International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:2, 2010

Regression Test Selection Technique for Multi-Programming Language

Walid S. Abd El-hamid, Sherif S. El-Etriby, and Mohiy M. Hadhoud

***Abstract*—**Regression testing is a maintenance activity applied to modified software to provide confidence that the changed parts are correct and that the unchanged parts have not been adversely affected by the modifications. Regression test selection techniques reduce the cost of regression testing, by selecting a subset of an existing test suite to use in retesting modified programs. This paper presents the first general regression-test-selection technique, which based on code and allows selecting test cases for any programs written in any programming language. Then it handles incomplete program. We also describe RTSDiff, a regression-test-selection system that implements the proposed technique. The results of the empirical studied that performed in four programming languages java, C#, C++ and Visual basic show that the efficiency and effective in reducing the size of test suit.

***Keywords*—**Regression testing, testing, test selection, software evolution, software maintenance.

I. Introduction

SOFTWARE maintenance typically involves code changes to satisfy customer requirements like fixing bugs, addition of a new functionality, improving existing functionalities, and so on. After incorporating the change, the impact is analyzed and the software is re-validated using regression testing.

Regression testing is performed on modified software to provide confidence that changes are correct and do not adversely affects other portions of the software. Regression testing is expensive; it can account for as much as one-half of the cost of software maintenance. The cost of selecting regression test cases to rerun must be lower than the cost of running the remaining test cases for test selection to make sense [14].

The difference between regression testing and development testing is that, during regression testing, an established test suite of tests may be available for reuse. The simplest regression testing technique, *retest-all*, reruns all test cases in test suite. But, this technique may take a long time and resources. An alternative approach, *regression test selective techniques*, in contrast, attempt to reduce the time required to retest a modified program by selectively reusing tests.

Most regression test selection techniques are *white-box* (*code based*), that is, they select tests based on information about the data between original code and the modified code [2], [7], [10], [12], [17], [19], [22]. Only a few techniques are *black-box* (*specification-based*) methods, that is, they select tests based on architecture and design information represented with the Unified Modeling Language (UML). But designs for impact analysis and test selection require the designs to be complete and up-to date [1], [3], [8], [9].

In this paper we address a general code based regression test selection technique. Our technique compares the source code of original program and modified program and determines the difference between them then selects the test cases that execute changed code from the original test suite. The technique has several advantages over other regression test selection techniques. Unlike many techniques, our algorithms detect the difference between original and modified version written in any programming language and select tests that formerly executed statements that have been deleted from the original program. The proposed technique is *safe*: where it select *every* test from the original test suite that can expose faults in the modified program. The main benefit of this approach is that, in many cases, a small subset of the test suite is selected, which reduces the time required to perform the testing.

Finally, our technique is more general than most other techniques. They handle all language constructs and all types of program modifications for procedural languages. We have implemented our algorithms and conducted empirical studies on several subject programs and modified versions. The results suggest that, in practice, the algorithms can significantly reduce the cost of regression testing a modified program.

The rest of the paper is organized as the following: Section II Provide background information about regression testing. In Section III describes our regression test selection algorithm. The case study results are presented in Section IV. In Section V we present the related work. Finally, conclude the paper in Section VI.

II. Problem Definition

Let P' be a modified version of P, and T be the test suite used to test P. During regression testing of P', T and information about the testing of P with T are available for use in testing P'.

W. S. Abd El-hamid is with the Faculty of Computers and Information, Menofyia University, Gamal Abdul-nasser Str. Shebin El-Kom, Menofyia, Egypt (Fax. +2 048 2223694; e-mail: walid_mufic@yahoo.com.)

Sherif. S. El-etriby is with the Faculty of Computers and Information, Menofyia University, Gamal Abdul-nasser Str. Shebin El-Kom, Menofyia, Egypt (e-mail: sherif.ali@ci.menofia.edu.eg).

M. M. Hadhoud is with the Faculty of Computers and Information, Menofyia University, Gamal Abdul-nasser Str. Shebin El-Kom, Menofyia, Egypt (e-mail: mnhadhoud@yahoo.com).

Program *P*	**Modified Program *P′***
Public class A{	Public class A{
Void f1( ){….}	Void f1( ){….}
Void f2( ){….}	Void f3( ){….}
Void f3( ){….}	
}	}
Public class B extends A{	Public class B extends A{
	Void f1( ){….}
Void f4 (){….}	Void f4( ){….}
}	}
Public class X{….}	
Public class D{	Public class D{
Void f5(A a)	Void f5(A a)
{	{
a.f1( );	a.f1( );
}	}
}	}
	Public class Y{….}

Fig. 1 Original program P and its modified P'

A number of safe regression-test-selection techniques that vary in precision and efficiency have been presented (e.g., [2], [7], [10], [17], [19]). We can view these techniques as a family of regression-test-selection techniques that use information about the program's source code to select *T′*. Fig. 1 illustrates a general regression-test-selection system.

