Regression Test Selection for Trusted Database Applications

Ramzi A. Haraty and Wissam Chehab
Division of Computer Science and Mathematics, Lebanese American University, Lebanon

Abstract: *Regression testing is any type of software testing, which seeks to uncover regression bugs. Regression bugs occur as a consequence of program changes. Regression testing must be conducted to confirm that recent program changes have not harmfully affected existing features and new tests must be created to test new features. Testers might rerun all test cases generated at earlier stages to ensure that the program behaves as expected. However, as a program evolves the regression test set grows larger, old tests are rarely discarded, and the expense of regression testing grows. Repeating all previous test cases in regression testing after each major or minor software revision or patch is often impossible due to time pressure and budget constraints. This paper presents algorithms for regression testing for trusted database applications. Our proposed algorithms automate an important portion of the regression testing process, and they operate more efficiently than most other regression test selection algorithms. The algorithms are more general than most other techniques. They handle regression test selection for single procedures and for groups of interacting procedures. They also handle all language constructs and all types of program modifications for procedural languages.*

Keywords: *Regression testing, trusted applications.*

Received April 8, 2005; accepted July 26, 2005

1. Introduction

The purpose of regression testing is to isolate and perform only re-testable-type tests. This requires the ability to recognize reusable tests and obsolete tests. The isolation process is known as Regression Test Selection (RTS). Analyses for RTS attempt to determine if a modified program, when run on a specific test, will have the same behavior as before, without actually running the new program. The RTS analysis confronts a price/performance tradeoff. A more precise analysis might be able to eliminate more tests, but could take much longer to run.

Most research literature addresses one or both of two problems [9]: How to select regression tests from an existing test suite (the RTS problem)? And how to determine the portions of a modified program that should be re-tested (the coverage identification problem)?

There are three main philosophies to RTS in the literature [11]:

1. *Minimization*: Approaches seek to satisfy structural coverage criteria by identifying a minimal set of tests that must be rerun to cover changed code.
2. *Coverage*: Approaches are also based on coverage criteria, but do not require minimization. Instead, they seek to select all tests that exercise changed or affected program components.
3. *Safe*: Methods attempt instead to select every test that will cause the modified program to produce different output than original program.

Rothermel and Harrold [12] proposed the following criteria for regression testing:

1. *Inclusiveness*: It measures the extent to which a method chooses tests that will cause the modified program to produce a different output.
2. *Precision*: How well the RTS avoids tests that will not cause the modified program to produce different output than the original program.
3. *Efficiency*: It measures the computational cost and automatability, and thus practicality, of a selective retest approach.
4. *Generality*: It measures the ability of a method to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications.

Estimates indicate that software maintenance activities account for as much as two-thirds of the cost of software production. One necessary but expensive maintenance task is regression testing, performed on a modified program to introduce confidence that changes that have been made are correct, and have not adversely affected unchanged portions of the program. An important difference between regression testing and development testing is that during regression testing an established set of tests is available for reuse. One approach to reusing tests, the retest all approach, chooses all such tests, but this strategy may consume excessive time and resources.

Although many techniques for selective retest have been developed [1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 14, 16], there is no established basis for evaluation and comparison of these techniques. Classifying selective retest strategies for evaluation and comparison is difficult because distinct philosophies lie behind the existing approaches. Minimization approaches [4, 7, 13] assume that the goal of regression testing is to reestablish satisfaction of some structural coverage criterion, and aim to identify a minimal set of tests that must be rerun to meet that criterion. Coverage approaches [2, 3, 6, 9, 10, 14, 16], like minimization approaches, rely on coverage criteria, but do not require minimization. Instead, they assume that a second but equally important goal of regression testing is to rerun tests that could produce different output, and they use coverage criteria as a guide in selecting such tests.

Safe approaches [1, 8, 12] place less emphasis on coverage criteria, and aim instead to select every test that will cause the modified program to produce different output than the original program.

