Lebanese American University

Regression Testing for Trusted Database Applications

By

Wissam Mohieddine Chehab

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

Thesis Advisor: Dr. Ramzi A. Haraty

January 2005

Lebanese American University
Graduate Studies

We hereby approve the thesis of

Wissam Mohieddine Chehab

Candidate for the Master of Science Degree *

_____

Dr. Ramzi A. Haraty
Associate Professor of Computer Science
Lebanese American University

_____ [redacted] _____

Dr. May Hamdan
Assistant Professor of Mathematics
Lebanese American University

_____ [redacted]

Dr. Zahi Nakad
Assistant Professor of Computer Engineering
Lebanese American University

[redacted]

Date:

*** We also certify that written approval has been obtained for any proprietary material contained there in.**

_____

Dedication

I dedicate this thesis

To my parents

To my brother

To my sisters

To my friends

Acknowledgment

I would like to thank my advisor Dr. Ramzi A. Haraty for his guidance throughout my M.S studies. Thanks also to Dr. May Hamdan and Dr. Nakad for being on my thesis commtee.

I would like to express my gratitude to the Lebanese American University whose financial support during my graduate studies made it possible.

Finally I would like to thank my parents for their long support.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1) Background

Regression testing is any type of software testing, which seeks to uncover regression bugs. Regression bugs occur as a consequence of program changes [9].

After unit, integration and system testing, the newly developed system enters "maintenance mode", and once it is in use, new functionality may be desired or undetected faults may surface. This will require the completed, tested system to be modified, which will mean re-validation and re-testing.

Common methods of regression testing are re-running previously run tests and checking whether previously-fixed faults have re-emerged.

No matter how well software is tested and conceived, it will eventually have to be modified in order to fix bugs or respond to changes in user specifications. 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. On the other hand, for software revalidation by arbitrarily omitting test cases used in regression testing is risky [16]. Thus, we need to investigate methods to select safe subsets of effective fault-revealing regression test cases to revalidate the application.

Regression testing procedure is quite straight forward. First, select a subset of test cases to be run on the modified program; second retest the modified program and establish its correctness relative to the selected tests; third if necessary, create new tests for the modified program.

When regression testing is performed, the purpose would be 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) [19]. 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 analyses 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 [17]:

1. How to select regression tests from an existing test suite (the RTS problem)?
2. 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 [9]:

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 [9] 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.

1.2) Scope of the Thesis

In this thesis the work is narrowed to cover only regression testing for secure interfaces. A secure interface consists of the elements that are involved in enforcing the system's security policy. Examples of secure interface kernel include interfaces for operators , administrators, users, etc. The secure interface contains software elements of the trusted computing base (TCB) that are involved in implementing system security policies. So carrying out the new regression testing technique helps the security kernel to satify its requirements. Security regression testing is a reliable technique of gaining assurance that a system operates withing the constraints of a given set of policies and mechanisms. Despite its usefulness to system security, little research has been done in the discovery of regression testing methods specifically tailored for the security area. Instead, security testing relies on scattered collections of rules of thumb and testing philosphies [20]. Brute force approaches were used for security regression testing, such as black-box testing approach, which cause a large numbers of unnecessary tests to be generated; however, no attempt has been made to present suitable approaches to security regression testing.

This thesis presents a new selective approach to security regression testing. This approach constructs control dependence graphs for program versions. These versions are used to determine which tests from the existing testing suite may exhibit changed behavior on the new version.

A bundle technique is presented in this thesis that is neither coverage-criteria based nor requires complete information on corresponding program components. It constructs graphs representing control dependence of a program and its modified version, and uses these graphs to identify tests that may reveal different behaviour.

The new algorithm detects portions of code that differ in the two versions of the program, and selects for retests all tests that traverse these regions. The algorithm is general and not restricted to a subset of language constructs, or limited to a particular type of program modifications.

Chapter 2

Literature Review

2.1) Introduction

Estimates indicate that software maintenance activities account for as much as two-thirds of the cost of software production [39]. One necessary but expensive maintenance task is regression testing, performed on a modified program to introduce confidence that changes thave 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. An alternate approach, selective retest, chooses a subset of tests from the old test set, and uses this subset to test the modified program.

Although many techniques for selective retest have been developed [1, 3, 4, 13, 17, 20, 21 28, 29 37, 40, 41], 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 [3, 19, 21] 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 [1, 4, 17, 29, 37, 40, 41], 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 [10, 13, 20] 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.

These philosophies lead selective retest methods to distinctly different results in test selection. Despite these differences, there are identified categories in which selective retest approaches can be compared and evaluated. These categories are inclusiveness, precision, efficiency, generality, and accountability [9].

- Inclusiveness: measures the extent to which a method chooses tests that will cause the modified program to produce different output.
- Precision: measures the ability of a method to avoid choosing tests that will not cause the modified program to produce different output.
- Efficiency: measures the computational cost and automation ability, and thus practicality, of a selective retest approach.
- Generality: measures the ability of a method to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications.
- Accountability measures a method's support for coverage criteria, that is, the extent to which the method can aid in the evaluation of test suite adequacy.

These categories form criteria for evaluation and comparison of selective retest approaches.

2.2) Regression Testing

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 [9]. 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 [43].

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, Figure 1 shows test history information for procedure AVG.

```
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)
S10     return(avg)
```

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

Figure 1: AVG and its test history information.

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 $\in$ 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.

2.3) Evaluation Criteria

The five categories described below are the criteria for evaluating selective retest approaches.

2.3.1 Inclusiveness

Inclusiveness measures the extent to which S chooses modification-revealing tests from T for inclusion in T'. We define inclusiveness relative to a particular program, modified program, and test set as follows: suppose T contains n modification-revealing tests, and S (regression testing strategy) selects m of these tests. The inclusiveness of S relative to P, P', and T is the percentage calculated by the expression ((m/n) * 100).

For example, if T contains 50 tests, 8 of which are modification-revealing, and S selects only 2 of the 8 modification-revealing tests, then S is 25% inclusive relative to P, P', and T. If a method S always selects all modification revealing tests, we call S safe. If for all P, P and T, S is 100% inclusive relative to P, P and T then S is safe.

For an arbitrary choice of S, P, P', and T, there is no algorithm to determine the inclusiveness of S relative to P, P', and T. We can also perform experiments on a group W of programs, modified programs, and test sets to measure the strategy's inclusiveness relative to W by taking the average of the relative inclusiveness of S for each member of W .

2.3.2 Precision

Precision measures the extent to which a selective retest strategy omits tests that are non-modification revealing. The definition of precision relative to a particular program, modified program and test set, is as follows:

Suppose T contains n nonmodification-revealing tests, and S selects m of these tests. The precision of S relative to P, P', and T is the percentage calculated by the expression ((m/n) * 100).
For example, if T contains 50 tests, 44 of which are non-modification-revealing with respect to P', and S omits 33 of these 44 tests, then S is 75% precise relative to P, P', and T.

As with inclusiveness, there is no algorithm to determine, for an arbitrary choice of S, P, P', and T, the precision of S relative to P, P', and T. However, we can still measure precision in several ways. We may be able to show that S is precise by proving that S omits a superset of the non-modification-revealing tests.

If the precision of a strategy relative to every P, P' and T is 100%, we say the strategy is precise. A precise strategy always selects only modification-revealing tests, while an imprecise strategy selects some tests that cannot produce different output.

2.3.3 Efficiency

We measure the efficiency of a selective retest method in terms of its space and time requirements. Space efficiency is affected by the test history and program analysis information a strategy must store. Where time is concerned, a selective retest strategy is more economical than a retest-all strategy if the cost of selecting T' is less than the cost of running the tests in T-T' [16]. Thus, efficiency varies with the size of the test set that a method selects, as well as with the computational cost of that method. Methods for evaluating algorithms are well understood and will not be discussed in this thesis. However, we discuss several factors that must be considered when evaluating the efficiency of selective retest algorithms.

One factor influencing the computational expense of a selective retest method is whether or not the method must calculate information on program modifications. If a method must determine which program components have been modified, deleted, and added, or construct a mapping between components of P and P', that method may require more computational resources than a method that does not calculate such information.

2.3.4 Generality

The generality of a selective retest method is its ability to function in a wide and practical range of situations. Selective retest algorithms should function in the presence of arbitrarily complex code modifications. Also, although we could apply different methods in different settings, we prefer methods that handle all types of language constructs, and large classes of programs. The need for information on program modifications is also a generality issue since requiring knowledge of modifications may impose unreasonable restrictions.

2.3.5 Accountability

Studies suggest that structural test coverage criteria increase the effectiveness of testing [8]. If a program is initially tested with such a criterion, then after modifications it is desirable to confirm that the criterion remains satisfied. Alternatively, if a program is not initially tested using a coverage criterion, it is still possible to apply a criterion at regression test time, ensuring that all new or modified portions of the code have been covered properly [35]. The term accountability to refer to the extent to which a selective retest method promotes the use of structural coverage criteria. Selective retest methods may promote this use by identifying unsatisfied program components, or selecting tests that maximize the coverage achievable. Both coverage and minimization methods facilitate such efforts.

