Regression Testing Cost Reduction Suite

Mohamed Alaa El-Din
Arab Academy for Science, Technology and Maritime Transport (AASTMT)
Cairo, Egypt

Ismail Abd El-Hamid Taha
Arab Academy for Science, Technology and Maritime Transport (AASTMT)
Cairo, Egypt

Hesham El-Deeb
Modern University for Technology and Information (M.T.I) Cairo, Egypt

***Abstract*—The estimated cost of software maintenance exceeds 70 percent of total software costs [1], and large portion of this maintenance expenses is devoted to regression testing. Regression testing is an expensive and frequently executed maintenance activity used to revalidate the modified software. Any reduction in the cost of regression testing would help to reduce the software maintenance cost. Test suites once developed are reused and updated frequently as the software evolves. As a result, some test cases in the test suite may become redundant when the software is modified over time since the requirements covered by them are also covered by other test cases.**

Due to the resource and time constraints for re-executing large test suites, it is important to develop techniques to minimize available test suites by removing redundant test cases. In general, the test suite minimization problem is NP complete. This paper focuses on proposing an effective approach for reducing the cost of regression testing process. The proposed approach is applied on real-time case study. It was found that the reduction in cost of regression testing for each regression testing cycle is ranging highly improved in the case of programs containing high number of selected statements which in turn maximize the benefits of using it in regression testing of complex software systems. The reduction in the regression test suite size will reduce the effort and time required by the testing teams to execute the regression test suite. Since regression testing is done more frequently in software maintenance phase, the overall software maintenance cost can be reduced considerably by applying the proposed approach.

Keywords*—*Software maintenance cost; reduced test suite; reduced regression test suite; regression testing cost reduction.

I. INTRODUCTION

In regression testing as integration testing proceeds, number of regression tests increases and it is impractical and inefficient to re-execute every test for every program if one change occurs.

Test suite reduction techniques decrease the cost of software testing by removing the redundant test cases from the test suite while still producing a reduced set of tests that covers the same level of code coverage as the original suite.

Optimizing the cost of the regression testing without compromising the fault exposing capability is always challenging for the testing team. Testing team always face constraints like lack of resources, squeezed testing schedule, changing and ambiguous requirement, which in terms impacts and reduces the effectiveness of regression testing. The Test automation tool will help testing team speed-up the test execution.

Due to the differences in the execution costs between the test cases, the representative set with the smallest number of tests may not be the one with the minimum execution cost. As such, the cost of a test should be a more important consideration for achieving cost-effective testing than the size of the test suite. Thus, it is necessary to consider individual execution costs when choosing the test cases.

The traditional HGS algorithm is one of the most common algorithms aiming to reduce the cost of regression testing. It is proposed by Harrold, Gupta and Soffa to test suite reduction "Selecting a representative set of test cases from a test suite, providing the same coverage as the entire test suite" that has received considerable attention. This algorithm assumes that we could have

Ti (for i = 1, 2, 3, .., m) represent the subsets of T, with each subset Ti containing all of the test cases that satisfy the i-th test requirement. The HGS algorithm could determine the representative test cases for each subset and include them in the representative set. The HGS algorithm follows the following four steps:

1) Initially, all requirements are unmarked.

2) for each requirement that is exercised by only one test case each, add each of these test cases to the minimized suite and mark it.

3) Consider the unmarked requirements in increasing order of the cardinality of the set of test cases exercising a requirement. If several requirements are tied since the sets of test cases exercising them have the same cardinality, select the test case that would mark the highest number of unmarked requirements tied for this cardinality. If multiple such test cases are tied, break the tie in favor of the test case that would mark the highest number of requirements with testing sets of successively higher cardinalities; if the highest cardinality is reached and some test cases are still tied, arbitrarily select a test case among those tied. Mark the requirements exercised by the selected test. Remove test cases that become redundant as they no longer cover any of the unmarked requirements.

4) Repeat the above steps until all testing requirements are marked.

The traditional HGS algorithm suffers from some disadvantages since no clear reason is shown for the initial

choice of the test cases as starting point. Also, it did not assure the cover all tests with all possible cases of all the selection statements.

II. Problem Statement

Given a set T of test cases {t1, t2, t3, ..., tn}, a set of testing requirements {r1, r2,· · ,rm} that must be covered to provide the desired coverage of the program, and the information about the testing requirements exercised by each test case in T, the test suite minimization problem focus on finding a minimal cardinality subset of T that exercises the same set of requirements as those exercised by the un-minimized test suite T.