The remainder of the paper is organized as follows. Section 2 presents background work. Section 3 presents our algorithms. Section 4 discusses the empirical results. Finally, section 5 concludes the paper.

2. Background

Most work on regression testing addresses the following problem: Given program P, its modified version P', and test set T used previously to test P, find a way, making use of T, to gain sufficient confidence in the correctness of P'. Solutions to the problem typically consist of the following steps:

1. Identify the modifications that were made to P. Some approaches assume the availability of a list of modifications, perhaps created by a cooperating editor that tracks the changes applied to P [11]. Other approaches assume that a mapping of code segments in P to their corresponding segments in P' can be obtained using algorithms that perform slicing [15].
2. Select T' included in T, the set of tests to re-execute on P'. This step may make use of the results of step 1, coupled with test history information that records the input, output, and execution history for each test. An execution history for a given test lists the statements or code segments exercised by that test. For example, Table 1 shows test history information for procedure AVG.
3. Retest P' with T', establishing P' correctness with respect to T'. Since we are concerned with testing the correctness of the modified code in P', we retest P' with each Ti Є T'. As tests in T' are rerun, new test history information may be gathered for them.
4. If necessary, create new tests for P'. When T' does not achieve the required coverage of P', new tests are needed. These may include functional tests required by specification changes, and/or structural tests required by coverage criteria.
5. Create T", a new test set history for P'. The new test set includes tests from steps 2 and 4, and old tests that were not selected, provided they remain valid. New test history information is gathered for tests whose histories have changed, if those histories have not yet been recorded. Figure 1 shows the solutions steps.

```
S1.   Count = 0
S2.   Fread (fileptr,n)
S3.   While (not EOF) do
S4.     If (n<0)
S5.       Return(error)
        Else
S6.       Numarray[count]
S7.       Count++
        Endif
S8.     Fread (fileptr,n)
      Endwhile
S9.   Avg = calcavg (numarray,count)
       Calcavg (numarray,count)
S10.  Return(avg)
```

Figure 1. Solutions steps of the problem.

Table 1. AVG and its test history information.

Test Number	Input	Output	Execution History
T1	empty file	0	S1,S2,S3,S9,S10
T2	-1	Error	Sl,S2,S3,S4,S5
T3	1 2 3	2	S1,S2,S3,S4,S6,S7, S8,S3,...,S9,S10

3. Our Algorithm for Secure Regression Testing

3.1. Observations

One critical necessary maintenance activity, security regression testing, is performed on modified secure interfaces to provide confidence that the software behaves correctly and modifications have not adversely impacted the system's security, in order that the trusted code remains trused.

An important difference between regression testing and development testing is that, during regression testing, an established suite of tests may be available for reuse. One absurd security regression testing strategy is to reruns all such tests, but this retestall approach may consume inordinate time and resources. On the other hand, selective security retest techniques, attempt to reduce the time required to retest a secure program by selectively reusing tests and selectively retesting the modified program. These techniques address two problems:

1. The problem of selecting tests from an existing test suite, and
2. The problem of determining where additional tests may be required.

Both of these problems are important. Our new strategy presents an enhanced regression test selection technique that is specifically tailored for trusted applications. The approach constructs control flow graphs for a secure procedure or program and its modified version and use these graphs to select tests that execute changed code from the original test suite.

The new strategy has several advantages over other regular regression test selection techniques. Unlike many techniques, our algorithms select tests that may now execute new or modified statements and tests that formerly executed statements that have been deleted from the original program.

We prove that under certain conditions the algorithms are safe, that is, they select every test from the original test suite that can expose faults in the modified program. Moreover, they are more precise than other safe algorithms because they select fewer such tests than those algorithms. Our algorithms automate an important portion of the regression testing process, and they operate more efficiently than most other regression test selection algorithms. Finally, our algorithms are more general than most other techniques. They handle regression test selection for single procedures and for groups of interacting procedures. They also 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 precisely and safely reduce in a significant way the cost of regression testing of a modified program.