2.4) Selective Retest Approaches

In this section, we will evaluate a representative sample of regression testing methods, other strategies are evaluated in [9].

2.4.1 Minimization Methods

Minimization approaches to selective retest have been proposed [3, 19, 21]. We discuss Hartmann and Robson's method, since it extends the other methods , and shows the strengths and weaknesses of minimization approaches.

The method uses systems of linear equations to express relationships between tests and program segments (basic blocks). These equations are obtained from matrices that track the segments of code reached by test cases, and the segments reachable from other segments. The solution to such a system of equations identifies a minimal set of tests T' such that each segment reachable from a changed segment is exercised by at least one test in T'. Dataflow information is used to ensure that only tests that traverse affected uses are selected.

This approach, like other minimization approaches, is not safe. If several tests exercise a particular modified statement and all of these tests exercise a particular affected statement, only one such test is selected, unless the others are selected for coverage elsewhere. Some of the tests that are omitted may produce different output if executed. For example, suppose statement S1 in procedure AVG (Figure 1) is erroneously modified, to "count=1". Tests T1 and T3 both traverse S1 and reach affected statement S9, which uses the value of count. Hartmann and Robson's method selects only one of these tests, omitting the other. However, only test T3 exposes this fault and if T1 is chosen instead, the fault will not be detected. Hartmann and Robson's approach also may omit tests that can reveal faults caused by non-dataflow dependencies, such as tests reaching S2 from S1 in the fragments on the left in Figure 2.

Fragment F 1	**Fragment F3**	**Fragment F4**
S1. call SetMode()	S 1. if P then	P 1. if P then
S2. call DrawLine(point1,point2)	S2. (do something)	S2. (do something)
	S3. a:= 2	
	endif	endif
Fragment F2	S4. if Q then	S4. if Q then
	S5. (do something)	S5. (do something)
	S6. print(a)	
----------	endif	endif
S2. call DrawLine(point1,point2)		

Figure 2: fragments showing statement deletion.

This method omits tests that are non-modification-traversing by ignoring tests that do not execute changed segments. The method also uses dataflow information to further increase precision.

Since Hartmann and Robson's method is a minimization method, it returns small test sets and thus reduces the time required to run regression tests. The method can be fully automated. However, due to the calculations required for solving systems of linear equations, the approach may be data and computation intensive on large programs. This method is defined and implemented for "C," and can handle all "C" structures. The method depends only on identifying code segments, so it could be implemented for any procedural language. The method may also be extended to handle inter-procedural test selection, by treating entire routines rather than basic blocks as segments. However, this extension reduces precision by admitting tests that are non-modification-traversing, such as tests that traverse a modified procedure but do not actually traverse modified code in the procedure. More importantly, Hartmann and Robson's method is defined only for situations where code modifications do not alter control flow. Thus, the method does not handle addition, deletion, or modification of predicate statements. Note that any work to handle changes in control flow will force reanalysis of the changed program, which is expensive. This method is oriented to achieve structural coverage of a program at a basic block level; it establishes such coverage by reusing as many existing tests as possible, without selecting tests that are redundant in terms of coverage.

2.4.2 Coverage Methods

A majority of existing selective retest methods axe most described as coverage methods. These include approaches proposed by Bates and Horwitz [37], Benedusi, Cimitile, and De Carlini [4], Gupta, Harrold, and SofFa [35], Harrold and Soffa [28, 29], Leung and White [17], Ostrand and Weyuker [41], Taha, Thebaut and Liu [1], and Yau and Kishimoto [40]. In this section we discuss the methods proposed by Harrold and Soffa, and Bates and Horwitz, since these methods let us illustrate some important facets of the use of coverage methods.

2.4.2.1 Harrold and Soffa's Method

Harrold and SofFa [28, 29] present a test selection method based on dataflow testing techniques. Their approach identifies changed def-use pairs in a program, and selects tests that exercise these pairs.

Inclusiveness: Harrold and Soffa's method specifically selects all tests that cover affected pairs, thereby selecting a superset of the set selected by Hartmann and Robson's method. Nevertheless, the approach may omit modification-revealing tests in at least three ways, and thus is not safe. First, the approach may omit tests that exercised statements deleted from P. Second, by relying only on data dependencies as a guide in test selection, the method misses tests that may be exposed by other forms of dependencies. Hence, for code changes such as that showed on the left in Figure 2, the method does not select any tests. This method may omit tests that execute modified output statements that contain no variable uses, although these statements may cause the program to produce different output.

Precision: Because Harrold and Soffa's approach only selects tests that traverse new or modified definition-use pairs, all tests selected necessarily traverse new or modified statements, so the method selects only modification-traversing tests. Moreover, by selecting tests that exercise definition-use pairs, Harrold and Soffa's approach is capable of greater precision than methods that select all modification traversing tests.

Efficiency: Harrold and Soffa's approach requires storage and/or calculation of dataflow information, but dataflow calculation is at worst an $O(n^2)$ operation that is well understood and is accomplished by many compilers [9]. However, the approach also requires knowledge of program modifications, and typically assumes that these modifications will be provided through a program development environment. To be efficient, such environments must handle incremental updates of dataflow information as changes are applied to programs. In doing so these environments acquire additional computation and storage costs.

Generality: Harrold and Soffa's approach is fairly general, because it requires only control flow graphs and test execution histories. Also, the method handles structural program changes

Accountability: Harrold and Soffa's method is highly accountable, lending direct support for dataflow coverage criteria [9].

2.4.2.2 Bates and Horwitz's Method

Bates and Horwitz [37] present a test selection method based on program dependence graph adequacy criteria. The program dependence graph encodes both control and data dependence information for a procedure [18]. Bates and Horwitz use slicing algorithms to group statements in P and P' into execution classes such that a test that executes any statement in an execution class executes all statements in that class. Next, they identify affected statements, which are statements that may exhibit different behavior in P', by comparing slices of corresponding points in P and P'. Finally, they select for retest all tests that exercise any statement in the same execution class as an affected statement.

Inclusiveness: Bates and Horwitz's method successfully identifies tests that traverse modified statements, because all modified statements are identified as "affected". Moreover, by using execution classes the method is able to identify tests that traverse new statements, but this method does not report for deleted statements, and thus is not safe.

Precision: The technique of selecting tests through affected statements, which ensures selection of tests of new statements, causes the method to admit tests that are non-modification traversing, and since Bates and Horwitz's method relies upon slices along data and control edges in the program dependence graph, its precision is adversely impacted by assumptions required in the presence of aliasing and dynamic memory allocation. As with dataflow-based methods, some precision loss can be prevented by using algorithms for identifying alias information [15].

Efficiency: Bates and Horwitz's method may require a large number of program slices. The method computes a control slice for every statement in P and every statement in P'. It then computes a backward slice on each statement in P that has a corresponding statement in P, and each statement in P' that has a corresponding statement in P. Moreover, since slice comparisons must be done with respect to statements in P', the costs of slicing on P' are obtained after modifications are complete, when testing has entered the critical phase. Finally, the method requires prior knowledge of changes, in the form of a mapping of statements in P to their modified versions in P'. This mapping must be computed after modifications are complete, hence in the critical phase.

Generality: this approach supports all types of program modifications, but is presented only for a restricted set of language constructs. Furthermore, no method for inter-procedural testing is suggested.
Accountability: Like other coverage methods, Bates and Horwitz's method is highly accountable, aiding in the satisfaction of program dependence graph coverage criteria.

2.4.3 Safe methods

Only three safe methods for selective regression testing exist. These are the methods of Laski and Szermer [20], Rothermel and Harrold [10], and Agrawal, Horgan, Krauser, and London [13]. We will discuss the second approach because it is based on control dependence graphs which we will refer later in our thesis.

2.4.3.1 Rothermel and Harrold's Method

Rothermel and Harrold [10] present a safe regression test selection method based on control dependence graphs. Control dependence graphs encode control dependence information for a procedure. This method constructs control dependence graphs for P and P, and instruments tests to report the regions (groups of statements sharing common control conditions) executed by the tests. Any test that executes a region of code that contains a changed statement is selected for retest.

Inclusiveness: Since this approach selects every modification-traversing test, it is safe. The approach also handles new and deleted code.

Precision: the method selects only modification-traversing tests, and thus is more precise than methods that choose non-modification revealing tests. However, the method makes no use of dataflow or other information that could reduce the size of selected test sets further.

