WALAILAK JOURNAL
http://wjst.wu.ac.th Engineering and Physical Sciences

Technique for Test Case Selection in Software Maintenance

Adtha LAWANNA*

Department of Information Technology, Faculty of Science and Technology, Assumption University, Huamak Campus, Bangkok 10330, Thailand

(*Corresponding author's e-mail: adtha@scitech.au.edu)

Received: 2 October 2012, Revised: 12 February 2013, Accepted: 29 October 2013

Abstract

The main problem of maintaining a program is that the large number of the selected test cases can increase the executing time and maintenance cost. Therefore, the random selection, the data flow technique and a safe regression test are proposed. Unfortunately, the results of the traditional technique cannot satisfy the development team. According to this, the technique for test case selection is proposed. There are 2 main methods used, which are finding the minimum numbers of test cases and selecting the relevant test cases. It can select the smaller size of a test suite when compared with the results from using the traditional techniques. Beside this, it also gives the minimum effects from the new faults, which may occurred during the maintenance process.

Keywords: Software maintenance, regression test, test suite, test case

Introduction

The development of software contains requirements analysis, design, implementation, testing, maintenance and more [1]. Software engineers must select a development process that is applicable for the group size, threat level and application field. Tools that are well-cohesive and support the project stresses must be designated. Quantities and auxiliary tools should be used to supply as much reflective and thoughtful as possible [2].

The software development cycle comprises many steps, some of which are reiterated until the system is complete and the users' and customers' needs are fulfilled [3]. However, before obliging funds for a software development or maintenance project, a user often wants an approximation of how long the project will take and how much the project will cost [4].

This paper involves the process of maintenance system. Software maintenance concerns with handling change in this part of the software-development life cycle. Maintainers interact continually with coworkers, clients, and users in order to effectively describe problems and catch their causes. Maintainers must be good detectives, testing software thoroughly and pursuing the sources of failure [5].

Normally, the regression test is used in the process of software maintenance although it is acknowledged as an expensive action. It requires large amounts of testing time as well as effort, and accounts for almost half of the software maintenance costs. Minimization of regression test strength is, therefore, the topic of considerable practical significance, and has the potential to substantially decrease software maintenance costs [6]. Regression test selection (RTS) techniques select a subset of effective test cases from an initial test suite (T) to test that the affected but unmodified parts of source code continue to work correctly [7,8]. Procedure of the current regression test selection technique can help reduce the testing costs in the environmental changes in which a source code undergoes numerous modifications. Regression test selection basically consists of 2 major accomplishments: (1) Determination of the affected portions - This includes determination of the unmodified parts of the source code that are affected by its

modifications; and (2) Test case selection - This includes determination of a division of test cases from the initial test suite T which can efficiently test the unmodified parts of the source code [9]. The purpose is to be capable to select the division of test cases from the initial test suite that has the potential to identify faults persuaded on account of the modifications [10]. Rothermel and Harrold have formally determined the regression test selection problem as follows: Let P be an application program and P' is a modified type of P. Let T be the test suite established initially for testing P. An RTS technique targets to select a division of test cases $T' \subseteq T$ to be fulfilled on P', such that every fault identified when P' is performed with T is also identified when P' is performed with T'. Leung and White have perceived that the procedure of an RTS technique can reduce the cost of regression testing compared to the retest-all approach, which includes running the whole test suite T to revalidate a modified program P', only if the cost of choosing a compact subset of test cases to be run on P' is not as much of the cost of running the tests that the RTS technique overlooks [11]. The retest-all tactic is considered impractical on account of resource, cost, and provide schedule limitations that projects are frequently subjected to. Another tactic is to randomly select test cases from T to accomplish regression testing [12]. Nevertheless, random selection of test cases can fail to expose many regression errors. RTS methods aim to overcome the disadvantages related with the retest-all tactic and in random selection of test cases by exactly choosing only those test cases that test the unmodified but affected parts of the source code [13]. However, selection of test cases based on human judgment inclines to become unsuccessful and defective for large software products. Even for abstemiously complex systems, it is typically tremendously difficult to manually classify test cases that are applicable to a change [14]. This method often clues to a large number of test cases being selected and rerun even for small changes to the original source code, leading to unreasonably high regression testing costs. Another problem that exteriors during regression testing stems from the fact that are typically supplied with only the functional explanation of the software, and therefore lack adequate information of the code to exactly select only those test cases that are related to a modification [15]. Many RTS techniques have appeared for procedural and object-oriented programs, each meant at leveraging definite optimization selections. RTS techniques have been planned by several authors.

