Chiang Mai J. Sci. 2013; 40(4) : 689-700
http://it.science.cmu.ac.th/ejournal/
Contributed Paper

Methods for Test Suite Selection in the Process of Software Maintenance

Adtha Lawanna
Department of Information Technology, Faculty of Science and Technology, Assumption, Bangkok, Thailand.
*Author for correspondence; e-mail: adtha@scitech.au.edu

Received: 25 September 2012
Accepted: 17 June 2013

ABSTRACT

Software maintenance is one of the essential processes of Software Development Life Cycle. The main ideas of maintaining software concern the correction of errors, the modification of codes, the prevention of future errors, and the improvement in executing time. While the modification has been implementing, the software system has to be retested to increase a level of confidence that it will be organized due to the requirements specification. At this point, a test suite needs to be designed for testing the modified modules and the entire software. A concept of the test suite selection is being developed by regression test selections such as the retest-all selections, a random or ad-hoc selection and the test suite minimization. Those selection techniques apply 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 factors that can affect the size of test suite but including the number of functions and faulty versions. Therefore, the method for test suite selection is proposed in order to cover those three factors by the integral technique which can produce the smaller size of a test suite when compared with the traditional regression selection techniques.

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

1. INTRODUCTION

Numbers of software are being developed for the world of business, including education and industry but no perfect software is produced in which all software deals with the environmental changes all the times [1]. So, software maintenance process is one of the most important following issues in the software development Lifecycle (SDLC) [2]. In SDLC, getting requirements is the first process that collects all needs and wants of users. The second process deals with the analysis. After this, the developers design the suitable software in the third process, then programmers write the program, test and install program are continuous processing. The last process in SDLC is to maintain the software which is the most interested topic in this paper. Many researchers take seriously to do the research on this topic [3,4]. They want to find the better ways to maintain software for dealing with the environmental changes. From the survey, there are four types of

software maintenance described as follows; *Adaptive* software maintenance is any effort that is the outcome of changes in a software application's operating environment. *Corrective* software maintenance is a changing a software application to reduce errors. *Preventive* software maintenance is denoted as maintenance performed for the goal of preventing problems. *Perfective* software maintenance is to improve quality of maintenance , ability, or other benefits of a computer application [4].

One of the major problems of software maintenance is to perform a suitable test suite that contains a set of test cases used for testing bugs, functions, and faults [5]. If test suite size is very large, then testing and executing time increases, this can reduce the performance of the entire system. Another problem after test suite selection may refer to faults or bugs that can be occurred in a modified source code, particularly, when some relevant test cases with faultless are not in the selected test suite [6]. According to this, regression test selections are being proposed. In general, there are two main strategies in regression test selection explained as follows; regression test minimization involves removing irrelevant test cases; and regression test prioritization can rank test cases into small groups and selecting the most relevant test cases. Of course, those techniques can reduce test suite size but may not handle any fault after selection [7,8].

This paper studies the traditional techniques of regression test selection in which retest-all selection, random/ad-hoc selection, and the interaction-based test-suite minimization, ITSM [9]. The record shows that the retest-all technique is simplest, but it introduces the maintenance cost because all test cases are retested. In the meantime, the random/ad-hoc can reduce the running time, but it cannot handle some faults when randomly selecting irrelevant test cases instead of the relevant test cases [10]. The interaction-based test-suite minimization, ITSM can provide the smaller numbers of test cases and offers the better way to handle bugs and faults than the traditional techniques. However, it still produces a larger number of the test cases. Therefore, the Method for Test Suite Selection (MTSS) is proposed to produce the better results compared with those traditional regression techniques. The challenge of MTSS is that it integrates three factors; the number of functions, lines of code, and faulty versions, and generate the algorithm that can produce the smaller number of a suitable test suite, including in provides the higher percentage of reducing faults compared with retest-all, random/ad-hoc and interaction-based test-suite minimization, ITSM.

Definitions

There are several technical terms used in this paper. Table 1 describes the symbols, key terms, particularly, description of those, including the set representative of them.

Regression Test

Basically, the processes of the regression test are used, as shown in Figure 1. The requirement specification goes through a whole program then the programmers modify the old program. At this time, the modified program is produced and tested. Normally, the software testers use the automated test case generation to generate the test suites, in which contain numbers of test cases [10].