Most of the existing approaches to reduction aim to decrease the size of the test suite disregarding the time/cost. Yet, the difference in the execution time/cost of the tests is often significant and it may be costly to use a test suite consisting of a few long-running test cases. [2]

The reduction in the original test suite could be computed according to the following formula:

$$C\ red\ [\%] = ((CR - C\ min)/\ CR) * 100 \qquad (1)$$

Where:
CR Original regression test suite
C min Reduced regression test suite

III. Alternative Approaches

Many techniques have been proposed to obtain the near-optimal solution for the test suite reduction problem. Even though the representative sets produced by these techniques are not guaranteed to be optimal, they can significantly decrease both the size of the test suite and the cost associated with its execution.

These approaches could include the usage of Greedy algorithm, selective redundancy approach and irreplaceability algorithm.

A. Greedy Algorithm

The Greedy algorithm is a commonly-used method for finding the near-optimal solution to the test suite reduction problem. This algorithm repeatedly removes the test which covers the most unsatisfied test requirements from the test suite set T to the requirements set until all of the requirements are covered. Many existing test suite reduction methods are based on the concept of the Greedy algorithm. In other words, many algorithms repetitively choose the "best" test case to obtain the near-optimal solution from the locally optimal solutions. [3]

B. Test Suite Reduction with Selective Redundancy

Test suite reduction that attempts to selectively keep redundant tests in the reduced suites. Experiments show that this approach can significantly improve the fault detection effectiveness of reduced suites without severely affecting the extent of test suite size reduction. This assures the achievement of high suite size reduction while simultaneously allowing for low fault detection effectiveness loss. The intuition driving is that when a non-reduced suite contains lots of redundancy with respect to a coverage criterion, it may be helpful to selectively keep some of that redundancy in the reduced test suite so as to retain more fault detection effectiveness in the reduced suite, hopefully without significantly affecting the amount of suite size reduction. [4]

C. Irreplaceability Algorithm

This algorithm is based on the concept of test irreplaceability which creates a reduced test suite with a decreased execution cost. Leveraging widely used benchmark programs, the empirical study shows that, in comparison to existing techniques, the presented algorithm is the most effective at reducing the cost of running a test suite. [5]

IV. Related Work

Researchers, practitioners and academicians proposed various techniques on test suite reduction, test case prioritization, and regression test selection for improving the cost effectiveness of the regression testing.

Rothermel and Harrold presented a technique for regression test selection. Their algorithms construct control flow graphs for a procedure or program and its modified version and use these graphs to select tests that execute changed code from the original test suite [6].

James A. Jones and Mary Jean Harrold proposed new algorithms for test suite reduction and prioritization [5]. Saifur-Rehman Khan, Aamer Nadeem proposed a novel test case reduction technique called Test Filter that uses the statement-coverage criterion for reduction of test cases [8]. T. Y. Chen and M. F. Lau presented dividing strategies for the optimization of a test suite [4]. M. J. Harrold etal presented a technique to select a representative set of test cases from a test suite that provides the same coverage as the entire test suite [8]. This selection is performed by identifying, and then eliminating, the redundant and obsolete test cases in the test suite. This technique is illustrated using data flow testing methodology.

A recent study by Wong, Horgan, London, and Mathur [3], examines the costs and benefits of test suite minimization. Rothermel et al [2] described several techniques for using test execution information to prioritize test cases for regression testing, including: techniques that order test cases based on their total coverage of code components, techniques that order test cases based on their coverage of code components not previously covered, and techniques that order test cases based on their estimated ability to reveal faults in the code components that they cover. Most of the techniques described in the above papers assume that source code of the software is available to the testing engineer at the time of testing. But in most of the organizations the testing is done in black box environment and the source code of the software is not available to the testing engineers. A simple greedy algorithm for the set-cover problem (and therefore for the test suite minimization problem) is described in [4]. The work presented in [9] uses a greedy technique for suite reduction in the context of model-based testing. This work showed that while suite sizes could be greatly reduced, the fault detection capability of the reduced suites was adversely affected. This

situation increases the degree of complexity of the proposal solutions for the test suite minimization problem.

Existing test suite minimization techniques are defined in terms of test case cover-age as they attempt to minimize the size of a suite while keeping some coverage requirement constant. A related topic is that of test case prioritization.