Data flow Technique simulated a tool by manually examining program medications, and creating a list of tuples that denote the definition use pairs, if it had been deleted from P in obtaining P'. A data flow testing tool can find the test cases in the test suite for each for each version of that program. For each version, maintainers create dataset of selected test cases T' that contained all such test cases [16].

Safe Technique by an implementation of Rothermel and Harrold's regression test selection algorithm, implemented as a tool named DejaVu which builds control flow graph representations of the procedures in 2 programs P in obtaining P', in which singular nodes are categorized by their corresponding accounts. The method assumes that a test history is available that records, for each test case t in T including each edge e in the control flow graph for P, whether t crossed e. This test history is collected by arrangement code that is introduced into the source code under test [17].

Random Technique creates a tool that, given a selection percentage n and a test suite T, randomly chooses n % of the test cases from T, outputting T', a selected test suite containing only the selected test cases [13]. Retest-All Technique required no implementation. Therefore, the technique for test case selection (TTCS) is proposed to respond these problems. The expectation of this technique is to produce the smaller numbers of test cases compared to RTS techniques mentioned above.

Materials and methods

To reach the objectives stated in the introduction, 8 subject programs (**Table 1**) are used in the experiment for which faults and test pool is available. These programs are being used in regression test selection, minimization, and prioritization. Particularly, Test suites are then designed by random selection of the test pools for each subject program.

Dataset

A summary of the properties of the 8 programs is found in **Table 1**. "*L*" here is the lines of code, i.e., lines of code, which are shown after comments have been unconcerned. "*N*" is the number of functions (if, while, case, etc.). And "*F*" refers to the faulty versions. The experiments need a set of 8 well-known subject programs written in C. The first is the program usually mentioned as Space, established at the European Space Agency and first applied for testing technique evaluation resolves by Frankl *et al.* [16]. The last 7 are developed by the Siemens suite of programs with hand scattered bugs or faults, first applied by Hutchins *et al.* [18] to compare data flow-based coverage criteria and control graph flow-based. The artifacts of all 8 programs have subsequently been adapted and prolonged by other investigators, particularly Rothermel and Harrold [6] and Graves *et al.* [11]. These programs are chosen because of the maturity of the related artifacts, and because of their historical significance. Many high-quality experimental software engineering articles have applied the Siemens suite and Space. Briefly, Space is related with 38 faulty versions (real bugs), 33 by the original research of Frankl. Each faulty version relates to a fault that was corrected "during testing and executing the program" [16].

Table 1 Dataset.

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	6,218	38
Tcas	9	148	41
Totinfo	7	346	23

Methods

Method 1: Finding the total numbers of test cases

The number of test cases in a test suite is determined by using *N*, *L*, and *F*. The computation is set up by the triple integral to integrate $f(N,L,F)$ with respect to *F* first, next with respect to *L*, and afterwards with respect to *N*, therefore, the general form is;

$$\sum T = \int_0^n \int_0^l \int_0^{\mathrm{f}} f(N,L,F)dndld\mathrm{f} \tag{1}$$

where, over a specific (*N*, *L*), the variable *F* is restricted between *g(*N,*L)* and *h(N,L)* and, for a precise *N*, the variable *L* is restricted between *s(N)* and *t(L)*.

Moreover, 5 studies are processed as follows;

$$\sum T = \int_0^n \int_0^{\mathrm{f}} \int_0^{\mathrm{l}} f(N,F,L)dnd\mathrm{f}ldl \tag{2}$$

$$\sum T = \int_0^{\mathrm{f}} \int_0^{\mathrm{n}} \int_0^{\mathrm{l}} f(F,N,L)d\mathrm{f}dndl \tag{3}$$

$$\sum T = \int_0^{\mathrm{f}} \int_0^{\mathrm{l}} \int_0^{\mathrm{n}} f(F,L,N)d\mathrm{f}d\mathrm{l}d\mathrm{n} \quad (4)$$

$$\sum T = \int_0^{\mathrm{l}} \int_0^{\mathrm{f}} \int_0^{\mathrm{n}} f(L,F,N)d\mathrm{l}d\mathrm{f}d\mathrm{n} \quad (5)$$

$$\sum T = \int_0^{\mathrm{l}} \int_0^{n} \int_0^{\mathrm{f}} f(L,N,F)d\mathrm{l}dnd\mathrm{f} \quad (6)$$