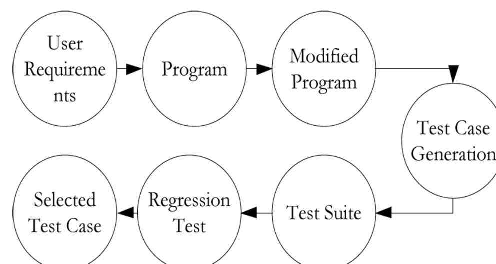

Figure 1. The methods of regression.

Table 1. Definitions.

Symbol	Key term	Description	Representative
T	total test suite	total test cases that conclude the relevant and irrelevant test cases	$T = \{T_1, T_2, T_3, ...T_n\}$
T^*	suitable test suite	the suitable or relevant test cases that are created by test case generator	$T^* = \{T_1, T_2, T_3, ...T_m\}$
T'	selected test cases	the set of the selected test cases which are chosen from T^*	$T' = \{T_1, T_2, T_3, ...T_s\}$
N	numbers of function	user requirements that affect to the entire software	$N = \{n_1, n_2, n_3, ...n_r\}$
L	lines of code	size of the estimated program	$L = \{l_1, l_2, l_3, ...l_s\}$
F	faulty versions	the modified software	$F = \{l_1, l_2, l_3, ...l_s\}$
b	bugs	faults or bugs that can be produced during the process of testing the software	-
$ITSM$	-	the interaction-based test-suite minimization	-
$MTSS$	-	methods for test suite selection which is the proposed model	-

Retest-All Selection

The oldest and simplest technique of regression test selection is the retest-all selections. It simply reuses all existing test cases in a test suite, this technique effectively “selects” all test cases in T but executing time is a problem [14].

Unfortunately, that there are no reports about what size is called small or proper. Another problem is a long-running time, especially, in the process of searching data inside database, including a long-time for checking bugs in any lines of code [15].

Random/Ad-Hoc Selection

The random/ad-hoc selection randomly selects some number of test cases from each 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. One of the majors studying this technique is to observe, in which, what is the suitable random numbers that can reduce the maximum numbers of test cases. Particularly, the different random numbers are tested in one experiment [16].

Unfortunately, that there are no reports about what size is called small or proper. Another problem is a long-running time, especially, in the process of searching data inside database, including a long-time for checking bugs in any lines of code [17].

Interaction-Based Test Suite Minimization (ITSM)

ITSM is the technique that can reduce a given test suite which can be created by test case generator based on the concept of a safe regression test by Rothermel and Harrold’s. But it can avoid the impact of the coverage of feature interactions. ITSM uses much less modeling effort, and does not need a definition of restrictions. It is useful where there has been a significant investment in an old test suite, where building new tests is high cost, and where restrictions are redundancy. Interaction-based test-suite augmentation, or enhancement, is the technique of adding tests

to the old test suite in order to produce full interaction coverage.

ITSM produces the set of existing tests from the previous software. The algorithm of ITSM can create the represented test as tuples of values to parameters. For example, a test suite is written in much less structured form, from relatively structured spreadsheets which can be converted to free text. According to this, automatic, or semi-automatic, tool support for translation of tests from such unstructured forms into tuples of parameter values is signicant for the performance of the ITSM approach [9].

2. MATERIAL AND METHODS

2.1 Subject Programs

In the experiment, we used eight C programs, with a number of modified versions including test suites for each program. The programs come from two sources: a group of seven C 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 2 shows these programs (e.g., number of functions, lines of code, faulty versions and test pool size).

Obviously, the size of the programs relevant to their number of functions, lines of code, and fault versions. For example, the program name "space" contains 6,218 lines of code, 136 functions and 41 faults. The definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases.

Table 2. The subject programs.

Program Name	Number of Functions	Lines of Code	Faulty Versions
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

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 test cases, they provided a single line of code, while in a few cases they modified between 2 and five lines of code.

Then, they discarded modifications that they realized either extremely easy to find (e.g., found by more than 350 test cases in each test suite) or extremely difficult to find (e.g., found by 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. It contains 136 C functions and 6,218 lines of code. Each of the programs has 33 versions contain a single fault that can be discovered while developing the program [18,19,20,21].

2.2 Proposed Methods

Step 1: Finding a test suite

The objective of this step is to determine the numbers all of test cases for each test suite in the subject programs which can be computed by integrating the numbers of function, lines of code and faulty versions. Those simple interactions are complex, and

not predictable from the study of the

-individual agents themselves. This step presents the case studies that evaluate the effectiveness of the algorithm on the subject programs. In response to this, the integral technique is used to integrate with respect to f first, and afterwards with respect to *l*, subsequently with respect to *n*; therefore, the general form is;

$$T = \int_0^n \int_0^l \int_0^{f} f(n,l,\mathrm{f})\,dn\,dl\,d\mathrm{f} \qquad (1)$$

; where, over a particular *(n,l)*, the variable f is restricted between *g(n,l)* and *h(n,l)* and, for a particular *n*, the variable *l* is restricted between *s(n)* and *t(l)* pictured in Figure 2. 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).