In contrast to test suite minimization techniques which attempt to remove test cases from the suite, the test case prioritization techniques [8, 10, and 11] only re-order the execution of test cases within a suite with the goal of early detection of faults. In [11], the ATACMIN tool [6] was used to find optimal solutions for minimizations of all test suites examined. This work showed that reducing the size of test suites while keeping all uses coverage constant could result in little to no loss in fault detection effectiveness. In contrast, the empirical study conducted in [12] suggests that reducing test suites can severely compromise the fault detection capabilities of the suites.

A new model for test suite minimization [7] has been developed that explicitly considers two objectives: minimizing a test suite with respect to a particular level of coverage, while simultaneously trying to maximize error detection rates with respect to one particular fault. A limitation of this model is that fault detection information is considered with respect to a single fault (rather than a collection of faults), and therefore there may be a limited confidence that the reduced suite will be useful in detecting a variety of other faults.

From the previous demonstration of the above related work, it could be concluded that suite size and fault detection effectiveness are opposing forces in the sense that more suite size reduction would intuitively imply more fault detection and effectiveness loss, since throwing away more test cases, in effect, throws away more opportunities for detecting faults. Thus, there seems to be an inherent tradeoff involved in test suite reduction: one may choose to sacrifice some suite size reduction in order to increase the chances of retaining more fault detection effectiveness.

V. ENHANCED HGS ALGORITHM (EHGSA)

The research approach target is to get the original regression testing and the reduced regression test suite reduction with selective redundancy by modifying the HGS algorithm. This approach is general and can be applied to any test suite minimization technique. EHGSA finds the minimum regression test with minimum machine time of the test suite covering all possible paths primary variables values of the all selection branch cases (IF) statements of both cases True/False (T/F) of the program tested.

The EHGSA algorithm have several advantages since it take into consideration all the possible braches cases of selection statements included in the program being tested. Also, it computes the real machine time for each branch case and the total time for each test of the test suite. The pseudo code of the EHGSA algorithm is illustrated in Fig. 1.

```
Begin:
Stage I: Create the Test Suite Text File
Input: n;   // number of selection statements
m: = 2^n;  // m is all possible tests ti
Open Test Suite Text File;
i:=0;
While (i < m)
j:=0;
While (j < n)
convert i to binary number b;
//ti: set primary values pj to binary i vales b
set primary values pj to b bit j vale;
j:=j+1;
End While
write Text Test Suite Line i ti;
i:=i+1;
End While
Close Test Suite Text File;
//  Test Suite Text File created

Stage II:
Step 1:  Establish Test Link List Class
Test Class Node Structure {
Test_ id;
Array Test_Coverage_marked_Selection_Cases;
Counter_Marked_Selection_Cases h;
Test_Machine_Time TT;
Pointer next_Node;
Pointer previous_Node; }

Step2: Apply Test Suit on Selection Statements
Open  Test_Suite_Text File T as Input;
// Array Primary Values PV
Array PV[n];
i :=0;
While ( ! T.eof( ) )
Read ( T , ti;);
j := 0;
While ( j < n)
Set  PV[ j ] := ti(j,jj );
j:= j +1;
End While
// Apply ti on Selection Statements Cases
If ( PV[k] )
// if statements staff
// Coverage Cases
// Calculate total machine test time of ti;
End If
Add test_ Node;
i =i+1;
End While
```

```
// Regression Testing Reduction Proposal Algorithm Step4:
// Find Test ti Max Coverage with Min Machine Time
Coverage = {};
Uncoverage={ all possible Coverage};
Min_Subset_Tests = {};
// Read  T Test Link List Nodes ti;
i:=0;
Max_Coverage := 0;
// Looking for Test ti with Max Coverage & Min Machine Time
MT  := Max_no;
// At Head  Test_Link_List T
While ( ! T.eof() )
Read T.Node ti;
If ( h >= Max_Coverage and  TT <= MT)
Max_Coverage := h;
Test := ti;
End If
i = i + 1;
End While

Min_Subset_Tests = Min_Subset_Tests + Test;
Coverage :::= Coverage + Test.Coverage;
Uncoverage := Uncoverage – Coverage;

Step5:
// Find Test ti cover Max Uncoverage with Min Machine Time

While ( Uncoverage != Null)
i:=0;
Max_Coverage := 0;
// Looking for Test ti with Max Coverage & Min Machine Time
MT  := Max_No;
// At Head  Test_Link_List T
While ( ! T.eof() )
Read T.Node ti;
If (Test_Coverage_marked_Selection_Cases
<=
Uncoverage  Max_Coverage and  TT <=
MT)
MT = TT;
Test := ti;
End If
i = i + 1;
End While
Min_Subset_Tests = Min_Subset_Tests + Test;
Coverage :::= Coverage + Test.Coverage;
Uncoverage := Uncoverage – Coverage;
End While
// Proposal Algorithm Output
Write Min_Subset_Tests;
End
```

