World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:8, No:1, 2014

A Model for Test Case Selection in the Software-Development Life Cycle

Adtha Lawanna

***Abstract*—**Software maintenance is one of the essential processes of Software-Development Life Cycle. The main philosophies of retaining software concern the improvement of errors, the revision of codes, the inhibition of future errors, and the development in piece and capacity. While the adjustment has been employing, the software structure has to be retested to an upsurge a level of assurance that it will be prepared due to the requirements. According to this state, the test cases must be considered for challenging the revised modules and the whole software. A concept of resolving this problem is ongoing by regression test selection such as the retest-all selections, random/ad-hoc selection and the safe regression test selection. Particularly, the traditional techniques concern a mapping between the test cases in a test suite and the lines of code it executes. However, there are not only the lines of code as one of the requirements that can affect the size of test suite but including the number of functions and faulty versions. Therefore, a model for test case selection is developed to cover those three requirements by the integral technique which can produce the smaller size of the test cases when compared with the traditional regression selection techniques.

***Keywords*—**Software maintenance**,** regression test selection, test case.

I. Introduction

AMOUNTS of software are being developed for various fields such as business, including education and industry [1]. The maintaining software is one of the most important following issues in software-development cycle [2], [3]. One of the major harms of software maintenance is to execute a suitable test suite that is used to test before maintaining the modified code [4]. Test suite comprises a set of test cases used for fixing bugs, functions, and faults [5]. If test suite size is huge, and then executing time increases, this can reduce the abilities of the entire software. Therefore, this paper proposes a model for selecting a minimum test suite to fix to this problem. Another problem after selecting the cases, we should avoid the unintended bugs that can be performed while running the program. The reason is that the reduction of test cases may remove some test cases that should not be deleted from a test suite because they affect the entire programs (e.g., execution time increases) [6]. According to this, the regression test techniques are proposed produce the appropriate test suite before selecting them for the process of modifying the new software version. In general, there are three main strategies in regression test explained as follows; Regression Test Minimization involves removing irrelevant test cases. Regression Test Selection can choose the appropriate test cases based on multiple regressions. Regression Test Prioritization can rank test cases into small groups and selection the most relevant test cases [7]. Moreover, this paper studies the retest-all technique, random/ad-hoc selection, and the control graph flow which is a safe regression test selection [8]. In addition, one of the main objectives of those techniques is to produce the small test suite while faultless is still preserved. The record shows that the retest-all technique is simplest, but it introduces the maintenance cost because all test cases are revised. In the meantime, the random/ad-hoc selection techniques can reduce the running time, but it cannot preserve faultless rate [9]. The safe test based regression test selection can reduce numbers of test cases and offers the better faultless rate than others [10]. Therefore, a model for test suite selection (MT) is proposed to handle those problems mentioned above. It gives the better results compared with the traditional regression techniques. The challenge of MT is that it standardizes the requirements (e.g., the number of functions, the lines of code, and the faulty versions) and integrates them to find the small amounts of the average test cases. According to this, it claims that our selection technique can reduce many more test cases than some of the traditional regression selection techniques.

Basically, the software testers use the automated test case generation to produce the test suites, in which contain numbers of the test cases. Sometimes, a test suite is called a test pool, whereas a reduced suite of test cases is required during the process of maintaining software [11]. However, the selected test cases are the most important of a reduced suite. A test suite can be changed, where there are the numbers of function are requested by the developers, test team and the users. Specifically, the entire program, which contains the lines of code, may produce bugs after faults are found [12]. To the survey, the traditional regression selection concerns faults that can change the properties of the program that contains with many lines of code [13]. Unfortunately, many techniques are working due to the assumptions of the numbers of function are solved by the test team already before coming to the part of a test case selection. Therefore, in the future works, many researchers are trying to concerns those three factors (numbers of function, lines of code, and faults) including the other factors, such as the structure of source code and a structure of the entire system.

A. Lawanna is Lecturer at Assumption University, Bangkok, 10240 Thailand (phone: 662-719-1079; fax: 662-719-1639; e-mail: adtha@scitech.au.edu).

II. Related Works

A. Retest-All Selection

The oldest and simplest technique of regression test selection is the retest-all selections. It is the technique that simply reuses all existing test cases in test suite and selected test case, this technique “chooses" all test cases in *T* but failure to preserve the faultless. This technique is very appropriate when the size of a source code is proper. In order to measure the size, it depends on the developers’ judgment. The problem starts when the size is getting bigger that causes running time increases. Unfortunately, that there are no reports about what size is called small or proper size. Another reason is about running time; it may refer to time consuming in searching data inside database or may be executing time for checking bugs in any lines of code. That’s why, the retest-all techniques cannot response to faultless and time constraints, but no test selection tools are available, developers often select test suite based on “hunches", or loose associations of test suite with functionality, line of codes and faulty versions [14].