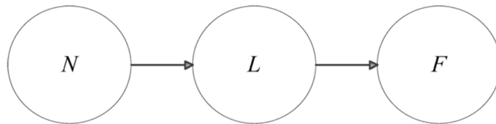

Figure 2. The main factors used in the process of maintaining software.

The most important activity of this step is to design the algorithm of finding the test cases in a test suite. Therefore, the Equation 1 is used in the algorithm, and it can be the alternative technique in which to find the numbers of the test cases in a test suite.

Algorithm of finding a test suite

If there is a test suite with $T = \int_0^n \int_0^l \int_0^{f} f(n,l,\mathrm{f})\,dn\,dl\,d\mathrm{f}$ *where* $0 \le N \le n$ *then*

Select a test suite with $T = \int_0^n \int_0^l \int_0^{f} f(n,l,\mathrm{f})\,dn\,dl\,d\mathrm{f}$

ElseIf there is a test suite with $T = \int_0^l \int_0^{f} f(l,\mathrm{f})\,dl\,d\mathrm{f}$ *where* $0 \le L \le l$ *then*

Select a test suite with $T = \int_0^l \int_0^{f} f(l,\mathrm{f})\,dl\,d\mathrm{f}$

ElseIf there is a test suite with $T = \int_0^{f} f(\mathrm{f})\,d\mathrm{f}$ *where* $0 \le F \le \mathrm{f}$ *then*

Select a test suite with $T = \int_0^{f} f(\mathrm{f})\,d\mathrm{f}$

EndIf

According to the algorithm used in this context, first is to integrate *f(n,l,*f*)* from *N,* which varied from 0 to *n*. Then, *f(l,*f*)* is integrated where *L* varied from 0 to *l* and the last integral is to integrate *f(*f*)* where *F* varied from 0 to f.

This algorithm shows how to compute and find the numbers the relevant and irrelevant test cases.

Example 1: Finding a test suite when assume that the function, *f(n, l,* f) = *n*+*l*+f defined by $0 \le n \le 20$, $0 \le l \le 200$, and $0 \le f \le 2$

$$T = \int_0^{20}\int_0^{200}\int_0^{2}(n + l + \mathrm{f})\,dndld\mathrm{f}$$
$$T = \int_0^{200}\int_0^{2}\left[\frac{1}{2}n^2 + nl + n\mathrm{f}\right]_0^{20} dld\mathrm{f}$$
$$T = \int_0^{200}\int_0^{2}(200 + 20l + 20\mathrm{f})\,dld\mathrm{f}$$
$$T = \int_0^{2}\left[200l + 10l^2 + 20l\mathrm{f}\right]_0^{200} d\mathrm{f}$$
$$T = \int_0^{2}(44{,}000 + 4{,}000\mathrm{f})\,d\mathrm{f}$$
$$T = \left[440{,}000\mathrm{f} + 200\mathrm{f}^2\right]_0^2$$
$$T = 888{,}000$$

This means there are 888,000 test cases in a test suite, where $0 \le n \le 20$, $0 \le l \le 200$, and $0 \le f \le 2$.

Besides this, another situation is studied as shown in Example 2.

Example 2: Finding a test suite when assume that the function, $f(n, l, \mathrm{f}) = n+l+\mathrm{f}$ defined by $0 \le l \le 200$, $0 \le n \le 20$, and $0 \le f \le 2$.

$$T = \int_0^{200}\int_0^{20}\int_0^{2}(l + n + \mathrm{f})\,dldnd\mathrm{f}$$
$$T = \int_0^{20}\int_0^{2}\left[\frac{1}{2}l^2 + nl + \mathrm{f}l\right]_0^{200} dnd\mathrm{f}$$
$$T = \int_0^{20}\int_0^{2}(2{,}000 + 200n + 200\mathrm{f})\,dnd\mathrm{f}$$
$$T = \int_0^{2}\left[2{,}000n + 200\frac{n^2}{2} + 200n\mathrm{f}\right]_0^{20} d\mathrm{f}$$
$$T = \int_0^{2}(40{,}000 + 4{,}000 + 4{,}000\mathrm{f})\,d\mathrm{f}$$
$$T = \left[440{,}000\mathrm{f} + 4{,}000\frac{\mathrm{f}^2}{2}\right]_0^2$$
$$T = 888{,}000$$