Fig.1. EHGSA Pseudo Code

VI. Implementation

Pointing out the test suites with minimum machine time where the test suite covers all possible paths of the selection statements by applying algorithm in the following sample case study (Fig. 2) with four if statements n=4, each test ti has n primary variable values, p[i], (i = 0, 1, 2, 3).

```
1: read text test suite file line ti
(p[0], p[1], p[2], p[3]);
B1: if (p[0] > 0)
B1T:  // Branch 1 True Statements
B1F: else

// Branch 1 False Statements
End If
B2: if ( p[1] > 0)
B2T: // Branch 2 True Statements
B2F: else
// Branch 2 False Statements
End If
B3: if ( p[2] > 0)
B3T:  // Branch 3 True Statements
B4: if ( p[3] > 0)
B4T:  // Branch 4 True Statements
B4F: else
// Branch 4 False Statements
End If
B3F: else
// Branch 3 False Statements
End If
```

Fig.2. Sample Case Study [4]

TABLE I. The Test Suite File for all possible primary variables values m x n. where: n: number of selection statements, m = 2n

Test	p[3]	p[2]	p[1]	p[0]
t0	0	0	0	0
t1	0	0	0	1
t2	0	0	1	0
t3	0	0	1	1
t4	0	1	0	0
t5	0	1	0	1
t6	0	1	1	0
t7	0	1	1	1
t8	1	0	0	0
t9	1	0	0	1
t10	1	0	1	0
t11	1	0	1	1
t12	1	1	0	0
t13	1	1	0	1
t14	1	1	1	0
t15	1	1	1	1

TABLE II. EHGSA ALGORITHM OUTPUT ALL POSSIBLE REGRESSION TESTING WITH MACHINE TIME.M X ((2 * N) + 1).

Test/ Case	B1T	B1F	B2T	B2F	B3T	B3F	B4T	B4F	Time
t0		X		X		X			0.03
t1		X		X		X			0.03
t2		X		X	X			X	0.043
t3		X		X	X		X		0.042
t4		X	X			X			0.029
t5		X	X			X			0.029
t6		X		X	X		X		0.042
t7		X	X		X		X		0.041
t8	X			X		X			0.029
t9	X			X		X			0.029
t10	X			X	X			X	0.042
t11	X			X	X		X		0.41
t12	X		X			X			0.028
t13	X		X			X			0.028
t14	X		X		X			X	0.041
t15	X		X		X		X		0.040

The EHGSA Algorithm Final Result for the Reduction Subset Tests is: t15, t0, & t14.

VII. TRADITIONAL HGS ALGORITHM RESULTS

Apply the HGS algorithm over the same selection statements case study Fig2. The HGS is used the test suite consists of only five test {t1, t2, t3, t4, t5} [4]. The HGs algorithm used the following test suite.

TABLE III. THE HGS TEST SUITE INITIAL SUITE

Test	p[0]	p[1]	p[2]	p[3]
t0	1	1	0	0
t1	0	0	1	0
t2	0	1	0	0
t3	0	1	1	1
t4	0	0	1	1

TABLE IV. HGS ALGORITHM OUTPUT REGRESSION TESTS

Test/ Case	Bt1	Bf1	Bt2	Bf2	Bt3	Bf3	Bt4	Bf4
T1	X		X			X		
T2		X		X	X			X
T3		X	X			X		
T4		X	X		X		X	
T5		X		X	X		X	

The HGS Algorithm Final Result for the Reduction Subset Tests is: t1, t2, & t4.

VIII. EXPERIMENTAL RESULTS

The EHGSA algorithm stage one has generates the text test suite file for all possible variables values PV.

The EHGSA algorithm stage two its input is the text test suite file then generate the original regression testing: CR.

The EHGSA algorithm stage two has criteria to find the Reduced Regression Test Suite CMIN of the original regression testing: CR that coverage all possible selection statement branch test cases with minimum cost (machine time) "Regression Testing Cost Reduction Suite".