Efficiency: The running time of the method is bounded by the time it takes to construct control dependence graphs, which is $O(n^2)$, so the method is efficient. Moreover, much of the computation required by the method, such as construction of the control dependence graph for P and collection of test history information, may be completed during the preliminary regression testing phase. The only work that must be done during the critical phase of regression testing is the construction of the control dependence graph for P, and the execution of a tree walk on P and P'. Furthermore, since the goal of Rothermel and Harrold's method is to identify tests, the method can examine fewer parts of programs than Laski and Szermer's method, for which the goal is to identify corresponding program components. This improvement is due to the fact that when Rothermel and Harrold's algorithm walks control dependence graphs looking for differing regions, it does not need to explore sub-graphs within a changed region; it takes advantage of the fact that all tests entering the region are modification-traversing and does not look at enclosed regions. Moreover, the approach can be fully automated. Finally, since Rothermel and Harrold's method requires test histories

listing just the regions executed by a test, its space requirements are much smaller than methods requiring test histories on a per statement basis.

Generality: the method applies to all programs in procedural languages, because control dependence graphs may be constructed directly from control flow graphs. The method also applies to all types of program modifications.

Accountability: By identifying changed regions of code, the method identifies the areas of the code in which coverage needs to be verified, but does not aid with any particular coverage criteria.

2.5) Security Regression Testing

Security regression testing is a good technique of gaining assurance that a modified system operates within the constraints of a given set of policies and mechanisms. Despite its obvious usefulness to computer security, little research has been done in the discovery of regression testing methods specifically tailored for the security area. Instead regression security testing relies on scattered collection of rules of thumb and testing philosophies. However, no attempt was made to date to explore suitable approaches for security regression testing.

Moreover, the evaluations show that a majority of current approaches are concerned with coverage or minimization rather than safety. Accountability is an important issue since coverage criteria are useful for improving test adequacy. However, in many practical situations the most important concern is a method's safety. Testing professionals are hesitant, in practice, to discard any tests that may expose errors. Thus, effective safe approaches are needed. Existing safe approaches would benefit most from improvements in precision and accountability.

As the evaluations also suggest, one major drawback for most current methods is their need for information on program modifications or mappings between "old" and "new" sections of code. This need leads techniques to either assume that knowledge of modifications will be provided by an incremental editor, or that some algorithm will be used to calculate a mapping. The former assumption adds requirements to the production environment that may not be easily satisfied.

The latter assumption increases the cost of the method and forces more work to be done during the critical phase of regression testing.

Finally, few unsatisfactory solutions have been offered to the problem of interprocedural regression test selection.

So our work will introduce a new technique for security regression testing, which will be based on control dependence graph. The aim of our method is to select every set of tests from the original suite that can expose faults in the modified application, and if necessary create new ones possibly to meet coverage criterion at a reasonable cost. Under these conditions the method is safe. Unlike other methods, the new method should handle all language constructs and all types of program modifications.

Chapter 3

Security Regression Testing Technique

3.1) Introduction

Software maintenance activities can account for as much as two-thirds of the overall cost of software production [39]. 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 retest-all 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 Background

The following notation is used throughout the rest of this chapter. Let P be a procedure or program, P' be a modified version of P, and S and S' be the specifications for P and P' , respectively.

```
Procedure MedRec
S1. Read(user, password)
S2. find permission where user = usr and
                password = pwd no-lock.
P3. for each medrec
P4. If medrec.perm_level<=usr_perm_level
S5.     Browse_medrec:add(medrec).
    Else
S6.     Msg(full access denied, general info only)
S7.     Browse:add(partial_medrec)
    Endif
S8.     Refresh Browse_medrec
    End_For_each
S9.  msg(read complete!)
S10. release(user, password)
```

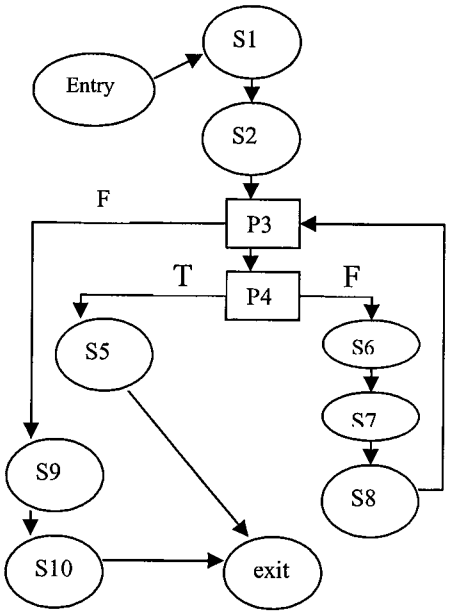


Figure. 3.1 Procedure MedRec and its CFG

P(i) refers to the output of P on input i, P'(i) refers to the output of P' on input i. S(i) refers to the specified output for P on input i, and S'(i) refers to the specified output for P' on input i.

Let T be a set of tests (a test suite) created to test P. A test is a three-tuple, (identifier, input, output), in which an identifier identifies the test; input is the input for that execution of the program; and output is the specified output, S(input), for this input. For simplicity, the outcome refers to a test (t, i, S(i)) by its identifier t and refers to the outputs P(i) and S(i) of test t for input i as P(t) and S(t), respectively.

3.2.1 Control Flow Graphs

A control flow graph (CFG) for procedure P contains a node for each simple or conditional statement in P; edges between nodes represent the flow of control between statements. Figure 3.1 shows procedure MedRec and its CFG. In the figure, statement nodes, shown as ellipses, represent simple statements. Predicate nodes, shown as rectangles, stand for conditional statements. Labeled edges (branches) leaving predicate nodes represent control paths taken when the predicate evaluates to the value of the edge label. Statement and predicate nodes are labeled to indicate the statements in P to which they correspond. The figure uses statement numbers as node labels.

However, the actual code of the associated statements could also serve as labels. Case statements can be represented in CFGs as nested if-else statements. In this case, every CFG node has either one unlabeled out edge or two out edges labeled "T" and "F". A unique entry node and a unique exit node represent entry to and exit from P, respectively. The CFG for a procedure P has size and can be constructed in time, linear in the number of simple and conditional statements in P [18].

3.3 Program Dependence Graph

3.3.1 Introduction

Structural testing techniques use intermediate representations of programs to select test data and determine test adequacy. A common program representation is the control flow graph, in which each node represents a program statement, and each edge

represents a transfer of control between statements. Several recent testing techniques make use of control dependence information that is not explicitly represented in a control flow graph. For example, control dependence information has been used to select data and determine adequacy [2], and to extend data flow testing techniques [6]. Control dependence information has also been used to generate reduced test sets for programs [11]. Moreover, several techniques used for regression testing [5, 35, 37] need control dependence information to identify the retesting required after changes are made to a program. Finally, both static and dynamic slicing techniques require control dependencies [7, 12, 14]. Thus, a program representation that contains explicit control dependence information is extremely useful for testing.

One program representation that encodes both control and data dependencies is the program dependence graph (PDG) [18]. Previous techniques for PDG construction [18, 34] rely on a control flow graph to identify control dependencies and compute data flow information. However, building a control dependence sub-graph during the parsing phase of compilation using these techniques is only applicable to structured programs.

We present a new technique for constructing a PDG that handles both structured and unstructured programs. Our algorithm constructs the PDG during the parse phase of compilation and the resulting PDG contains control flow information along with the usual control and data dependence information. For structured programs, the algorithm constructs the control dependence sub-graph without using the control flow graph of the procedure. If structured transfers of control, such as "break", "continue" and "exit", are encountered, the algorithm can still identify the program's structure and construct the control dependence sub-graph without requiring the entire control flow graph. If explicit "goto" statements are encountered, the resulting graph requires additional processing to obtain the exact control dependence sub-graph.

A major part of the program dependence graph construction algorithm involves ordering the nodes in the control dependence sub-graph to identify control flow in the program. Our ordered control dependence sub-graph incorporates control flow either implicitly through node order or explicitly through the creation of control flow edges. To obtain the data dependence sub-graph, we perform data flow analysis directly on

the control dependence sub-graph augmented with explicit control flow information where required. Using data flow sets, we add data dependence edges to get the data dependence sub-graph and the PDG. There are several advantages to use this approach. For many programs, our approach may result in substantial savings in the time and memory it takes to construct the PDG, since we eliminate construction and analysis of the control flow graph. However, our PDG can be used for all applications that require information on control flow, such as data flow analysis, test case generation, and regression testing. Further, our PDG contains a program's control flow but retains the program's exact control dependencies, whereas previous techniques incorporated control flow but only approximated control dependencies.