As we can see that even the rearrangement in the process is involved, finding number of test cases and the result after using this algorithm will not be changed. Moreover, when we also compute the test suites for other five studies that can be happened to the system as follows; $f(n,\mathrm{f},l)$, $f(l,n,\mathrm{f})$, $f(l,f,n)$, $f(\mathrm{f},n,l)$, and $f(\mathrm{f},l,n)$. According to the reason above, no matter what functions are rearranged and

put into the algorithm, we still achieve the same results among the computations. According to this point, this algorithm can be used to find the integral test cases for the subject program or data set used in the experiments. Moreover, it is also can be applied to others modified programs. However, it needs a clear defined of requirement specification, especially, value of the number of function, lines of code, and faults. So far, this algorithm cannot support special requirements such as the structure of programming.

Step 2: Finding the numbers of the selected test cases

The second step is to find the numbers of the selected test cases (T') in a test suite which can be computed by using the total numbers of test cases in a defined test suite regarding step one and the specification requirements (N,L,F). The value of the selected test cases is equivalent to the average test suite which can be computed by using Equation 2;

$$T' = \frac{T}{N \times L \times F} \qquad (2)$$

Equation 2 is created to find the numbers of the selected test suite which contains the appropriate test cases for the process of software maintenance. This technique is the application in the mathematical model that also can be preceded in the practical computer simulation. It supports the developers to produce the simulation model as the optimization technique for dealing with many more software. According to the computation above, we can find the average test suite or the selected test cases as follows;

$$T' = \frac{888{,}000}{20 \times 200 \times 2}$$

$$T' = 111$$

By using Equation 2, we will select only 111 test cases to be a test suite representative for the whole test suite.

Step 3: Selecting the suitable test suite

This section is one of the most important steps of the proposed model. According to step 1, a test suite is created as $T = \{T_1, T_2, T_3,...,T_n\}$. The test suite contains large amounts of the test cases, which relies on N, L, and F. After applying step 2 for finding the numbers of the selected test cases. This can reduce the size of a test suite to be practical for maintaining the software.

Step 3.1 Defining the relevant test suite

Definition of the relevant test cases (T^*), the status of the test cases are typed as “Pass”, “Fail”, and “Not tested”. The relevant test cases should be tested with the status “Pass”. Therefore, the test cases with “Fail” and “Note tested” will not be used and typed as the irrelevant test cases. So, this step shows the set of the only the relevant test cases as $T^* = \{T_1, T_2, T_3,...,T_m\}$ Note that $T > T^*$

Step 3.2 Finding the selected test suite from T^*

The detail algorithm for finding the suitable test cases is explained as follows;

1. Create the priority of testing.

The priority of testing relies on the user requirements. This means the testers should create the test cases concise to the needs and wants from users.

2. Select the test case with the first priority.

According to, would be the first priority of selecting the suitable test case.

3. Continue Select the suitable test cases

The test cases are chosen until the numbers of the selected test case is equivalent to T'.

Therefore, the proposed model will give the suitable test cases as a selected test suite (the numbers of the test cases equal the numbers of computation from Equation 2).

Relevant to this, the proposed algorithm is probably applied during the software maintenance process. The reason is that this algorithm can select the suitable test cases in order to represent the entire test suite.

Table 3. The number of test cases in a test suite.

Name	*N*	*L*	*F*	*T*
print-tokens	18	402	7	10,814,202
print-tokens2	19	483	10	23,493,120
replace	21	516	32	98,650,944
schedule	18	299	9	7,895,394
schedule2	16	297	10	7,674,480
space	136	6,218	38	102,702,258,304
tcas	9	148	41	5,406,588
totinfo	7	346	23	10,472,728

3. Results and Discussion

The experiment was set up at the software engineering laboratory, Department of Information and Computer Science, faculty of Science and Technology, Q-10 Building, Huamak Campus, Bangkok, Thailand.

3.1 A Test Suite

According to the scientific data, the first step is to find all possible of test cases in a test suite in the different programs which rely on three factors shown in Table 3. In 1997, Rothermel and Harrold invent the automated test case generation tool to produce the test cases in a test suite for each program or software as shown in Table 4. However, there are many researchers try to improve an ability of reducing the number of test cases by performing different technique due to the concept of the regression test selection (e.g., data flow, safe technique, random-safe, or *Irus*) [20].