Apply the EHGSA algorithm over different programs contains different number of selection statements SS has following parameters:

- Number of Selection Statements: SS
- Number of Primary Variables: PR = SS
- Possible Primary variables Values of for both branch cases T/F : PV = 2^{SS}
- Possible selection Branch Test Cases : BTC = 2 * SS
- Original Regression Testing: CR = 2^{SS}
- Reduced Regression Test Suite C_{MIN}

The reduction in the original test suite could be computed according to the formula (1)

TABLE V. EHGSA ALGORITHM EXPERIMENT RESULTS

SS	PR	PV	BTC	CR	C_{MIN}	C_{RED}[%]
4	4	16	8	16	3	81.25%
5	5	32	10	32	4	87.50%
6	6	64	12	64	5	92.19%
7	7	128	14	128	6	95.32%
8	8	256	16	256	7	97.27%
9	9	512	18	512	7	98.63%
10	10	1024	20	1024	8	99.22%
11	11	2048	22	2048	8	99.60%

The following figure illustrate the results in a bar chart which clarify that the reduction in cost of regression testing for each regression testing cycle is ranging highly improved in the case of programs containing high number of selected statements

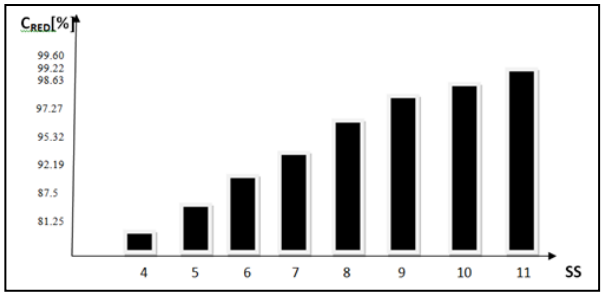

Fig.3. EHGSA Reduction Cost Results

IX. CONCLUSION

Selecting the reduced testing cases, in appropriate accurate approach; needs browsing all the possible paths of cases of the selection statements included in the cod.

The paper proposed algorithm automatically generates the test suite that cover all possible test primary variables values of all cases true/false for all selection statement of the tested program code. This algorithm computes the machine time of each test case on a dynamic base using the linked list with test node. The EHGSA finds the subset tests covering all possible test paths of all selection statements with minimum machine time which in turn reduced the regression testing cost.

REFERENCES

[1] Sriraman Tallam, Neelam Gupta," A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization", Proceeding PASTE '05 Proceeding of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software and engineering 2005.

[2] Prof. A. Ananda Rao and Kiran Kumar J "An Approach to Cost Effective Regression Testing in Black-Box Testing Environment", May 2011

[3] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: a Survey," Software Testing, Verification and Reliability, Vol. 22, No. 2, March 2012.

[4] Dennis Jeffreyand and Neelam Gupta., "Test Suite Reduction with Selective Redundancy", Dept. of Computer the University of Arizona Tucson, AZ 85721. IEEE Computer Society Washington, DC, USA ©2005

[5] Chu-Ti Lin, Kai-Wei Tang, Cheng-Ding Chen, Gregory M. Kapfhammer., "Reducing the Cost of Regression Testing by Identifying Irreplaceable Test Cases", Aug 28, 2012

[6] H. Zhong, L. Zhang, and H. Mei, "An Experimental Study of Four Typical Test Suite Reduction Techniques," Information and Software Technology, Vol. 50, No. 6, pp. 534-546, May 2008.

[7] Prashant Malangave and Dr. Dinesh B. Kulkarni, "Efficient Test Case Prioritization in Regression Testing", Walchand Collage of Engineering Dept. of Computer Science & Eng, 2008

[8] M. J. Harrold, R. Gupta, and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite," ACM Trans. on Software Engineering and Methodology, Vol. 2, No. 3, pp. 270-285, July 1993.

[9] A. M. Smith and G. M. Kapfhammer, "An Empirical Study of Incorporating Cost into Test Suite Reduction and Prioritization," Proceedings of the 24th ACM SIGAPP Symposium on Applied Computing, Software Engineering Track, March 2009.

[10] Luciano S. de Souza1; Ricardo B. C. Prudˆ encio2, Flavia de A. Barros," Multi-Objective Test Case Selection: A study of the influence of the Catfish effect on PSO based strategies". Anais do XV Workshop de Testes e Tolerância a Falhas - WTF 2014.

[11] Dennis Jeffrey, Neelam Gupta," Test Suite Reduction with Selective Redundancy" Proceeding of ICSM '05 Proceedings of the 21st IEEE International Conference on Software Maintenance, Pages 549-558, 2005.

[12] Anannado Rao and Kirzn Kumar J," An Approach to Cost Effective Regression Testing in Black-Box Testing Environment"IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 3, No. 1, May 2011.