3.3.2 Background

A PDG encodes control dependencies in a control dependence sub-graph (CDS). To facilitate analysis and obtain the CDS, a control flow graph is often augmented with unique entry and exit nodes. Figure 3.2 gives a program segment and its control flow graph. For statements (nodes) X and Y in a control flow graph, if X is control dependent on Y then Y must have two exits where one of the exits from Y always causes X to be executed, and the other exit may result in X not being executed [18]. We say that X is control dependent on Y with the label (true or false) that definitely causes X to execute. A statement X may be control dependent on several statements in the program. Since these statements form nested sequences of control dependencies, we can always identify immediate control dependencies for X. For example, in the control flow graph of Figure 3.2, statement (node) S6 is control dependent on both P5-false and P3-false since both of these must hold for statement S6 to execute, but statement S6 is immediately control dependent only on P5-false. The nodes in a CDS represent statements or regions of code that have common control dependencies. The CDS of the PDG for the program segment of Figure 3.2, is shown on the right of Figure 3.2. The node numbers correspond to statement numbers in the program. A CDS contains several types of nodes. Statement nodes, shown as ellipses in Figure 3.2, represent statements in the program. Circles represent region nodes, which summarize the control dependencies for statements in the region. Predicate nodes, from which two edges may originate, are represented as squares in

the figure. Predecessors of a node in the CDS are known as its parent regions, and successors of a region or predicate node are known as its children. Further, children with the same parents are known as siblings of each other in the CDS. Control dependencies are explicitly represented in the CDS. For example, it is clear from the CDS that statement (node) S6 is control dependent on both P5-false and P3-false, but immediately control dependent on P5false. Region nodes in the CDS summarize the control dependencies of a group of statements. For example, in Figure 3.2, both statement node S6 and predicate node P7 are control dependent on P5-false; without region node R3, there would be two edges labeled "F" from node P5. A second sub-graph of the PDG, the data dependence sub-graph (DDS), encodes data dependencies among statements. The DDS is obtained by creating edges between nodes in the CDS to represent data dependencies. For example, in Figure 3.2, the DDS would contain an edge from S2 to S6 since S2 defines sum and S6 uses that definition of sum. Similarly, there would be data dependence edges for each of the other definition-use pairs in the program. In Figure 3.2, the DDS are omitted for simplicity.

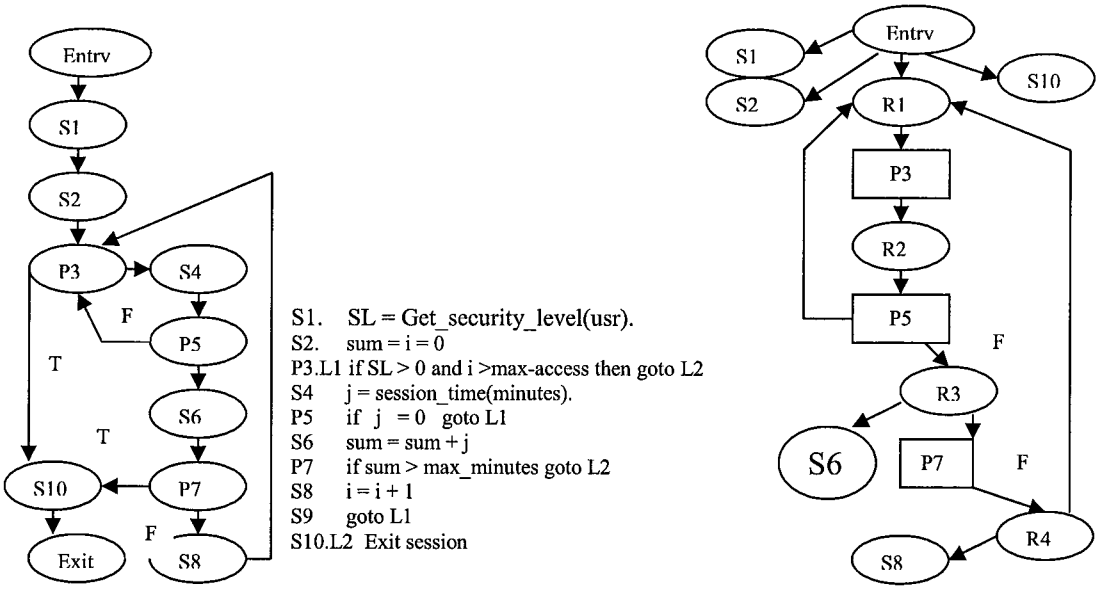


Figure 3.2: Program applying time and sessions number access constraints on the user, with its control flow graph and its control dependence graph.

3.3.3 PDG Construction

The algorithm for constructing the PDG takes procedure P and produces the PDG for P in two steps. Step one constructs the CDS for P, and step two uses the CDS to construct the DDS for P. We address the two steps in the next two subsections.

3.3.3.1 Constructing the CDS

We give an overview for constructing the CDS; details are given in [27]. Our algorithm ConstructCDS, listed in Figure 3.3, accepts an abstract syntax tree (AST) for a procedure P and outputs the CDS for P. For simplicity, we assume that P is written in a language containing the following types of statements: simple statements (assignment), structured statements (`if-else`, while), structured transfers of control (continue, break, return, and exit), and unstructured transfers (goto). Other constructs and statements are handled similarly. ConstructCDS uses a left to right preorder traversal of P's AST and takes appropriate actions as each node is encountered, and as each sub-tree in the AST is reduced. ConstructCDS handles two important tasks:

1. It creates CDS nodes that represent exact control dependencies in P, and
2. It encodes control flow for use in algorithms that require it.

ConstructCDS uses a stack, CDStack, and the usual set of stack operations, to maintain nesting, and therefore control dependencies. When ConstructCDS begins, an "entry" region that represents the entire procedure is created and placed on CDStack. Subsequently, nodes are added to the CDS as children of the node that is on the top of CDStack (the active node). Whenever a statement that begins a new region of control dependence, such as a structured statement or label, is found, a new node is added to the CDS and pushed onto CDStack. Subsequent statements, added under this new active node, are then properly nested. When the end of a structured statement is detected, CDStack is popped. In most cases, our node order in the CDS encodes control flow implicitly. From a parent region, control flow moves to the leftmost child, then left to right among siblings until it reaches a region or predicate node. At predicate nodes control flows down outgoing control dependence edges. By ordering nodes as they are created, ConstructCDS preserves this implicit control flow.

```
Algorithm ConstructCDS
Input     AST: abstract syntax tree for procedure P with root program.
Output    CDS: control dependence sub-graph for P.
Declare   CDStack: stack of information about region nodes
          Push (N), Pop (N): push and pop operations on CDStack. AdjustFlag: Boolean.
          LabelTable: table to record labels for control flow edges
          AddNode (N1, N2, L): create CDS node N1, add it to CDS under N2 with label L
          (Active: the region on top of CDStack)
Begin
   While more AST nodes do
   Get ASTNode

   Case ASTNode is
        Program:
          Create entry region as root of CDS; Push (entry)
          Create exit region for later use
        Assignment:
          AddNode (statement, Active, -)
        While:
          AddNode (header region, Active, -); Push (header region) AddNode
          (predicate, header region, -);
          AddNode (body region. predicate, true); Push (body region)
        End while:
          Pop (body region)
          Resolve unresolved nodes
          Pop (header region)
          Replace top of CDStack with "while" follow region if necessary. Set AdjustFlag
          if required
        If-else:
             AddNode (predicate, Active, -); Push (predicate)
        Begin if-clause/else-clause:
          AddNode (region, Active, true or false);
          Push (region)
        End if-else:
          Pop (if-else region)
          Replace top of CDStack with if-else follow region if necessary
        End if-clause/end else-clause
          List unresolved nodes Pop (if/else-clause regions)
          If else-clause region has no children remove it from CDS
        Structured control transfer:
          AddNode (statement, Active, -)
          Calculate follow information and update CDStack insert special flow and dependence
          edges
        Goto:
          AddNode (statement, Active, -); update LabeLTable; set AdjuatFlag
        Label:
          AddNode (label region, Active, -); Push (label region);
          Update LabelTable
     End case;
   End while;

   Add exit region to CDS under Active; Resolve remaining unresolved nodes and control dependence
   edges for breaks; if AdjuatFlag then call Adjust
End.
```

Figure 3.3 Algorithm to construct the CDS of a PDG, complete with control flow edges [34].

3.3.3.2 Constructing the DDS

After the CDS is constructed using our technique, we perform data flow analysis on it, and use the data flow sets to construct the DDS. The DDS may contain three types of data dependence edges, expressing flow-, anti- and output-dependence, depending on the application. Flow-dependence edges are added to the CDS from nodes containing definitions of a variable to nodes containing reachable uses of that variable. Anti-dependence edges connect nodes containing uses of a variable to definitions of that variable that follow. Output-dependence edges connect definitions of the same variable. We can develop both forward and backward data flow analysis algorithms that use the CDS. Here, we present our algorithm to compute reaching definitions, which is used to compute flow dependence information for the DDS.

To compute sets of definitions that reach each statement in the program, we first compute local definition information, and then we propagate it throughout the program using the implicit and explicit control flow edges in our CDS until the sets stabilize. The number of iterations required depends on the loop nesting in the program. The algorithm ReachingDefs is given in Figure 3.4. ReachingDefs assumes that local data flow information, consisting of the usual GEN and KILL sets, has been computed and is attached to CDS nodes. The GEN set consists of the definition, if any, in that statement (node); the KILL set consists of all other definitions of the GEN set's variable in the program. The GEN set is easily computed as the CDS is built. Using the GEN sets, KILL sets are computed for each statement node. Predicate nodes and region nodes have neither GEN nor KILL sets.