Algorithm of finding the total numbers of test cases;

Step1: $\sum T = \int_0^{n} \int_0^{l} \int_0^{\mathrm{f}} f(N,L,F)dnd\mathrm{l}d\mathrm{f}$ *where* $0 \le N \le n$

Step2: $\sum T = \int_1^{l} \int_1^{\mathrm{f}} f(L,F)d\mathrm{l}d\mathrm{f}$ *where* $0 \le L \le l$

Step3: $\sum T = \int_0^{\mathrm{f}} f(F)d\mathrm{f}$ *where* $0 \le F \le \mathrm{f}$

Step4: $T_{avr} = \dfrac{\sum T}{T_{net}}$ (7)

where T_{net} is the total numbers of possible test cases (*N*L*F*) found in each test suite. Accordingly, T_{avr} can give the appropriate numbers of the test case for the maintenance process.

Method 2: Selecting the test cases
Algorithm of selecting the test cases
Step1: Check frequency of testing test cases.
Step2: Select a test case with the highest frequency value.
Step3: Do step 1 & 2 until the numbers of selected test case equal T_{avr} value.

Obviously, all computations of the different 6 studies are tested as; $f(N,L,F)$, *f(N,F,L)*, *f(F,N,L)*, *f(F,L,N)*. *f(L,N,F)*, and *f(L,F,N)*, Those studies give the same results as equal as the result computed from $f(N,L,F)$.

Results and discussion

According to the scientific data, the first step is to find the net of test cases in any test suite in the different program by the safe test regression technique described in **Table 2**.

Table 2 The average test cases in each test suite

Name	***N***	***L***	***F***	T_{avr}
Print-tokens	18	402	7	4,130
Print-okens2	19	483	10	4,115
Replace	21	516	32	5,542
Schedule	18	299	9	2,650
Schedule2	16	297	10	2,710
Space	136	6,218	38	13,585
Tcas	9	148	41	1,608
Totinfo	7	346	23	1,052

Table 3 shows the average test cases for the 8 subject programs due to the random selection (RD), the data flow technique (DF), a safe regression test (SRT), and the technique for test case selection (TTCS).

Table 3 Numbers of test cases by several methods.

Name	***RD***	***DF***	***SRT***	***TTCS***
Print-tokens	382	486	318	205
Print-okens2	299	452	389	247
Replace	426	407	398	275
Schedule	483	430	225	155
Schedule2	57	456	234	154
Space	71	394	4361	3,129
Tcas	203	475	83	95
Totinfo	214	433	199	185

Reduction rate

Several of the studies on software maintenance concerning reduction of the number of test cases are single compared to retest all with the only conclusion that removing some test cases can be practiced. This is a problem examined in experimental studies in general. Particularly, they evaluate time reduction in small programs and the size of the differences need to be measured in milliseconds. From the survey, few of the studies consider both fault detection and cost reduction. Therefore, one of the objects set in many studies is to reduce many test cases without or with very low fault. Then Eq. (8) is derived to identify the reduction rate.

$$\text{Re}\,duction \quad Rate = \left(\frac{T_{net} - T_{ave}}{T_{net}} \right) \tag{8}$$

Obviously, numbers of studies concern higher reduction rate in order to maintain test case size in the software maintenance system. **Table 4** shows the reduction rate of each program by the couple of techniques. We can see that TTCS can reduce more numbers of test cases in the different test suite than the traditional techniques.

Figure 1 shows the graph of reduction rate of selected techniques and TTCS which is concise to **Table 4**.

Table 4 Comparison of reduction rate.