B. Random/Ad-Hoc Selection

It randomly chooses some number of test cases from test suite. The random algorithms can be varied by human judgments. This technique claims that it is a fast selection, which depends on random functions. Particularly, the different numbers of random selection are required in one experiment. One of the majors studying with this technique is to observe, in which, what is the suitable random numbers that can reduce the maximum numbers of test cases. Besides this, it is also required to reduce the faults in a source code after running. However, we found that this technique cannot guarantee the abilities of reduction and faultless rate [15].

C. The Safe Test Technique

This paper focuses Rothermel and Harrold's regression test selection tool because their results are better than the retest-all and random/ad-hoc selection. This technique can be used to construct the control flow graphs for a program or procedure and its modified program and uses the flow graphs to select test cases that execute the revised code from the original test suite. They describe that, under certain conditions, the set of test cases their technique selects includes every test case from the original test suite that can expose faults in the modified program or procedure. Particularly, although their algorithms may choose some test case that cannot expose faults, they are at least as accurate as other safe regression test selection techniques. Unlike many other regression test selection techniques, their algorithms can handle all types of program modifications and all language constructs. They have implemented their algorithms; initial empirical studies prove that their technique can significantly reduce the cost of regression testing modified program [16].

D. Subject Programs

In this paper, the eight subject programs, with a number of modified versions and the test suites for each program are provided. The programs are from two sources: a group of seven programs collected and constructed initially by Rothermel and Harrold and an interpreter for an array definition language, used within a large aerospace application, space. Table I shows the details of the subject programs, in which those programs are originated by Hutchins and team. These programs are written in C, and size varied from 138 to 516 lines of code. They applied a test pool of black-box to generate these programs which test cases using the category partition method with Siemens Test Specification Language tool. Afterward, they applied additional white-box technique to ensure that each exercisable statement, edge, and also definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. Hutchins and team also generated faulty versions of each program, which varied between 7 and 41 versions by modifying existing code in the base version; in most the test cases they provided a single line of code whereas in a few cases they changed between 2 and 5 lines of code. Then, they discarded the modifications that they realized either very easy to determine the changes (e.g., found by more than 350 test cases in each test suite) or very difficult to find the fewer than three test cases) with their previously created test cases. Another program, Space has been used as a subject for several regression test selections. As Table I describes, it contains 136 C functions and 6,218 lines of code. Each of the program has 33 versions contains a single fault that can be discovered while developing the program [16].

TABLE I
THE SUBJECT PROGRAMS

Name	*n*	*l*	f
print-tokens	18	402	7
print-tokens2	19	483	10
replace	21	516	32
schedule	18	299	9
schedule2	16	297	10
space	136	6218	38
tcas	9	148	41
totinfo	7	346	23

The subject programs from Table I are often used for the research area on techniques of choosing the test cases, e.g., regression test selection, minimization, and prioritization.

III. Proposed Methods

A. Standardize the Requirements

According to the subject program as data set used throughout this paper, the main requirements are standardized at the first step of the proposed methods.

Definitions in this step are provided as follows; *N*: the number of functions; *x*: the elements of *N; F*(*x*): the neighborhood of *x*; *X*\: the standardized of *N*. Given a program, let $x \in X$ be a requirement. Denote that *F(x)* the neighborhood of x and $F(X_l) = U_{x \in X_l} F(x), F(x_i)$ contains requirements which are close to some other requirements in x_l . Suppose that *P* is a frequency function of the requirement

running from 0 to 100%; equivalently there is a distribution of requirement. Then the requirement size of x_l is defined as

$$Size(X_l) = \frac{\sum_{x \in F(X_l)} P(x)}{\sum_{x \in X} P(x)} \quad (1)$$

The benefit of a requirement x with respect to a set, W of the requirements is determined as;

$$BFT(x) = \sum_{y \in U_{i=1}^{k} F(x_l) - F(W)} P(y) \quad (2)$$

where

$$F(W) = U_{w \in W} F(w) \quad (3)$$

The benefits of a requirement set or $\{x_1, x_2, x_3, ...x_k\}$ is defined as;

$$\sum_{y \in U_{i=1}^{k} F(x_l - F(W))} P(y) \quad (4)$$

The procedures of the standardization of the number of functions are described as;

1. Determine the neighborhood *F(x)* for every requirement $x \in F$
2. Set $x_1 = \phi$.
3. Select a number of function from *F* — *X* with the maximal benefit with respect to *F(X\)* and add it to *X*.
4. Repeat step 3 until *F*(*Z*) - *F*(*Xi*) is empty or X_1 has *k* elements.