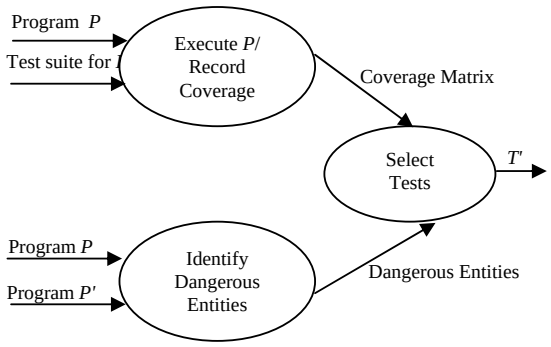

Fig. 2 A general regression test selection technique

In this system, a program *P* is executed with a test suite *T*. In addition to the results of the execution the pass/fail information the system records coverage information about which entities in *P* are executed by each test case *t*. The types of entities recorded depend on the specific regression-test-selection technique. After all test cases have been run, the coverage information is compiled into a coverage matrix that associates each *t* in *T* with the entities that it executes. In addition to computing coverage information, these techniques compare *P* and *P′*, and identify in *P* a set of dangerous entities. After dangerous edges have been identified, the select tests component of the regression-test-selection system uses the dangerous entities and the coverage matrix to select the test cases in *T* to add to *T′*.

Rothermel and Harrold outline the following approach to regression testing to solve this problem (*T* is the original test suite):

1- Identify changes made to *P* by creating a mapping of the changes between *P* and *P′*.
2- Use the results of step 1 to select a set *T′* subset of *T* that may reveal change-related faults in *P′*.
3- Use *T'* to test *P′*.
4- Identify if any parts of the system have not been tested adequately and generate a new set of tests *T″*.
5- Use *T″* to test software

III. Our Regression Test Selection

Our algorithm, TestSelection, given in Fig. 3, takes as input an original version of a program (*P*), a modified version of that program (*P′*) and the test suite *T* for *P*. The algorithm outputs a set that contains tests that are modification-traversing for *P* and *P′*.

TestSelection performs its comparison first at the class, then at the method level and finally at the node level. The algorithm first compares each class in *P* with the like named class in *P′*, and produces sets of class pairs (*C*). For each pair of classes, TestSelection then matches methods in the class in *P* with methods having the same signature in the class in *P′*; the result is a set of method pairs (*M*). Then the algorithm constructs Enhanced CFGs (ECFGs) for the two methods and match nodes in the two ECFGs. Finally TestSelection call procedure compare, passing *E* and *E′* as parameters and return *T′*, the set of test cases selected.

Algorithm TestSelection
Input: *P* : original program
P′: modified program
T : test set used to test *P*
Output: *T* ′: the subset of *T* selected for use in regression testing *P′*
Begin:
1: *T* ′=Ø
2: compare classes in *P* and *P* ; add matched class pairs to *C*
3: **for** each pair *(c, c′)* in *C* **do**
4: compare methods; add matched method pairs to *M*
5: **for** each pair *(m ,m′)* in *M* **do**
6: create ECFGs *G* and *G′* for methods *m* and *m′* with entry nodes *E* and *E′*
7: *T* ′ = *T* ′ ∪ Compare(*E*,*E′*)
8: **end for**
9: **end for**
10: **return** *T* ′
11: **end** TestSelection

Fig. 3 TestSelection Algorithm

A. Class Level

TestSelection begins its comparison at the class level (line 2). The algorithm matches classes that have the same fully-qualified name; the fully-qualified name consists of the package name followed by the class name. Matching classes in *P* and *P′* are added to *C*. Classes in *P* that do not appear in set *C* are deleted classes, whereas classes in *P′* that do not appear in set *C* are added classes. In the example programs in

Fig. 1, we consider class X in original program *P* as deleted class and class Y in modified program *P'* as added class.

B. Method Level

After matching classes, TestSelection compares, for each pair of matched classes, their methods (lines 3–4). The algorithm first matches each method in a class with the method with the same signature in another class. Then, if there are unmatched methods, the algorithm looks for a match based only on the name. This matching accounts for cases in which parameters are added to (or removed from) an existing method which we found to occur in practice and increases the number of matches at the node level. Pairs of matching methods are added to *M*. Like the approach used for classes, methods in *P* that do not appear in set *M* are deleted methods, whereas methods in *P'* that do not appear in set *M* are added methods. In the example programs in Fig. 1, we consider method f2( ) in class A in original program *P* as deleted method and consider method f1( ) in class B in modified program *P'* as added method.