Name	***RD***	***DF***	***SRT***	***TTCS***
Print-tokens	0.8831	0.8939	0.9230	0.9500
Print-okens2	0.9159	0.8945	0.9055	0.9400
Replace	0.9076	0.9110	0.9282	0.9500
Schedule	0.9483	0.8268	0.9151	0.940
Schedule2	0.9111	0.8542	0.9137	0.9430
Space	0.5730	0.5708	0.6790	0.7700
Tcas	0.7755	0.7295	0.9484	0.9410
Totinfo	0.8080	0.8088	0.8108	0.8240

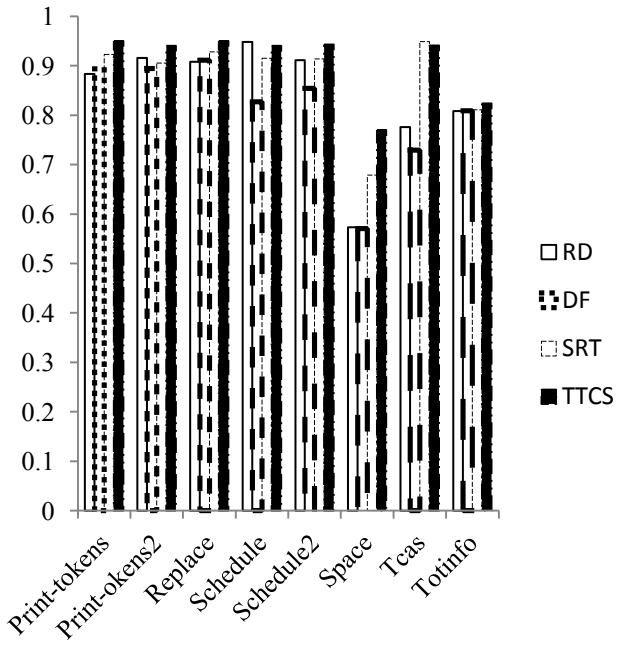

Figure 1 Graph of reduction rate.

Fault rate

As widely recognized, it is nearly impossible to produce faultless code. Therefore, a solving fault in software (e.g. debugging) becomes a critical task in the software-development life cycle in part of maintenance.

Typically, fixing bugs involves 2 steps: it firstly is to find the location of the fault, and then replace the faulty statement(s) with the proper one(s). Frequently, it is not easy to construct the suitable substitute statements, for this step is about human judgments. Therefore, the traditional approaches to fixing bug automation mainly focus on fault localization, especially to reduce the number of delivered faults.

However, debugging software is an expensive and mostly depends on human judgments. From the experiments, we found that the fault rate follows Eq. (9);

$$Fault \quad Rate = \left(\frac{F - \mathrm{f}}{F} \right) \tag{9}$$

where F is a number version of fault faulty and f is deleted fault.

Table 5 Fault rate.

Name	***RD***	***DF***	***SRT***	***TTCS***
Print-tokens	0.1169	0.1061	0.0770	0.0500
Print-okens2	0.0841	0.1055	0.0945	0.0600
Replace	0.0924	0.089	0.0718	0.0500
Schedule	0.0517	0.1732	0.0849	0.0580
Schedule2	0.0889	0.1458	0.0863	0.0570
Space	0.4270	0.4292	0.321	0.2300
Tcas	0.2245	0.2705	0.0516	0.0590
Totinfo	0.1920	0.1912	0.1892	0.1760

The results in **Table 5** show that RD cannot guarantee the performance of reducing fault after the selection is done. We can keep the ability of debugging by SRT and TTCS. According to this, Regression Test is still being developed for the enhancement by different techniques (e.g. Minimization and Prioritization).

Figure 2 depicts the graph of fault rate of selected techniques and TTCS. It shows the effectiveness of fault detection under 4 techniques.

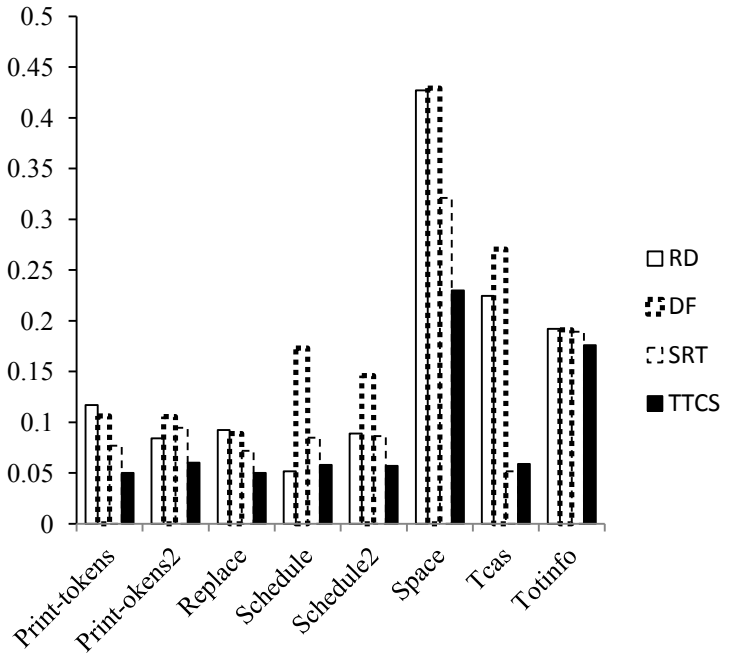

Figure 2 Graph of fault rate.