3.2. Algrithm for Secure Regression Testing

Our algorithm SelectTests as shown in Figure 2, takes a procedure P, its changed version P', and the test history for P, and returns T', the subset of tests from T that could possibly expose errors if run on P. The algorithm constructs CDG's for P and P', and then calls procedure Compare with the entry nodes E and E' of the two CDG's. Compare is a recursive procedure. Given any two CDG nodes N and N', Compare method marks these nodes "visited", and then determines whether the children of these nodes are equivalent. If any two children are not equivalent, a difference between P and P' has been encountered. In this case, the only tests of P that may have traversed the change in P are those that traversed N in P. Thus, Compare returns all tests known to have traversed N. If, on the other hand, the children of N and N' are equivalent, Compare calls itself on all pairs of equivalent non--visited predicate or region nodes that are children of N and N', and returns the union of the tests (if any) required to test changes under these children.

```
Algorithm SelectTests
Input:     procedure P, changed version P',
           and test set T
Output:    test set T'
Begin
      Construct CDG and CDG', CDG's of P and P'
      Let E and E' be entry nodes of CDG and CDG'
      T' = Compare (E, E')
End

Procedure Compare
Input:     N and N': nodes in CDG and CDG'
Output:    test set T'
Begin
    Mark N and N' "visited"
    If the children of N and N' differ return
    (all tests attached to N) else
        T' = NULL
        For each region or predicate child node of N
        not yet "visited" do
            Find C', the corresponding child of N' T'
            = T' U Compare(C, C')
        End (* for *)
    End (* if *)
  End
```

Figure 2. The SelectTests algorithm.

4. Empirical Results

We have implemented a security regression testing tool as a support system. The objective of the support system is to prove the validity and applicability of the concepts and strategies presented earlier. The developed system helps testers and application maintainer understand the secure applications, identify code changes, support software and requirements updates, enhance, and detect change effects. It helps create a testing environment to select test cases to be rerun when a change is made to the trused application using our 3-phase regression testing methodology.

We use a prototype of a grant revoke application. We propose a random number of modifications to the application. Then, we study modifications using our maintenance tool and report the regions and the test cases that should be rerun according to the regression testing strategy implemented in the tool. The experimental work is done on a PC, running Pentium IV 3.2 GHz, 512 MB RAM, and using the PC version of progress.

The application is a grant revoke secure application as shown in Figure 3, which contains most of the language constructs, statement, and controls that we have studied. The variables identified in the trusted system can be identified as follows:

- PrivName: The type of object privilege that can be granted (all, select, insert, update, delete).
- Grantor: User granting an object privilege.
- Grantee: User being granted an object privilege.
- GranteeType: The type of grantee for a particular grant operation as defined in the first sentence of grant object privelege requirement, and a grantee can be a user, role, or public.
- Selected object: Object selected for a particular grant operation.
- Grantedobject: Object for which grant privelege have previously been granted (identified through grant option).
- Object owner: The owner of the object.

((grantor_owns_object) OR (has_grantable_obj_privs)) AND (grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID) OR granteeType = PUBLIC) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)	(NOT (grantor_owns_object)) AND (NOT (has_grantable_obj_privs)) AND (grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID)) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)
grant_obj_priv_OK = True	grant_obj_priv_OK = False

Figure 3. Behavioral specifications for "granting object privilege" capability.

A role is a gourp of related users, and the related variables are: RoleId and the GranteeRoleId.

Granting Object Privilege (GOP):

A normal user (the grantor) can grant an object priv. To another user, role or public (the grantee) only if:

a. The grantor owns the object.
b. The grantor has been granted the object privileges with the grant_option.

Grantor_owns_object_relation:

Grantor_owns_object = true if grantor = objOwner else = false.