To propagate data flow information, we use IN and OUT sets where required. The IN set of a node consists of those definitions that reach the point immediately before the statement; the OUT set consists of those definitions that reach the point immediately after the statement. Since IN and OUT sets for a statement node may differ, we require both of these at each statement node in the CDS. However, since region and predicate nodes have identical IN and OUT sets, we use the OUT set to represent both of them.

On each iteration, algorithm ReachingDef s considers each node N in the CDS and computes it's IN and OUT sets using either of two sets of equations depending on the node's type. If N is a region or predicate node, OUT [N] is computed as the union of the OUT sets of its control flow predecessors. If N is a statement node, IN [N] is the union of the OUT sets of its control flow predecessors and OUT [N] is synthesized using IN [N], GEN [N] and KILL [N].

```
Procedure ReachingDefs
Input       control dependence sub-graph (CDS), with control flow edges (CF),
            GEN / KILL sets computed for each statement node
Output      reaching definitions at each node
Declare     IN sets for each statement node OUT sets for each node

 Begin
    While changes do
       Foreach node N in CDS do
          If N is a region/predicate node then OUT [N] = U OUT [P], P is a control
             flow predecessor of N
          Else
             IN [N] = U OUT [P],    P is a control flow predecessor of N
             OUT [N] = GEN [N] U (IN [N] - KILL [N])
           Endif
       Endfor
 Endwhile

 End ReachingDefs
```

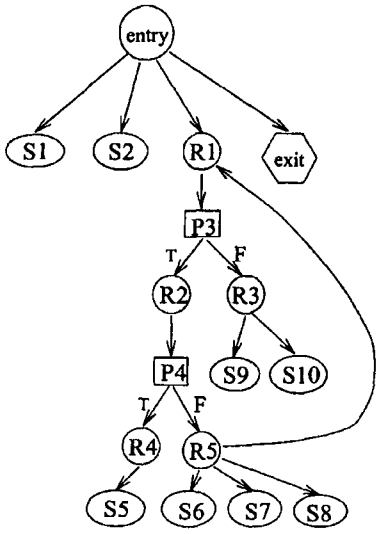
Figure 3.4. Algorithm to compute reaching definitions using the CDS and control flow edges.

3.4 Security Regression Test Selection

Our technique for security regression test selection has several advantages. Unlike many other previous techniques, our algorithm is specifically tailored for security regression testing. It selects tests that may now traverse new statements, and tests that formerly traversed statements that have been deleted from the original program. Thus, our algorithm selects every test from the original test suite that can possibly expose errors in the modified program. Moreover, it is more precise than most existing algorithms since it does not select tests that cannot traverse changed statements. Our algorithm is also simpler and more efficient than most existing algorithms, in part because it does not require a complete mapping of the "corresponding parts" of the original and modified programs. The algorithm is not difficult to implement, and thus allows automation of a substantial portion of the selective retest process. Finally, our algorithm is more general than many previous approaches: it is not restricted to a subset of language constructs, or limited to particular types of program modifications. The algorithm is easily extended to facilitate selective retest at the integration and system levels where its savings are even more spectacular.

3.4.1 Background

For statements X and Y in a program, if X is control dependent on Y then there must be at least two paths out of Y, where one path always causes X to be executed and the other path may result in X not being executed. A statement X may be control dependent on several statements in the program, but we can always identify immediate control dependencies for X. For example, in procedure MedRec in Figure 3.5, statement S7 is control dependent on predicates P4 and P3, but immediately control dependent only on P4.




```
Procedure MedRec
S1. Read(user, password)
S2. find permission where user = usr and
                        password = pwd
no-lock.
P3. for each medrec :
P4. If
medrec.perm_level<=usr_perm_level
S5.     Browse_medrec:add(medrec).
    Else
S6.     Msg(full access denied, general
info only)
S7.     Browse:add(partial_medrec)
    Endif
S8.     Refresh Browse_medrec
    End_For_each
S9.  msg(read complete!)
S10. release(user, password)
```

Figure 3.5: Procedure MedRec and its CDG.

Our way to encode control dependence information is with a control dependence graph (CDG) [18]. The CDG for procedure MedRec is given in Figure 3.5. A CDG contains several types of nodes. Statement nodes, shown as ellipses, represent simple statements. Predicate nodes, drawn as squares, stand for conditional statements, and have one or two labeled exiting edges that represent possible control paths. Region nodes, represented by circles, summarize control dependencies for statements in the region; the entry node, labeled entry, can be thought of as a region. An exit node represents the program's exit.

A CDG represents control dependencies explicitly. In Figure 3.5 directed edges denote immediate control dependencies, and the hierarchical structure of the CDG encodes control dependencies generally. For example, it is clear from the CDG that statement node S7 is control dependent on both P3-true and P4-false, but immediately control dependent on P4-false. Figure 3.5 also illustrates the use of region nodes. For example, nodes S6, S7, S8 and R1 are control dependent on P4-false; without region node R5, there would be four edges labeled "F" from node P4. Thus, R5 summarizes control dependence on P4-false.

The CDG of Figure 3.5 also illustrates two interesting control dependence relationships in A1 that occur due to the presence of the exit from the loop body in statement S5. First, the while loop headed by region R1 is control dependent on predicate P4, which controls execution of S5. This creates a cycle in the control dependence graph (R1, P3, R2, P4, R5, R1). Second, statements S9 and 510, which follow the while loop, are control dependent on P3 because an execution that traverses P3-False reaches S9 and 510, while an execution that takes P3-True may not reach them.

3.4.2 Issues in Security Regression Testing

Given a secure 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'.

Typical solutions to this problem consist of the following steps:

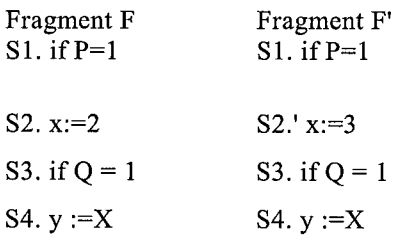
1. Identify the modifications made to P, and obtain a mapping between code segments in P and P'.
2. Using the results of step 1, select T' C_ T, a set of tests that may reveal modification-related errors in P'.
3. Run T' on P, establishing P's correctness with respect to T'.
4. If necessary, create new tests for P. These may include new functional tests required by changes in specifications, and/or new structural tests required by applicable coverage criteria.
5. Create T", a new test set-history for P.

Our primary concern in this work is security selective retest, so when we focus on steps 1 through for a program P, its modified version P, and a set of tests T for P, we require that a selective retest algorithm meet the following criteria: Precision, efficiency, and generality. Note that the retest-all approach is safe but is also imprecise.

3.4.3 Observations

Our goal is to distinguish potentially revealing tests in a modified program from those that cannot exhibit altered behavior. Toward this goal we offer the following discussion.

In order for a particular test Ti, run originally on program P, to exhibit different behavior when run on program P', the sequence of statements that T_i traverses in P must differ from the sequence it traversed in P. If test Ti, run on P', does not traverse any modified statements, does not traverse any new statements, does not miss any statements it traversed in P, and traverses all statements in the same order as it did on P, it cannot behave any differently in P' than it did in P.



test #	input	Execution history
T1	P=1,Q=1	S1,S2,S3,S4
T2	P=0,Q=1	S1,S3,S4

Figure 3.6. Modified versus affected statements.

For example, Figure 3.6 shows program fragments F and F', and two tests used on F. Statement S2 has been modified, yielding S2'. Of the two test cases, only T1 traverses S2'. T2 does not traverse S2' and thus, in the absence of other changes, cannot possibly exhibit different behavior, so it does not need to be rerun. A first task in secure selection of tests for retest is to distinguish tests that traverse changed code from those that do not. This statement may seem obvious, yet it has been ignored in

previous techniques [5, 7]. These techniques suggest selection of all tests that traverse affected statements in P'; affected statements are program statements that make use of the results of, or are otherwise affected by, some code change. Affected code is quite different from changed code since code can be affected without having been modified, deleted, or added. It is possible for a test to traverse an affected statement without traversing any changed code. Such tests cannot exhibit differences in program behavior, and need not be rerun. For example, in Figure 3.6, S4 is an affected point, because the definition in modified statement S2' reaches S4. However, test case T2, which traverses S4, does not traverse any modified points, and cannot possibly exhibit different behavior.

Note that changes in a program are reflected by changes in its CDG. Deletion, addition, or modification of a statement in P results in the addition, deletion, or modification of a node in P's CDG. Complex changes to P, of course, result in substantial differences in P's CDG. Also note that we can attach to each CDG region a list of tests known to enter that region, and we can keep test execution histories that list these regions.

We now present the fundamental theorem on which our technique is based:

Given the CDG of program P, the CDG' of program P', and test suite T in which test execution histories list the regions in P traversed by each test; the only tests in T that can traverse different sequences of statements in P and P are those attached to some region node R in P, such that R has a corresponding region node R' in P', and $R's$ immediate children have been changed.