Remark: the size of the test cases and the benefit are used. In fact, the benefit can be defined on other notions as long as it takes the concept of usefulness.

B. Determine the Test Cases

In response to step 1, the integral technique is used to integrate *F(X\)*with respect to (f,*l*,*n*) as;

$$T = \int_0^n \int_0^l \int_0^{\mathrm{f}} F(X\backslash) dn dl d\mathrm{f} \quad (5)$$

where, over a particular (*n*,*l*), the variable f is restricted between *a(n,l)* and *b(n,l)* and, for a particular *n*, the variable *l* is restricted between *c(n)* and *d(l)*. The number of functions can be changed any time depends on the user requirements, programmers, and test team. Those requirements cause inefficient of the modified code, including lines of code can affect the faulty version (e.g., bugs or faults). Example of the computation;

$$T = \int_0^n \int_0^l \int_0^{\mathrm{f}} F(X\backslash) dn dl d\mathrm{f}$$

$$T = \int_0^n \int_0^l \int_0^{\mathrm{f}} (n + l + \mathrm{f}) dn dl d\mathrm{f}$$

$$T = \int_0^l \int_0^{\mathrm{f}} (\frac{n^2}{2} + nl + n\mathrm{f}) dl d\mathrm{f} \quad (6)$$

$$T = \int_0^{\mathrm{f}} (\frac{n^2}{2} l + n\frac{l^2}{2} + nl\mathrm{f}) d\mathrm{f}$$

$$T = (\frac{n^2}{2})(l)(\mathrm{f}) + (\mathrm{n})(\frac{\mathrm{l}^2}{2})(\mathrm{f}) + (\mathrm{n})(\mathrm{l})(\frac{\mathrm{f}^2}{2})$$

Therefore, the computations of finding the numbers of the appropriate test cases can be done by using (6).

C. Determine the Average Test Cases

This step, the average test suite is computed by (7);

$$T_{avr} = \frac{T}{T_{net}} \quad (7)$$

The value of the net test cases T_{net} is given by;

$$T_{net} = n \times l \times \mathrm{f} \quad (8)$$

Equation (8) is useful for the computation, when the total numbers of the test cases are needed.

IV. Experimental Results

A. Standardization of the Requirements

According to the scientific data, the first step is to standardize the requirements of test cases in a test suite in the different programs shown in Figs. 1-3. The results show that after the subject programs are standardized, the amounts of each requirement are reduced. Accordingly, the complexities of the modified programs are also reduced; the reason is that the smaller numbers of the requirements can reduce the executing and testing time. However, the experiments must avoid the lack of the correctness after reducing some requirements.

The standardization must concern the relevant requirements that can affect the ability of the entire software. The properties of the standardization of this paper can help the software maintainers to produce the minimum errors at approximately 30% due to (1)-(4). However, the standardization technique does not depend on only these requirements because sometime the process of software maintenance deals with other requirements such as the human judgments, the ability of the maintenance team, and the hardware-software configuration.

B. The Average Test Suite

After the standardizations of the requirements of each subject program are done then the average test cases in the test suites are defined. According to (5), it can help us to find the proper test cases. Particularly, (6) is applied to find the average test cases in a test suite which needs the net test cases from the original requirements that normally can be computed by the multiplication of related requirements. The random

technique and the safe regression test selection are preceded and used for the evaluation. Moreover, the average test cases of the software based on these techniques are shown in Fig. 4. It shows that the average test cases of the random technique are lower than the results of the safe test.

C. Reduction Rate

The first contribution of this paper is the value of the reductions after selecting the test cases for the process of maintenance is higher than the traditional techniques. The formula that is used for calculate the value of the reduction follows;

Fig. 5 shows the comparison between these techniques, including the MT. It gives the higher reduction of test cases than others except the result of the space program. The random technique works well in this program this is because the more complexity caused by the higher number of the requirements can be handled by simply selection.

D. The Ratio between the Number of Test Cases and Faulty Versions

The ratio between the number of test cases and faulty version is one of the evaluations that can help the software maintainers to concern the possibilities of find the faults in the modified program after using the different regression techniques. Relevant to Fig. 6, it shows that the least of finding the faults can be found in the MT. According to the results, it guides the maintainers to concerns the influences of the related requirements in the process of testing software before running through the maintenance process. In fact, one of the limitations of maintain the program, including modifying source codes, depend on the understanding the changes made by users, programmers, testers, and maintainers. From the survey, many techniques concern only the lines of code and the effect from the faulty versions before selecting the test cases for the next process. This may not handle some of the most important requirements such as the number of functions. This is the reason why a model for test case selection is proposed.