C. Node Level

TestSelection uses the set of matched method pairs (*M*) to perform matching at the node level. First, the algorithm considers each pair of matched methods (*m*,*m'*) in *M*, and builds ECFGs *G* and *G'* for *m* and *m'* (lines 5–6).

When comparing two methods *m* and *m'*, the goal of our algorithm is to find, for each statement in *m*, a matching statement in *m'*, based on the method structure. Thus, the algorithm requires a modeling of the two methods that (1) explicitly represents their structure, and (2) contains sufficient information to identify differences and similarities between them. Although CFGs can be used to represent the control structure of methods, traditional CFGs do not suitably model many object oriented constructs. To suitably represent object-oriented constructs, and model their behavior, we define the ECFG. ECFGs extend traditional CFGs and are tailored to represent object-oriented programs as shown in Fig. 4.

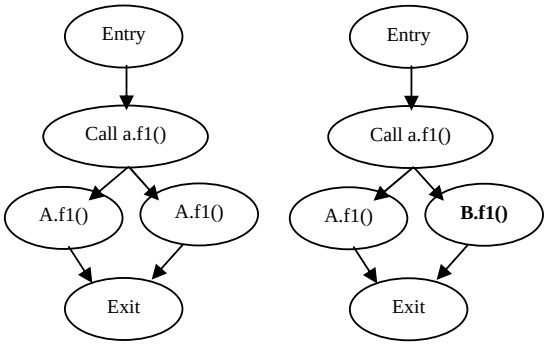

Fig. 4 ECFGs for D.f5 in P and P' (Fig. 1)

Finally, TestSelection calls procedure Compare given in Fig. 5, passing *E* and *E'* as parameters. Compare identifies differences between nodes in *G* and *G'* (line 7), and creates and returns *T '*, the set of test cases selected for the change between nodes.

```
procedure Compare( N, N' )
input  :  N and N ' : nodes in G and G'
output :  T' : set of test cases for changed node
begin
1. mark N  " N '-visited "
2. for each successor C of N  in G do
3.    L =  the label on edge (N, C )
4.    C ' = the node in G' such that ( N ', C ') has label L
5.    if C is not marked  "C '- visited"
6.       if  not LEquivalent (C, C ')
7.          return TestsOnEdge ((N, C))
8.       else
9.           Compare(C, C ')
10.      endif
11.   endif
12. endfor
13. end Compare
```

Fig. 5 Compare algorithm

The function Compare is called with pairs of nodes *N* and *N'*, from *G* and *G'*, respectively, that are reached simultaneously during the algorithm's comparisons of traversal trace prefixes. Given two such nodes *N* and *N'*, Compare determines whether *N* and *N'* have successors whose labels differ along pairs of identically labeled edges. If *N* and *N'* have successors whose labels differ along some pair of identically labeled edges, test that traverse the edges are modification-traversing due to changes in the code associated with those successors. In this case Compare selects those tests. If *N* and *N'* have successors whose labels are the same along a pair of identically labeled edges, Compare continues along the edges in *G* and *G'* by invoking itself on those successors.

Fig. 5 describes Compare's actions more precisely. When Compare is called with ECFG nodes *N* and *N'*, Compare first marks node *N* " *N'*-visited" (line 1). After Compare has been called once with *N* and *N'* it does not need to consider them again, this marking step lets Compare avoid revisiting pairs of nodes. Next, in the for loop of lines 2-12, Compare considers each control flow successor of *N*. For each successor *C*, Compare locates the label *L* on the edge from *N* to *C*, then seeks the node *C* ' in *G'* such that (*N'*, *C* ') has label *L*; Next, Compare considers *C* and *C* '. If *C* is marked "*C* ' -visited," Compare has already been called with *C* and *C* ', so Compare does not take any action with *C* and *C* '. If *C* is not marked "*C* '-visited," Compare calls LEquivalent with *C* and *C* '. The LEquivalent function takes a pair of nodes *N* and *N '* and determines whether the statements *S* and *S'* associated with *N* and *N '* are lexicographically equivalent. If LEquivalent(*C*,*C* ') is false, then tests that traverse edge (*N*, *C*) are modification-traversing for *P* and *P*' ; Compare uses TestsOnEdge to identify these tests and adds them to *T*'. If LEquivalent (*C*,*C* ') is true, Compare invokes itself on *C* and *C* 'to continue the graph traversals beyond these nodes.