It follows from the theorem that a simple approach to test selection is to select all tests that enter the parent region of any changed node. It is these tests, and only these tests, that may exhibit changes in program behavior. The problem with this simple approach is that it requires us to obtain a mapping of regions in CDG to regions in CDG', so that we compare all corresponding regions.

Such a mapping is difficult to obtain in the presence of complex or multiple code changes. However, by traversing nodes in CDG properly we avoid this problem, and obtain an opportunity for optimization as well. Observe that once we have detected a difference among the children of region R in P, and selected the tests attached to R, we need not consider any of R's control dependence successors in CDG (unless they can be reached from some other region not control dependent on R). For example, if statement S6 in procedure A1 (Figure 7) is modified, and the condition in predicate P4 is changed to "n > 0", then we want to rerun tests attached to R5 and R2. However, the only tests that may reach R5 are those that reached R2. Therefore, once we have examined R2 and selected its tests, we don't need to proceed further down in CDG. The tests attached to R5 contributed along this chain of control dependencies have already been selected.

So, an efficient test selection algorithm need not locate all regions immediately enclosing changes; it need only search the CDG until some region R enclosing changes is found, and return the tests attached to R. In doing so, the algorithm automatically selects tests attached to regions control dependent on R. By traversing the CDG in this fashion, we obviate the need for complete information on code changes, or for complete information on the corresponding sections of code in two program versions. We need only check, at any node reached in the CDG, whether the children of that node in the new program differ from the children of the corresponding node in the old program. If so, we need not worry about identifying nested changes.

3.4.3 Our Algorithm for Secure Regression Testing

Our algorithm SelectTests, given in Figure 3.7, 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 3.7: SelectTests algorithm

3.4.3.1 Example of Test Selection Using SelectTests

Let's consider procedure MedRec' a shown in figure 3.8, a changed version of procedure MedRec. In MedRec', statement S7 has mistakenly been deleted, and statement S5a has been added. When called with MedRec and MedRec', SelectTests constructs the CDG's for MedRec and MedRec', and calls procedure compare with entry and entry'. Procedure Compare finds the children of these nodes equivalent, and invokes itself (invocation 2) on R1 and R1'. Recursive calls continue in this manner on nodes P3 and P3' (invocation 3), R2 and R2' (invocation 4), and P4 and P4' (invocation 5). In each case the children of the nodes are found equivalent.

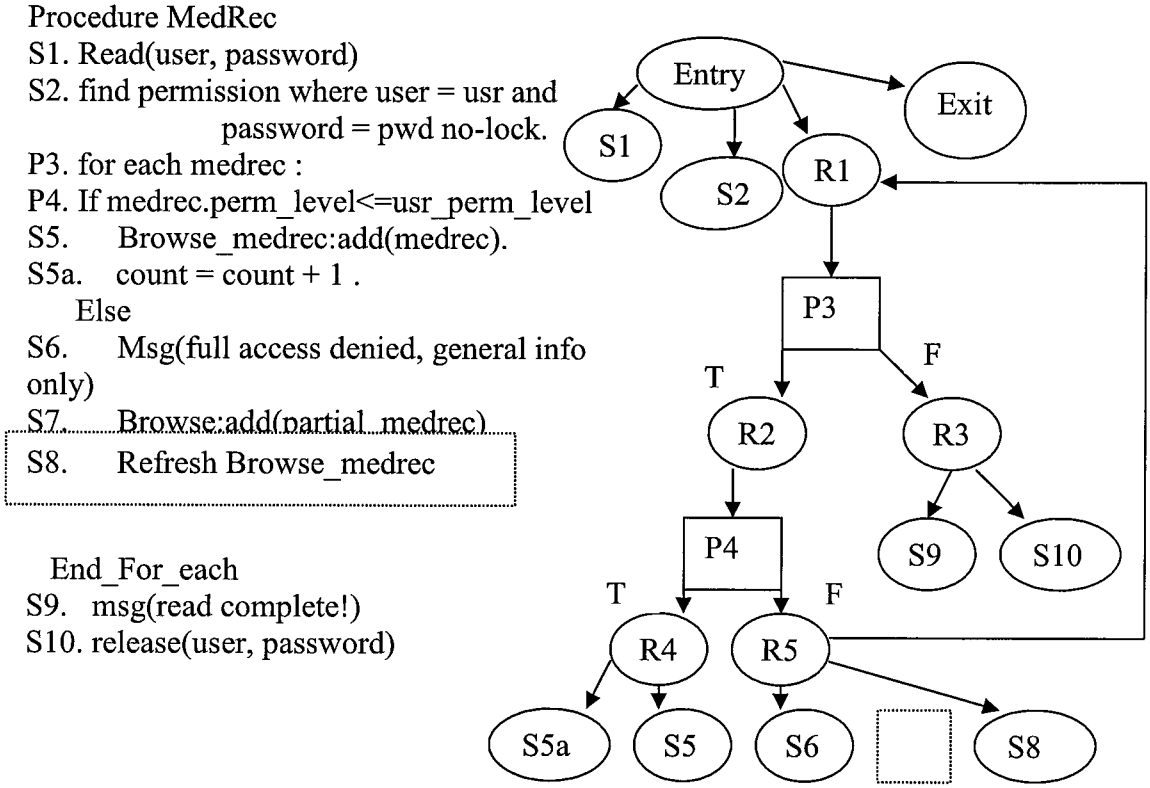


Figure 3.8. Procedure A2 and its CDG.

In invocation 6 (on R4 and R4'), procedure Compare discovers nonequivalent children, and thus returns test T2, the only test attached to R4, to invocation 5. Next, Compare calls itself with R5 and R5'. Compare discovers nonequivalent children again, and returns T3, the only test attached to R5. (Because R1 has already been visited, Compare does not examine it again.) Returning up the tree, {T2, T3} is passed back to invocation 4, and then to invocation 3. Here, Compare calls itself with R3 and R3', finds no differences, and returns a null set. Invocation 3 passes {T2, T3} up the tree to invocation 2, then to invocation 1, and finally to the main procedure. The resulting test set, {T2, T3}, contains all the tests that could possibly exhibit different behavior in A2. If the deletion of S7 had been the only change, only {T3} would have been returned. Had the addition of S5a been the only change, only {T2} would have been returned. Most other methods [1, 19, 21, 28, 29, 32, 37, 41] fail to identify T2 and/or T3 as necessary.

To see how SelectTests handles predicate changes, imagine what happens if line P4 in procedure A1 is also changed (erroneously) to "n>0". This change alters only the text associated with node P4 in program A2's CDG. Called with procedures A1 and A2, SelectTests proceeds as in the previous example until it reaches R2 and R2'. Here it finds non equivalent children, and returns {T2, T3}. Note that in this case, no analysis is needed on nodes under P4. No other methods for test case selection make use of the opportunities afforded by the nesting of control dependencies to reduce analysis in this fashion.

Procedure Compare in SelectTests requires a method for determining when the children of two CDG nodes differ. A simple algorithm for doing this checks corresponding nodes for identical text contents.

This simple algorithm is inexpensive and easy to implement, but can be imprecise. Consider, for example, the program fragments shown in Figure 3.7, in which two unrelated assignment statements, S1 and S2, are swapped. The simple algorithm considers statement order significant, so it finds the children of entry and entry' different, and returns all the tests attached to entry. An algorithm that distinguishes between semantic and syntactic changes would note that the behavior of the nodes under entry and entry' is not, in fact, different, and would continue searching the CDG's for real differences.

3.4.3.2 Enhancements and Contribution

This simple algorithm is inexpensive and easy to implement, but can be imprecise. Consider, for example, a secure program fragment, in which two unrelated assignment statements, S1 and S2, are swapped. The simple algorithm considers statement order significant, so it finds the children of entry and entry' different, and returns all the tests attached to entry. So, an algorithm that distinguishes between semantic and syntactic changes would note that the behavior of the nodes under entry and entry' is not, in fact, different, and would continue searching the CDG's for real differences.

On the other hand, any algorithm that is useful in this matter will increase the precision of the approach for an exchange with computational cost.

We observe that when the backward slices of two PDG nodes are equal then the statements are semantically equivalent [35]. So, in addition to comparing ordered CDG nodes for textual equivalence, we should compare nodes that are not textually equivalent with backward slicing. Note that the use of backward slicing for nodes is only applicable for "statement nodes" within same regions in both CDS's, since any swapping of positions of "statement nodes" that removes the node from outside its original parent region will result a change in the structure of CDS. Hence, backward slicing is not applicable.

It is necessary to be able to determine when different secure program statements have equivalent behaviors. Given program points p1 and p2 in programs P1 and P2, respectively, we say that p1 and p2 have equivalent behavior if for every initial state on which both P1 and P2 terminate normally; p1 and p2 produce the same sequence of values [5, 43].