3.2 The Selected Test Suite

This paper shows three results of finding average test cases in a test suite described as follows; the first is a random/ad-hoc technique which uses the random number of 5,10 and 20 respectively (the method of random technique is described in [20]). The second is the interaction-based test

-suite minimization, ITSM (the process of finding test suite is found in article [9]). And the last one is the MTSS; the average test suite shown in Table 5. We can observe that a random technique produces the biggest size of test suite and there is no significant on analysis, while the ITSM and the MTSS give off the smaller numbers of the test cases in each example.

3.3 The Reduction Rate

Several of the studies on software maintenance concerning reduction of the numbers of the test cases are only compared to retest all with the conclusion that removing some test cases can be practice. 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, many of the studies set one of the objects is to reduce as many as test

cases without or affected by low faultless. Then equation 3 is derived to identify the percent of reduction.

$$\% \text{ Reduction} = \left(\frac{T^{*} - T'}{T^{*}}\right) \times 100\% \quad (3)$$

where; T^{*} is the suitable test suite and T' is the selected test suite respectively.

Table 4. The test suites performed by Rothermel and Harrold.

Name	N	L	F	T^{*}
print-tokens	18	402	7	4,130
print-tokens2	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 5. The selected test cases (T').

Name	Random/Ad-Hoc	ITSM	MTSS
print-tokens	435	250	214
print-tokens2	584	310	256
replace	687	315	285
schedule	388	225	163
schedule2	365	220	162
space	698	4100	3196
tcas	482	110	99
totinfo	513	190	188

Table 6. The comparison of test suite in several techniques.

Name	Random/Ad-Hoc	ITSM	MTSS
print-tokens	89.48	93.95	95.04
print-tokens2	85.81	92.47	94.00
replace	87.60	94.32	95.04
schedule	85.36	91.51	94.15
schedule2	86.53	91.88	94.32
space	94.86	69.82	76.97
tcas	70.02	93.16	94.09
totinfo	51.24	81.94	82.41

Table 7. The percentage of Fault.

Name	Random/Ad-Hoc	ITSM	MTSS
print-tokens	10.51	6.03	5.16
print-tokens2	14.17	7.51	6.20
replace	12.38	5.67	5.12
schedule	14.60	8.45	6.11
schedule2	13.43	8.08	5.94
space	5.13	30.17	23.52
tcas	29.91	6.78	6.09
totinfo	48.67	17.97	17.78

Table 6 shows the numbers of the reduced test suites by a random/ad-hoc, the interaction-based test-suite minimization, ITSM, and the MTSS. We can see that MTSS can select the smaller numbers of test cases in the different test suites conditioned by the subject program which detailed in Table 6. However, in the space program, a random/ad-hoc provides higher reduced test cases than others. This is because in the space program is very complex that involves the multiple human judgments, including the test team has the limitation of resources.

Moreover, it needs to deal with the complicate simulations, which influenced by many experts, especially to design the structure of related models before performing the suitable simulations.

3.4 The Percentage of Fault

The fact is that it has been recognized that it is nearly impossible to produce faultless code. As a result, 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 two steps: the first is to nd the location of the fault, and then replacing the faulty statement(s) with the correct one(s). Therefore, the traditional approaches to fixing bug automation mainly focus on fault localization. The percentage of fault follows Equation 4;

$$\% \text{ Fault} = \left(\frac{T' - b}{T^*}\right) \times 100\% \qquad (4)$$

; where b is the bugs found in the selected test cases. In this paper, the value of fault or bug is assumed at least one found in the whole test suite. This means that the only one bug can be produced. Example of finding % Fault of the program named Schedule regarding the comparative studies is described as;
Random/Ad-Hoc:

$$\% \text{ Fault} = \left(\frac{338 - 1}{2{,}650}\right) \times 100\% = 14.60$$ By

using the same methods, the results of ITSM and MTSS are 8.45 and 6.11 respectively. However, debugging is required to protect the critical problem in the whole process of SDLC. Therefore, the MTSS can give the lower % of producing faults, when compared with the traditional techniques. Equation 4 can be used for evaluation and the results of faultless are shown in Table 7; the results show that random/ad-hoc guarantees the performance of reducing fault after the selection is done when a number of random are not high. In the real-world phenomenon of software maintenance involves with many bugs or faults that can be occurred after modifying the program. Therefore, it should apply higher number of random to deal with the reason above. At the final decision, we are still cannot summarize

that what technique is the best. The MTSS can provide the better value of faultless and significant enough when dealing with three factors mentioned in this paper. However, the selection technique is often relied on the user requirements, which is the most important of the software maintenance.