Relation grantee_constraints:

There are three cases:

a. If the granteeType is user then the grantee is a user, and to ensure that the grantee is granted privileges as a user and not through the grantee's role, the RoleId must not be equal to granteeRoleId.
b. If the granteeType = role then the RoleId must be valid and the granteeRoleId must be equal to RoleId.
c. If the granteeType is public (all users) then the other vars could take any value.

Relation granted object privileges:

a. The selected object is the object for which the privilege was granted (the selected object is the granted object).
b. The privilege was granted with the option to grant others the privilege (grant_option is true).
c. The owner of the object is not the grantor.
d. The owner of the object is not the grantee.

Relation GrantObjPriv:

1. GOP (A): Grantor can grant privilege to a grantee because the grantor owns the object.
2. GOP (B): Grantor can grant privilege to a grantee because the grantor has been granted object privileges with Grant Option.

Also, the following situations must be verified: a) Grantor is not the grantee, b) All possible combinations of the GranteeType (user, role, public), and c) All possible privileges on operations (all, update, select, etc).

The difference between the true and false case for the GrantObjPriv is that the true case establishes the required conditions: The grantor_owns_object relationship that is associated with GOP (A) where the grantor owns the object, or the granted_obj_priv and grantee constraints that is associated with GOP (B).

The false case establishes the conditions where the grant operation fails: grantor is not the object owner, and grantor has not been granted object privelege.

4.1. Results

To aquire and analyse empirical results, the tool was used on the grant revoke trusted application and its modified versions. Figure 4 presents the CDG table of the application grant revoke.

The test history of the grant revoke secure application is divided into groups of tests, each represents a class of tests that reach a set of regions. The tests groups that form the original test suit is represented by the table in Figure 5.

Node #	Label	Father Id	Loop Id	Value
1	Entry	0	0	
2	Exit	1	0	
3	P1	1	0	If grantor = SelectedObjOwner then do:
4	R1	3	0	
5	S2	4	0	Grantor_owns_object = true.
6	S3	1	0	Def var f1 as logical.
7	S4	1	0	Def var f2 as logical.
8	S5	1	0	Def var f3 as logical.
9	S6	1	0	Def var f4 as logical.
10	P7	1	0	If selectedobjpriv = grantedobjpriv then do:
11	R2	10	0	
12	S8	11	0	F1 = true.
13	P9	11	0	If selectedobj = grantedobj then do:
14	R3	13	0	
15	S10	14	0	F2 = true.
16	P11	11	0	If selectedobjowner <> grantor then do:
17	R4	16	0	
18	S12	17	0	F3 = true.
19	P13	17	0	If selectedobjowner <> grantee then do:
20	R5	19	0	
21	S14	20	0	F4 = true
22	P15	1	0	If grant_option and f1 and f2 and f3 and f4 then do:
23	R6	22	0	
24	S16	23	0	Has_grantable_obj_privs = true.
25	S17	1	0	Define var f5 as logical.
26	S18	1	0	Define var f6 as logical.
27	S19	1	0	Define var f7 as logical.
28	S20	1	0	Define var f8 as logical.
29	S21	1	0	Define var f9 as logical.
30	S22	1	0	Define var f10 as logical.
31	S23	1	0	Define var valid_roleId as integer.
32	P24	1	0	If not ((grantor_owns_object) OR (has_grantable_obj_privs)) then do:
33	R7	32	0	
34	P25	33	0	If (selectedObjPriv = "ALL" OR SelectedObjPriv = "UPDATE" OR SelectedObjPriv = "SELECT" OR SelectedObjPriv = "INSERT" OR SelectedObjPriv = "DELETE") then do:
35	R8	34	0	
36	S26	35	0	F5 = true.
37	P27	33	0	If (granteeType = "user" OR (granteeType = "role" AND granteeRoleID = valid_roleid) or Granteetype = "public") then do:
38	R9	37	0	
39	S28	38	0	F6 = true.
40	P29	33	0	If (grantor <> grantee) then do:
41	R10	40	0	
42	S30	41	0	F7 = true.
43	P31	1	0	If ((grantor_owns_object) OR (has_grantable_obj_privs)) AND f7 AND f6 and f5 then do:
44	R11	43	0	
45	S32	44	0	Grantt = true.
46	R12	43	0	
47	S33	46	0	Grantt = false.