Furthermore we use a table or a hash table to store slices of each node that is a child of node N in P and N' in P' who aren't identical and then we attempt to match the slices of children of node N' in P'. We can further improve the hash table approach. We calculate and stores complete slices on each non-identical node in the PDG, we reduce slice calculation by summarizing the slices computed for nodes higher in the hierarchy, and using these summaries in subsequent slice computations. For example, given a PDG, when we encounter and attempt to slice back on a node Si, slices of nodes Si-n have been previously computed and found equivalent. We need slice no farther back from these nodes

3.4.3.2.1 Program Slicing

In this section we discuss three slicing problems:
S.1. For a given program point p in program P, find [a superset of] the points q of P such that if the behavior at q were different then the behavior at p would be different.
S.2. For a given program point p in program P, find [a superset of] the points q of P such that if the behavior at p were different then the behavior at q would be different.5

In other words, problem S.1 asks for the set of points that might affect the behavior at p, while problem S.2 asks for the set of points that might be affected by p.
S.3. For a given program point p in program P, find a projection Q of P such that when P and Q are run on the same initial state, the behaviors at p in P and Q are identical.

Problems S.1 and S.3 are closely related. Often, a solution to one provides a solution to the other . We refer to the two problems as backward slicing. Problem S.2 is referred to as forward slicing.

A simple algorithm that computes backward slices can help identifying true diffrerences between two secure programs, permitting to focus attention on meaningful changes of the program.

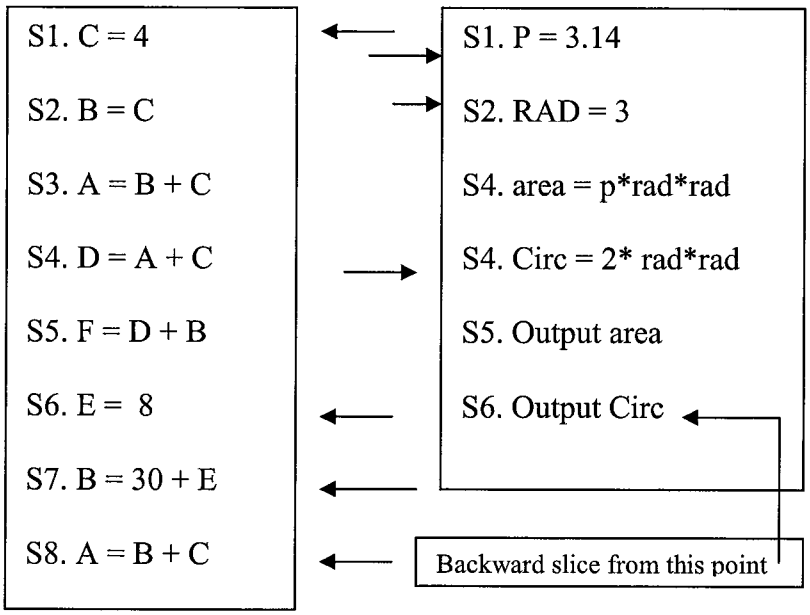


Figure 3.9. showing 2 examples of bakward slicing.

Backward slices in figure 3.9 are : S8-S7-S6-S1, and S6-S4-S2-S1 respecitvely.

Chapter 4

Support System

We have implemented a security regression testing tool as a support system for this thesis. 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.

The system tool is made for Progress datatbase applications programmed using the Progress language.

Our security regression testing tool is composed of six parts:

1. Trusted application parsing.
2. Trusted code analysis.
3. CDS construction.
4. Trusted regression testing selection.
5. Backward slicing.
6. Test case reduction.

In the following, we will discuss the functionality implemented in each part of the tool. Note that step one to step three is applied for both original trused application and its modified version. Both parsing and analysis are necessary steps primary and essential to CDS construction, which will be the basis of our test selection algorithm.

4.1) Trusted Application Parsing

A parser is a program that dissects source code so that it can be translated into object code. In the parsing phase, we divide trusted code into small components that can be analyzed. For example, parsing a procedure would involve dividing it into statements, predicates, etc. and identifying the type of each component. Parsing is an essential part of our strategy discipline. For example, compilers must parse source code to be able to translate it into object code. Likewise, our application tool processes complex commands. This includes eventually all trusted applications.

Parsing is often divided into lexical analysis and semantic parsing. Lexical analysis concentrates on dividing strings into components, called tokens, based on punctuation and other keys. Semantic parsing then attempts to determine the meaning of the string.

Initially, our tool accepts the full path and name of both trusted program and its modified version to be parsed, and the result will be the basis of work in the subsequent steps.

4.2) Trusted Code Analysis

Trusted code analysis involves building syntax trees for the parsed code, removing all non significant code like remarks, useless code, irreleavant definitions etc, and then use these syntax trees to gather control flow information for use in CDS construction.

Performing all these tasks requires a mechanism of storing the result in one step using a lexical analyser, and passing them to succeeding steps.

4.3) CDS Construction

In this section, we use our algorithm for constructing the control dependence subgraph (CDS). The tool accepts an abstract syntax tree (AST) for a procedure P and outputs the CDS for P. For our tool, the language contains the following type of statements: simple statements (assignment), structured (if-else, while, for-each, etc.), structured transfers of control (continue, break, return no-apply, quit, exit, etc.), and may contain unstructured transfers. Other constructs and statements are handled similarly. The tool uses a left to right preorder traversal of P's AST, and takes appropriate actions as each node is encountered. Our tool uses a stack, and the usual use of stack operations, to maintain nesting, and therefore control dependencies.

In most cases, our tool encodes control flow implicitly through node order. By ordering nodes, our tool preserves the implicit control flow. The reslt of CDS construction will be used subsequently for the implementation of the trusted regression test selection.

4.4) Trusted Regression Test Selection

At this stage, the CDS of the secure application and its modified version are ready, and the entry nodes of each one of them is used as input for selection test phase which will run recursively. The tool works as follows: given any two CDG nodes N and N', the tool marks these nodes as "visited", and then determines whether the children of these nodes are semantically and syntactally identical.

4.5) Backward Slicing

In addition to comparing ordered CDG nodes for textual equivalence, the tool uses backward slicing to compare nodes that are not textually equivalent, and stores the slices of both children nodes in a table. It then attempts to match the slices of both nodes.

The tool improves the table approach by summarizing the slices computed for nodes higher in the hierarchy, and using these slices in the subsequent slice computations.

4.6) Tool Interface

The information gathered in the previous steps is displayed. The tester can navigate into the CDS trees of both main and modified trusted application. While navigating through the module statements, the information gathered on each statement is displayed. In figure 4.1 we give the screen that gives the full information.

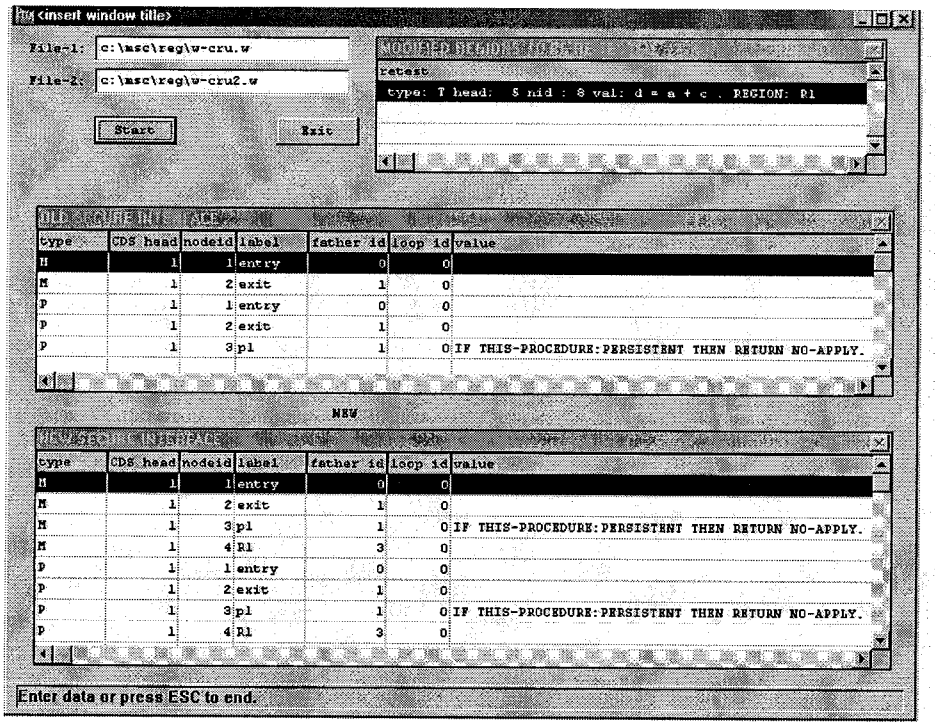


Figure 4.1 The tool's information display screen.

4.7) Modification Detection

The affected regions due to modification are determined and displayed in the tool's interface. It gives the tester specific details about the region to be re-tested and which part of the region is affected directly or inderctly in the modification process. Figure 4.1 displays the browser that contains the result of selection strategy.