IV. Case Study Results

To evaluate our approach for regression test selection, we used RTSDiff to perform two empirical studies. Our study utilized two software subjects: Calculator and Sorting. Each software subject consists of an original version *P*, several

modified versions *(V1,…,Vn)* and a set of test cases that used to test *P*. These softwares are written in four programming languages: Java, C#, C++ and VB.Net.

The first subject for our studies is the implementation of the Calculator software that implemented in Java, C#, C++ and VB.Net programming languages. We obtained three versions, along with a test suite that had been used to test the software.

Fig. 6 shows the results for the four programming languages for Calculator software, the figure shows the percentage of test cases that were selected for each version of this software.

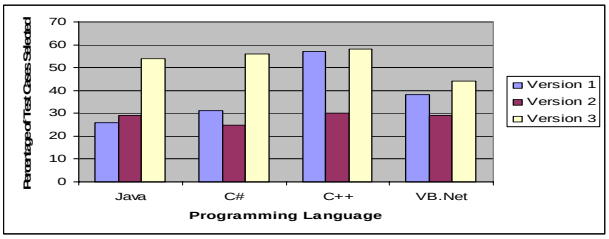

Fig. 6 Percentage of test cases selected for Calculator Software

We have analyzed the versions of the software and results are tabulated in Table I.

TABLE I
RESULTS OF CALCULATOR SOFTWARE CASE STUDY

Language	% Test cases recommended			% of average test cases recommended	% of test effort saved
	V1	V2	V3		
Java	26	29	54	36	64
C#	31	25	56	37	63
C++	57	30	58	48	52
VB.Net	38	29	44	37	64

The second subject for our studies is the implementation of the Sorting software that implemented in Java, C#, C++ and VB.Net programming languages. We obtained four versions, along with a test suite that had been used to test the software.

Fig. 7 shows the results for the four programming languages for Sorting software, the figure shows the percentage of test cases that were selected for each version of this software.

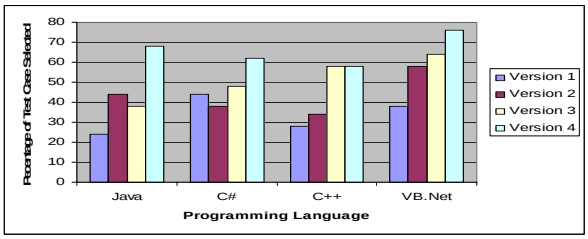

Fig. 7 Percentage of test cases selected for Sorting Software

We have analyzed the versions of the software and results are tabulated in Table II.

TABLE II
RESULTS OF SORTING SOFTWARE CASE STUDY

Language	% Test cases recommended				% of average test cases recommended	% of test effort saved
	V1	V2	V3	V4		
Java	24	44	38	68	43	57
C#	44	38	48	62	48	52
C++	28	34	58	58	44	56
VB.Net	38	58	64	76	59	41

V. RELATED RESEARCH WORK

Typically regression test selection techniques are either code-based or model-based. Code-based techniques [2], [7], [10], [12], [13], [17], [19] use the information obtained from two different versions of the code to analyze the change impact and select the tests.

Chianti [6] and JDiff [4] are comprehensive techniques for managing changes in Java programs. Chianti selects regression tests after analyzing the change impact analysis whereas JDiff performs only change impact analysis. As both these tools analyses the changes at statement level and are specific to Java programming.

In the case of model based techniques [1], [3], [8], [9], [14], [15] change information is obtained through two versions of models constructed during the requirements analysis phase or system design phase. But this techniques are used only when design are available in UML.

Reference [14] present an approach to identifying change impact analysis using UML sequence, use case and class diagrams. Their approach is somewhat different as their major focus is on the code based test cases, so mapping is between change identification at design level and its impact on code based test cases, which implies that the tester would have to wait for the code to develop and then test it using code-based test cases.

Reference [15] use UML activity diagrams to detect changes in design and then use a traceability matrix between activity diagram and the test suite. It covers activities at an abstract level and does not cover the attributes of a class. Also, it does not support object-oriented features.

Reference [8] propose a regression testing technique based on UML sequence and class diagrams. Their approach does not take into account the pre and post conditions of the operations which affect behavior of a class.

Our technique is based on code model to allow regression test selection for all programs written in any programming language.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented the first general regression test selection technique that based on code generation. Our technique handles most of the programming language where we applied our technique on java, C#, C++ and Visual basic software. We also present a tool called RTSDiff that implement our technique. With RTSDiff we performed

empirical studies to evaluate the effectiveness of our technique. Our empirical studies indicate that the technique can be effective in reducing the size of the test suite for software written in any programming language, but this work detect only the static change in the software.