Conclusions

This paper contributes 2 benefits, which are the higher reduction rate of selecting adequate test cases or deleting redundancy test cases in the test suites, and in the meantime we can keep the satisfied faultless rate. However, we cannot summarize that our technique is the best because there are several factors (e.g. functions, faulty version, bugs, run time execute time, numbers of test cases and test suite size). Those factors affect the process of software maintenance while retesting, rerunning and re-debugging the programs, all of which particularly take very long time. By 2 main objectives, the selected test cases must not affect the performance of keeping faultless after the test case selection. For the future work, the standardization of several functions should be realized before applying the integral technique because it can help the maintainers to authorize the most and the least functions that directly affect the maintenance process.

References

[1] A Abran and H Nguyemkim. Analysis of maintenance work categories tough measurement. *In*: Proceeding of the Conference on Software Maintenance, Los Alamitos, CA, USA, 1991, p. 104-13.
[2] RS Arnold. A road map guide to software re-engineering technology. *In*: Proceeding of the Conference on Software Reengineering, Los Alamitos, CA, USA, 1993, p. 3-22.
[3] M Dawson. Iteration in the software process. *In*: Proceeding of the 9th International Conference on Software Engineering, Los Alamitos, CA, USA, l987, p. 36-4l.
[4] NF Schneidewind. The state of software maintenance. *IEEE Trans. Software Eng.* 1987; **13**, 303-10.
[5] G Alkhatib. The maintenance problem of application software: an empirical analysis. *J. Software Mainten. Res. Pract.* 1992; **4**, 83-104.
[6] G Rothermel. A safe efficient regression test selection technique. *ACM Trans. Software Eng. Meth.* 1997; **6**, 173-210.

[7] G Rothermel and MJ Harrold, Analyzing regression test selection techniques. *IEEE Trans. Software Eng.* 1996; **22**, 529-51.
[8] FI Vokolos and PG Frankl. Empirical evaluation of the textual differencing regression testing technique. *In*: Proceeding of the International Conference on Software Maintenance, Bethesda, MD, USA, 1998, p. 44-53.
[9] WE Wong, JR Horgan, S London and AP Mathur. Effect of test set size and block coverage on the fault detection effectiveness. *In*: Proceeding of the 5th International Symposium on Software Reliability Engineering, Washington, DC, USA, 1994, p. 230-8.
[10] MJ Harrold, R Gupta and ML Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Software Eng. Meth.* 1993; **2**, 270-85.
[11] MR Garey and DS Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Vol I. W.H. Freeman, New York, 1979, p. 147-58.
[12] HK Leung and LA White. A cost model to compare regression test strategies. *In*: Proceeding of Software Maintenance, Ottawa, ON, Canada, 1991, p. 201-8.
[13] WE Wong, JR Horgan, AP Mathur and A Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *In*: Proceeding of the 21st Computer Software and Application Conference, Morristown, NJ, USA, 1997, p. 522-8.
[14] JM Voas, PIE: A dynamic failure-based technique. *IEEE Trans. Software Eng.* 1992; **18**, 717-27.
[15] TL Graves, MJ Harrold, MJ Kim, A Porter and G Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Software Eng. Meth.* 2001; **10**, 184-208.
[16] PG Frankl and SN Weiss, An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Software Eng.* 1993; **1**9, 774-87.
[17] MJ Harrold, L Larsen, LJ Lloyd, D Nedved, M Page, G Rothermel, M Singh and M Smith. Aristotle: a system for the development of program-analysis-based tools. *In*: Proceeding of the 33rd Annual on Southeast Reginal Conference, New York, USA, 1995, p. 110-9.
[18] M Hutchins, H Foster, T Goradia and T Ostrand. Experiments on the effectiveness of dataflow and control flow-based test adequacy criteria. *In*: Proceeding of the 16th International Conference on Software Engineering, Los Alamitos, CA, USA, 1994, p. 191-200.