Figure 4. CDG table for initial grant revoke secure application.

Execution History/Traversed Regions	Test Class
Entry, R1	T1
Entry, R2, R3	T2
Entry, R2, R4, R5	T3
Entry, R6	T4
Entry, R7, R8	T5
Entry, R7, R9	T6
Entry, R7, R10	T7
Entry, R11	T8
Entry, R12	T9

Figure 5. Original test suite table.

In Table 2, we present a summary of test cases presented. We classify these results into two parts. In the first part, we give the results of phase one of our regression testing methodology for secure applications. In the second part, we give the results of phase two, which include a count of test case classes selected by our tool.

Phase 1 results include a list of the following:

1. Directly affected regions.
2. Indirectly affected regions.

Phase 2 results include a list of the following:

1. Test case classes selected by our strategy.
2. Percentage of test case reduction as shown in Figure 6.

Table 2. Summary of results.

Modification Cases	Directly Affected Regions	Indirectly Affected Regions	Percentage of Test Case Selections	Percenatage of Reduction
1..Modify Statement	26	10	27	72.7
2. Add Statement	6	5	16.5	83.25
3. Delete Statement	20	9	33.16	67
4. Move Statement	14	1	30.5	72.25
5.Modify Predicate	16	7	33	66
6. Delete Predicate	9	7	22	88
7. Add Predicate	16	14	65.37	33.5
8. Move Predicate	30	17	87	13

4.2. Discussion of Results

Using our new strategy in trusted regression testing, the tool did a good test reduction and selection job. Out of 80 test vectors of the original test suite used to test the trusted application, we had on average 39% of test cases selected with average of 17 regions directly affected, and

9 regions indirectly affected. This ratio is greatly affected by the number of modifications and the distribution of test cases within the regions. The number of affected regions per modifications depends on the interaction level between the regions in the trused application. On the other hand, execution time was negligable, and this varies according to the size of the trusted application.

We repeated each experiment five times for each (base trusted program, modified version). The experminetal results showed that our strategy reduced the size of selected tests, and the overall savings were promising.

In fact, our tool reduced test case by more than 60% on average comparing to select-all approach. On the other hand, 60% reduction of test cases is equal to days, hours, even weeks of testing effort. These results show that our approach is precise, and directed towards safety, and greater precision in regression testing of trusted applications.

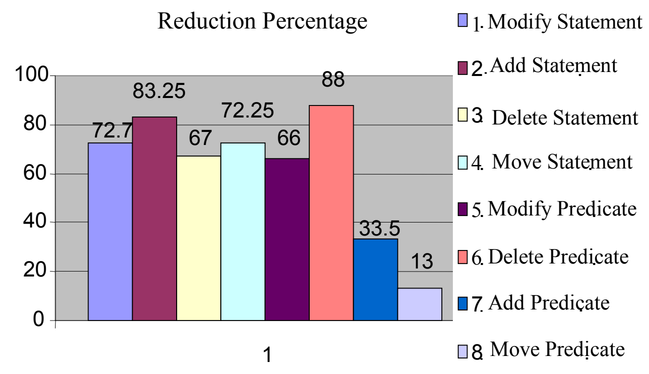

Figure 6. Percentage of test case reduction (total test cases = 40).

5. Conclusion and Future Work

In this paper, we presented regression testing algorithms for trusted database applications that are efficient and more general than other techniuqes. The algorithms handle regression test selection for signle procedures and groups of interacting procedures. They handle language constructs and different types of program modifications for procedural languages.