From the whole picture of the abilities of the comparative studies, the MT gives the better capability than others. This is because;

(1) It offers the smallest size of the test cases.
(2) The MT gives the maximum of the reduction rate.
(3) The numbers of faults are smaller than others.

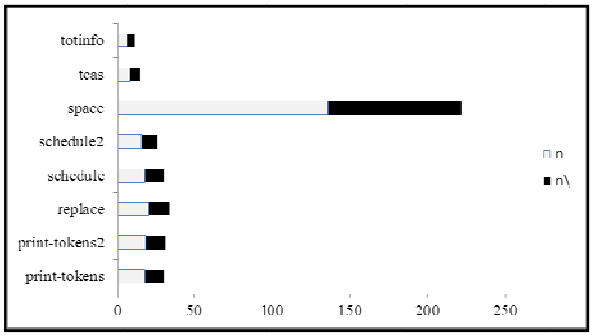

Fig. 1 Before and after the standardization of the number of functions

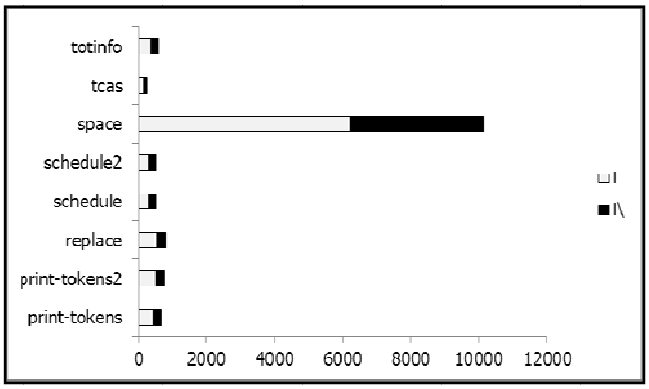

Fig. 2 Before and after the standardization of the lines of code

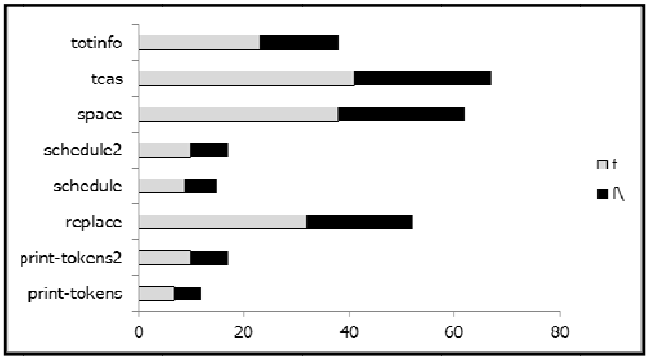

Fig. 3 Before and after the standardization of the faulty versions

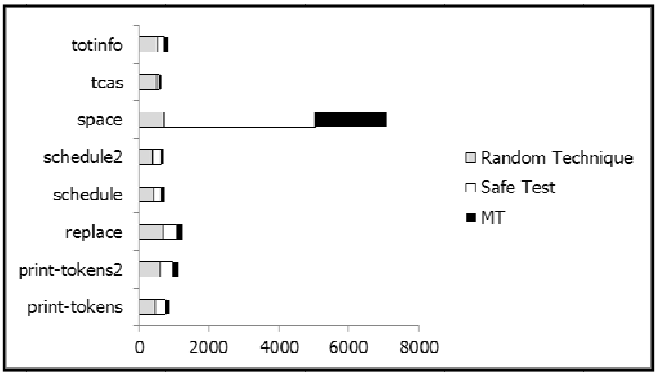

Fig. 4 The studies of the comparison techniques

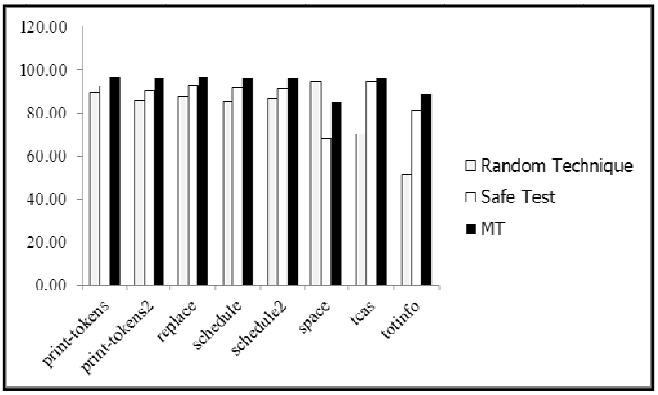

Fig. 5 The reduction of the comparison techniques

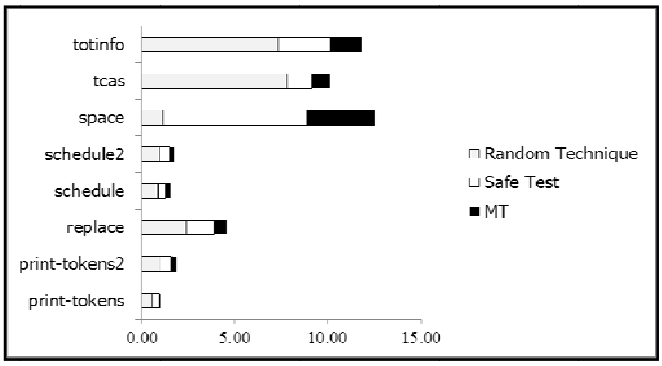

Fig. 6 The comparison of finding faults

$$\text{Re}\,duction = \frac{T_{net} - T_{avr}}{T_{net}} \quad (9)$$

V. Conclusion and Future Works

This paper contributes two benefits, which are the higher reduction and faultless when compared with the traditional technique such as a random/ad-hoc selection, and the safe test based regression test selection. However, the MT cannot guarantee that it is the best because there are many complexities involves in the process of maintenance (e.g., functions, faulty version, bugs, run time execute time, numbers of test cases and test suite size). Those factors may decrease the abilities of the entire source code while retesting, rerunning and re-debugging the programs, particularly, in all processes of maintaining software mostly spent very long time. By two main objectives, the selected test cases must not affect the performance of keeping faultless after the test suite selection. For future works, we will apply the concept of test case deletion, test case addition, or partition techniques to improve the performance of any software.

References

[1] A. Abran, and K. Nguyen, "Measurement of the maintenance process from a demand-based Perspective," *JSMR, USA*, vol. 5, no. 2, 1993, pp. 63–90.

[2] W. Royce, "Managing the development of large software systems," *9thInt. Conf. Software Engineering* USA, 1987, p. 1-9.