Our future improving the efficiency of the tool to can detect the dynamic change in object oriented, gathering additional subjects, performing empirical studies to evaluate the effectiveness of our technique and applying this technique on large software.

REFERENCES

[1] L.C. Briand, Y. Labiche and S. He, "Automating regression test selection based on UML designs", 2008, pp 16-30.
[2] Anjaneyulu Pasala, Yann G. Rothermel, M.J. Harrold, J. Debhia, Regression test selection for C++ software, Journal of Software Testing, Verification, and Reliability, 2000 pp 77–109.
[3] ick LH, Fady A, Appala Raju G and Ravi P Gorthi, “Selection of regression test suite to validate software applications upon deployment of upgrades”, 19th Australian Software Engineering conference, 25-28 March 2008, pp 130-138.
[4] Ravi P Gorthi, Anjaneyulu Pasala, Kailash KP and Benny Leong, "Specification-based approach to select regression test suite to validate change software", 15th Asia-Pacific Software Engineering conference, 2008, pp 153-160.
[5] Apiwattanapong, T., Orso, A., and Harrold, M.J., “JDiff: A Differencing Technique and Tool for Object--Oriented Programs”, Journal of Automated Software Engineering, Vol 14, No. 1, March 2007, pp 3-36.
[6] Anjaneyulu P, Yannick LH Lew, and Ravi Prakash G, “How to Select Regression Tests to Validate Applications upon Deployment of Upgrades”, Vol. 6, No. 1, 2008, pp 55 – 62.
[7] Xiaoxia R, Barbara G R, Maximilian S and Frank T, “Chianti: A prototype change impact analysis tool for Java”, Proceedings of 27th International Conference on Software Engineering (ICSE), St. Louis, USA, May 15-21, 2005, pp 664-665.
[8] T. Koju, S. Takada, N. Doi, "Regression test selection based on intermediate code for virtual machines", Proceeding of International Conference on Software Maintenance (ICSM 03), 2003 , pp 1-10.
[9] Orest P, Hunay U, and Andrews A, "Regression Testing UML Designs", Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM), Philadelphia, Pennsylvania, September 24-27, 2006, pp254-264.
[10] A. Ali, A. Nadeem, M.Z. Iqbal, M. Usman, "Regression testing based on UML design models", 13th IEEE International Symposium on Pacific Rim Dependable Computing, 2007, pp 85-88.
[11] A. Orso, N. Shi and M.J. Harrold, "Scaling regression testing to large software systems", Proceeding of the 12th ACMSIGSOFT International Symposium on Foundation of Software Engineering, 2004, pp 241-251.
[12] G. Rothermel, M.J. Harrold, "Analysing regression test selection techniques", IEEE Transactions on Software Engineering , 1996, pp 529–551.
[13] M. Skoglund and P. Runeson, "A Case Study of The Class Firewall Regression Test Selection Technique on a Large Scale Distributed Software System" IEEE, 2005, pp74-83.
[14] E. Martins and V.G. Vieira, "Regression test selection for testable classes", ENCS 2005, pp 453-470.
[15] L.C. Briand, Y. Labiche, G. Soccar, "Automating impact analysis and regression test selection based on UML designs", International Conference on software Maintenance, 2002, pp 252-261.
[16] Y. Chen, R.L. Probert, D.P. Sims, Specification based Regression test selection with risk analysis, in: Proceedings of Conference of the Center for Advance Studies on Collaborative Research, 2002.
[17] H. Agrawal, J.R. Horgan, E.W. Krauser and S.A. London, "Incremental Regression Testing", Proceedings of IEEE Conference on software Maintenance,1993, pp 348-357.
[18] M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, Regression test selection for java software, in: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’01), 2001, pp 312-326.
[19] T. Apiwattanapong, A. Orso, M.J. Harrold, "A differencing algorithm for object-oriented programs", Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), 2004, pp 2–13.
[20] G. Rothermel, M.J. Harrold, "A. Safe, Efficient regression test selection technique", ACM Transactions on Software Engineering and Methodology, 1997, pp 173–210.
[21] Baradhi G and Mansour N, "A Comparative study of Five Regression Testing Algorithms", Proceedings of Australian Software Engineering Conference (ASWEC), Sydney, Australia, 1997, pp 174-183.
[22] Graves T L, Harrold M J, Kim J, Porter A and Rothermel M, "An empirical Study of Regression Test Selection Techniques", ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2, April 2001, pp 184-208.