4. CONCLUSION

This paper contributes two benefits, which are the higher reduction rate and faultless rate when compared with the traditional techniques that are a random/ad-hoc selection, and ITSM. 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 may affect the process of software maintenance 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 Case Based Maintenance (e.g., deletion, addition, or partition techniques) to improve the ability of the test suite selection.

REFERENCES

[1] Abran A. and Nguyen K., Measurement of the maintenance process from a demand-based perspective, *Software Maintenance and Evolution: Research and Practice*, 1993; **5(2)**: 63-90.

[2] Royce W., Managing the development of large software systems, *In Proceedings of IEEE WESCON*, 1970; 1-9.

[3] Leau Y.B., Loo W.K., Tham W.Y. and Tan S.F., Software Development Life Cycle AGILE vs Traditional Approaches, *International Conference on Information and Network Technology*, 2012; 162-167.

[4] Davis A.M., Bersoff H. and Comer E.R., A strategy for comparing alternative software development life cycle models, *Journal IEEE Transactions on Software Engineering*, 1988; **14(10)**: 1462-1477.

[5] Swanson E.B., The dimensions of maintenance, *In Proceedings of the 2nd International Conference on Software Engineering, IEEE Computer Society Press: Los Alamitos CA*, 1976; 492-497.

[6] Barton J., Czeck E., Segall Z. and Siewiorek D., Fault injection experiments using FIAT, *IEEE Transactions on Computers*, 1990; **39(4)**: 575-582.

[7] Jenn E., Arlat J., Rimen M., Ohlsson J. and Karlsson J., Fault injection into VHDL models: The MEFISTO tool, *In Proceedings of the 24th IEEE International Symposium on Fault Tolerant Computing*, 1994; 66-75.

[8] Leung H.K.N. and White L.J., Insights into testing and regression testing global variables, *Software Maintenance*, 1990; **2**: 209-222.

[9] Blue D., Segall I., Tzoref-Brill R., Zlotnick A., Interaction-based test suite minimization, *ICSE*, 2013; 182-191.

[10] Agrawal H., Horgan J., Krauser E. and London S., Incremental regression testing, *In Proc. of the Conf. on Softw. Maint*, Sept 1993; 348-357.

[11] Zelkowitz M.V., Wallace D.R. and Binkley D.W., Experimental Validation of New Software Technology, in Jurisbo N. and Moreno A.M., eds., *Lecture Notes on Empirical Software Eng.,* World Scientific, 2003; 229-263.

[12] Agrawal H., Horgan J., Krauser E. and London S., Incremental regression testing, *Proceeding of Conference on Software Maintenance* Sept. 1993; 348-357.

[13] Taha A.B., Thebaut S.M. and Liu S.S., An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis, *Procceeding of the 13th Annual International Computer Software and Applications Conf*erence, Sept. 1989; 527-534.

[14] Basili V.R. and Selby R.W., Comparing the effectiveness of software testing strategies, *IEEE Trans. Software Eng.*, 1987; **13(2)**: 1278-1296.

[15] Wong E. and Mathur A.P., Fault detection effectiveness of mutation and data-flow testing, *Software Quality J*, 1995; **4(1)**: 69-83.

[16] Rothermel G. and Harrold M., A safe efficient regression test selection technique, *ACM Trans. Softw. Eng. Methodol.*, 1997; **6(2)**: 173-210.

[17] Rothermel G. and Harrold M., Empirical studies of a safe regression test selection technique, *IEEE Trans. Softw. Eng.*, 1998; **24(6)**: 401-419.

[18] Rothermel G. and Harrold M., Analyzing regression test selection techniques, *IEEE Trans. Softw. Eng.*,1996; **22(8)**: 529-551.

[19] Rothermel G. and Harrold M., A safe efficient regression test selection technique, *ACM Trans. Softw. Eng. Methodol.*, 1997; **6(2)**: 173-210.

[20] Rothermel G. and Harrold M., Empirical studies of a safe regression test selection technique, *IEEE Trans. Softw. Eng.*, 1998; **24(6)**: 401-419.

[21] Graves T., Harrold M.J., Kim J.M., Porter A. and Rothermel G., An empirical study of regression test selection techniques, *In Proceedings of the 20th International Conference on Software Engineering*, Apr. 1998.