[3] A.M. Davis, H. Bersoff, and E.R. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," *IEEE Trans. on Softw. Eng. USA.*, vol. 14, no. 10, Oct. 1988,pp. 1462-1477.

[4] E.B. Swanson, "The dimensions of maintenance,"*2ndInt. Conf. Software Engineering*, USA, 1976, p. 492–497.

[5] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault injection experiments using FIAT," *IEEE Trans. on Comp. USA.*, vol. 39, no. 4, pp. 575–582, 1990.

[6] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: The MEFISTO tool," *IEEE Trans. on* Comp. *USA.*, vol. 39, no. 4, Apr. 1990, pp. 575-582.

[7] H.K.N. Leung, and L.J. White, "Insights into Testing and Regression Testing Global Variables," *JSMR, USA.*, vol. 2, no. 4, Dec. 1990, pp. 209-222.

[8] H. Agrawal, J. Horgan, E. Krauser, and S. London, "Incremental regression testing," *Conf. Software Maintenance*, USA., Sep. 1993, p. 348-357.

[9] M.V. Zelkowitz, D.R. Wallace, and D.W. Binkley, *Experimental Validation of New Software Technology*. Empirical Software Engineering, World Scientific, 2003,pp. 229–263.

[10] A.B. Taha, S.M. Thebaut, and S.S. Liu, "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis," *13th Conf. Computer Software and Applications*, USA., Sep. 1989, p. 527-534.

[11] V.R. Basili, and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. on Soft. Eng. USA.*, vol. 13, no. 12, Dec. 1987, pp. 1278–1296.

[12] E. Wong, and A.P. Mathur, "Fault Detection Effectiveness of Mutation and Data-flow Testing," *SQJ, USA.*, vol. 4, no. 1, 1995, pp. 69–83.

[13] G. Rothermel, and M. Harrold, "A safe efficient regression test selection technique," *ACM Trans. on Softw. Eng. USA.*, vol. 6, no. 2, Apr 1997, pp. 173-210.

[14] G. Rothermel, and M. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Trans. on Softw. Eng. USA.*, vol. 24, no. 6, Jun. 1998, pp. 401-419.

[15] F.I. Vokolos, and P.G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," *the Int. Conf. on Software Maintenance*, USA., Nov. 1998, p. 44-53.

[16] G. Rothermel, and M. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. on Softw. Eng. USA.*, vol. 22, no. 8, Aug. 1996, pp. 529-551.