[3] A. Zoitl, T. Strasser and A. Valentini, "Open Source Initiatives asbasis for the Establishment of new Technologies in Industrial Automation: 4DIAC a Case Study", published in IEEE International Symposium of Industrial Electronics (ISIE), 2010, pp. 3817-3819

[4] F.I. Vokolos, and P.G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," Proceeding of the International Conference on Software Maintenance, Shrewsbury, NJ, USA., November 16-20, 1998, pp. 44-53.

[5] H. Agrawal, J. Horgan, E. Krauser, and S. London, "Incremental regression testing," Proceeding of the Conference on Software Maintenance, Bellcore, Morristown, NJ, USA., September 27-30, 1993, pp. 348-357.

[6] D.H. Kitson, "A Tailoring of the CMM for the Trusted Software Domain," Proceedings of the Seventh Annual Software Technology Conference. Salt Lake City, Utah, 1995, April 9–14.

[7] M.Dorigo, V.Maniezzo and A.Colorni, "Ant System: Optimization by a colony of cooperating agent.", IEEE Transactions on Systems, Man and Cybernetics, vol. B (26), 1996, pp. 29-41.

[8] E. Brinksma, J. Tretmans, and L. Verhaard, "A Framework for Test Selection. Protocol Specification, Testing and Verification.", XI. Elsevier Science Publishers B.V., 1991, pp. 233-248.

[9] S. Yoo and M. Harman, “Pareto efficient multiobjective test case selection.”, In Proceedings of the 2007 International Symposium on Software Testing and Analysis, 2007, pp. 140–150.

[10] E. Wong, and A.P. Mathur, “ Fault detection effectiveness of mutation and data-flow testing,” SQJ., vol. 4, no. 1, 1995, pp.69–83.

[11] G. Rothermel, M.J. Harrold, “A safe, efficient regression test selection technique.” ACM Transactions on Software Engineering and Methodology, vol. 6, no. 2, April 1997. pp. 173–210

[12] G. Rothermel, and M.J. Harrold, “ Analyzing regression test selection techniques,” IEEE Trans. on Software Engineering and Methodology, vol. 22, no. 8, 1996, pp. 529-551.

[13] G. Rothermel, and M.J. Harrold, “Empirical studies of a safe regression test selection technique,” IEEE Trans. on Software Engineering and Methodology, vol. 24, no. 6, 1998, pp. 401-419.

[14] L. Yifan, and G. Jun, “Method of Automatic Regression Test Scope Selection Using Features Digraph,” Lecture Notes in Electrical Engineering., vol. 236, 2013, pp. 561-570.