Future work includes using more components for case studies, performing additional empirical results to evaluate the effectiveness of our technique, and applying a variety of code changes to our tool in a production re-test environment.

References

[1] Agrawal H., Horgan J., Krauser E., and London S., "Incremental Regression Testing," *in Proceedings of the Conference on Software Maintenance*, pp. 348-357, September 1993.

[2] Bates S. and Horwitz S., "Incremental Program Testing Using Program Dependence Graphs," *in Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[3] Benedusi, A. Cimitile, and De Carlini U., "Post-Maintenance Testing Based on Path Change Analysis," *in Proceedings of the Conference on Software Maintenance*, pp. 352-61, October 1988.

[4] Fischer K. F., Raji F., and Chruscicki A., "A Methodologyf or Retesting Modified Software," *in Proceedings of the National Telecommunications Conference*, vol. B-6-3, pp. 1-6, November 1981.

[5] Harrold M. J. and Soffa M. L., "An Incremental Approach to Unit Testing During Maintenance," *in Proceedings of the Conference on Software Maintenance*, pp. 362-367, October 1988.

[6] Harrold M. J. and Soffa M. L., "Interprocedural Data Flow Testing," *in Proceedings of the Third Testing, Analysis and Verification Symposium*, pp. 158-67, December 1989.

[7] Hartmann J. and Robson D. J., "Techniques for Selective Revalidation," *IEEE Software*, vol. 16, no. 1, pp. 31-8, January 1990.

[8] Laski J. and Szemer W., "Identification of Program Modifications and its Applications in Software Maintenance," *in Proceedings of the Conference on Software Maintenance*, pp. 282-90, November 1992.

[9] Leung H. K. N. and White L. J., "A Study of Integration Testing and Software Regression at the Integration Level," *in Proceedings of the Conference on Software Maintenance*, pp. 290-300, November 1990.

[10] Ostrand T. J. and Weyuker E. J., "Using Dataflow Analysis for Regression Testing," *in Proceedings of 6th Annual Pacific Northwest Software Quality Conference*, pp. 233-47, September, 1988.

[11] Rothermel G. and Harrold M. J., "A Comparison of Regression Test Selection Techniques," *Technical Report 114*, Clemson University, Clemson, SC, April 1993.

[12] Rothermel G. and Harrold M. J., "A Safe Efficient Regression Test Selection Technique," *ACM Transactions on Software Engeneering and Methodology*, vol. 6, no. 2 , April 1997.

[13] Sherlund B. and Korel B., "Modification Oriented Software Testing," *in Proceedings of Quality Week'1991*, pp. 1-17, 1991.

[14] Taha A. B., Thebaut S. M., and Liu S. S., "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis," *in Proceedings of the 13th Annual International Computer Software and Applications Conference*, pp. 527-34, September, 1989.

[15] Yang W., "Identifying Syntactic Differences Between Two Programs," *Software Practice and Experience*, vol. 21, no. 7, pp. 739-55, July 1991.

[16] Yau S. S. and Kishimoto Z., "A Method for Revalidating Modified Programs in the Maintenance Phase," *in Proceedings of the Eleventh Annual International Computer Software and Applications Conference* (*COMPSAC'87*), pp. 272-277, October 1987.

Ramzi A. Haraty is an associate professor and the assistant dean of the School of Arts and Sciences at the Lebanese American University in Beirut, Lebanon. He is also the chief financial officer of the Arab Computer Society. He received his BSc and MSc degrees in computer science from Minnesota State University, Minnesota, and his PhD in computer science from North Dakota State University, North Dakota. His research interests include database management systems, artificial intelligence, and multilevel secure systems engineering. He has well over 80 journal and conference paper publications. He is a member of Association of Computing Machinery, Arab Computer Society and International Society for Computers and Their Applications.

Wissam Chehab is a Master of computer science student at the Lebanese American University in Beirut, Lebanon. His research interests inclde database management systems and software engineering.