4.8) Conclusion

The support system is an effective way for supporting testing efforts. It tackles the problem of specification modifications and code modifications for trusted applications. Also, The tool has an iterface that maps the original trused application and its modified version to CDG and CDG' respectively, and represents the two graphs separately in two different browsers. Furthermore, the tool detects automatically new changes in the modified trusted application, and displays the affected node, the region to be retested, and the contents of the modified node.

Chapter 5

Empirical Results

To empirically investigate the use of our regression testing methodology, we have performed a study on a prototype of trusted application. In this chapter, we describe this study and discuss its results.

5.1) Experimental Design

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 (figure 5.1), 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 is 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 is 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 is 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	FALSE

Figure 5.1 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:
-Grantor is not the grantee
-All possible combinations of the GranteeType (user, role, public)
-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:

1- the grantor_owns_object relationship that is associated with GOP(A) where the grantor owns the object OR.
2- granted_obj_priv and grantee constraints that is associated with GOP(B).

The false case establishes the conditions where the grant operation fails:

1- grantor is not the object owner.

2- grantor has not been granted object privelege.

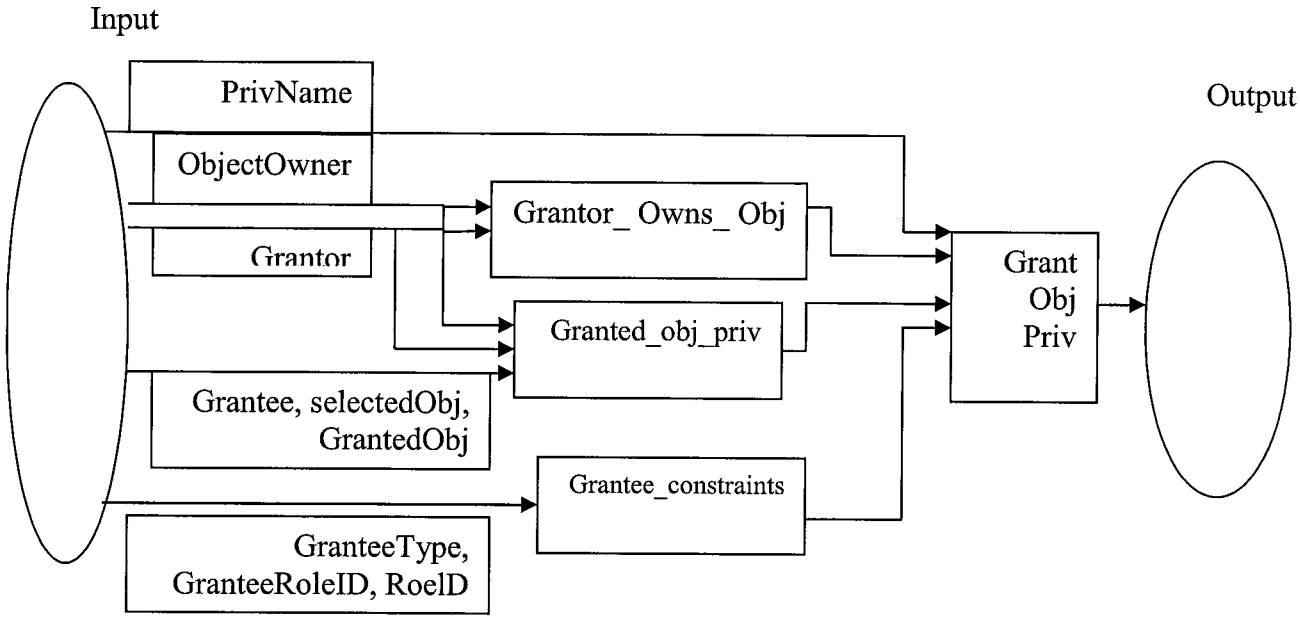


Figure 5.2 Grant-revoke trusted model application.

5.2) Results

To aquire and analyse empirical results, the tool was used on the grant-revoke trusted application and its modified versions. Figure 5.3 represents the CDG table of the initial secure application grant-revoke.

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 = " 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 "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 the
44	R11	43	0	
45	s32	44	0	GRANTt = TRUE .
46	R12	43	0	
47	s33	46	0	Grantt = false .

Figure 5.3 CDG table for initial grant-revoke secure application

The test history of the grant-revoke secure application is divided into groups of tests, each represent 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 5.4.

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 5.4. Original test suite table.

5.2.1 Modification Cases

In this section, we discuss each modification case alone (Modification include insertion, deletion, addition, swapping, etc). We give the directly and indirectly affected regions, and the test classe to be re-run. To get these results, we have to analyse the modified version of the trused application of each modification case. This task requires the same time required by the tool to analyse the initial which is around 3 seconds, in addition to the time required to identify modified regions, which is around 4 seconds, for each modification case.

Case1:

Modification: delete statement s17.

Directly affected regions: all.

Test classes traversing all affected regions: T1, T2, T3, T4,….,T9.

Percentage of test reduction: 0%.

Case2:

Modification: modify statement s17.

Directly affected regions: all.

Test classes traversing all affected regions: T1, T2, T3, T4,….,T9.

Percentage of test reduction: 0%.

Case3

Modification: modify statement s3.

Directly affected regions: R12.

Indirectly affected regions: none.

Test classes traversing all affected regions: T9.

Percentage of test reduction: 89%.

Case4

Modification: add a new statement s right after statement s3 .

Directly affected regions: R12.

Indirectly affected regions: none.

Test classes traversing all affected regions: T9.

Percentage of test reduction: 89%.

Case5

Modification: modify statement s3.

Directly affected regions: R12.

Indirectly affected regions: none.

Test classes traversing all affected regions: T9.

Percentage of test reduction: 89%.

Case6

Modification: randomly swapp places between statements s17,s18,...s22.

Directly affected regions: none.

Indirectly affected regions: none.

Test classes traversing all affected regions: none.

Percentage of test reduction: 100%.

Case7

Modification: delete statement s8.

Directly affected regions: R2.

Indirectly affected regions: R3, R4,R5.

Test classes traversing all affected regions: T2, T3.

Percentage of test reduction: 78%.

Case8

Modification: delete or modify s12.

Directly affected regions: R4.

Indirectly affected regions: R2, R4, R5.

Test classes traversing all affected regions: T3.

Percentage of test reduction: 89%.

Case9

Modification: add new statement s after statement s10.

Directly affected regions: R3.

Indirectly affected regions: R2.

Test classes traversing all affected regions: T2.

Percentage of test reduction: 89%.

Case10

Modification: delete or modify s14.

Directly affected regions: R5.

Indirectly affected regions: R2, R4.

Test classes traversing all affected regions: T3, T4.

Percentage of test reduction: 80%.

Case11

Modification: move statement S2 and put it after statement S10.

Directly affected regions: R1, R3.

Indirectly affected regions: R2.

Test classes traversing all affected regions: T1, T2.

Percentage of test reduction: 89%.

Case12

Modification: move s3 after P7.

Directly affected regions: All.

Indirectly affected regions: All.

Test classes traversing all affected regions: T1, T2, T3,….,T9.

Percentage of test reduction: 0%.

Case13

Modification: move statement S5 to the beginning of all code.

Directly affected regions: none.

Indirectly affected regions: none.

Test classes traversing all affected regions: none.

Percentage of test reduction: 100%.

Case14

Modification: add statement after statement S26 or modify statement S26.

Directly affected regions: R8.

Indirectly affected regions: R7.

Test classes traversing all affected regions: T5.

Percentage of test reduction: 89%.

Case15

Modification: delete statement S30.

Directly affected regions: R10.

Indirectly affected regions: R7.

Test classes traversing all affected regions: T7.

Percentage of test reduction: 89%.

Case16

Modification: add statement after statement S14 and modify statement S10 and S30.

Directly affected regions: R3, R5, R10.

Indirectly affected regions: R4, R2, R7.

Test classes traversing all affected regions: T2, T3, T7.

Percentage of test reduction: 66%.

Case17

Modification: modify predicate P25.

Directly affected regions: R7.

Indirectly affected regions: R8, R9, R10.

Test classes traversing all affected regions: T5, T6, T7.

Percentage of test reduction: 66%.

Case18

Modification: delete predicate P25.

Directly affected regions: R7.

Indirectly affected regions: none.

Test classes traversing all affected regions: T6, T7.

Percentage of test reduction: 78%.

Case19

Modification: modify statement S14, S12, P13.

Directly affected regions: R4, R5.

Indirectly affected regions: R2.

Test classes traversing all affected regions: T3.

Percentage of test reduction: 89%.

Case20

Modification: delete statement S26, S28, S30 , and modify P27, P28, P29.

Directly affected regions: R7, R8, R9, R10.

Indirectly affected regions: none.

Test classes traversing all affected regions: T5, T6, T7.

Percentage of test reduction: 66%.

5.2.2 Summary of Results

In figure 5.5, we present a summary of test cases presented in this section. 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.

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

Figure 5.5. Summary of Results. (Total Test cases = 40)

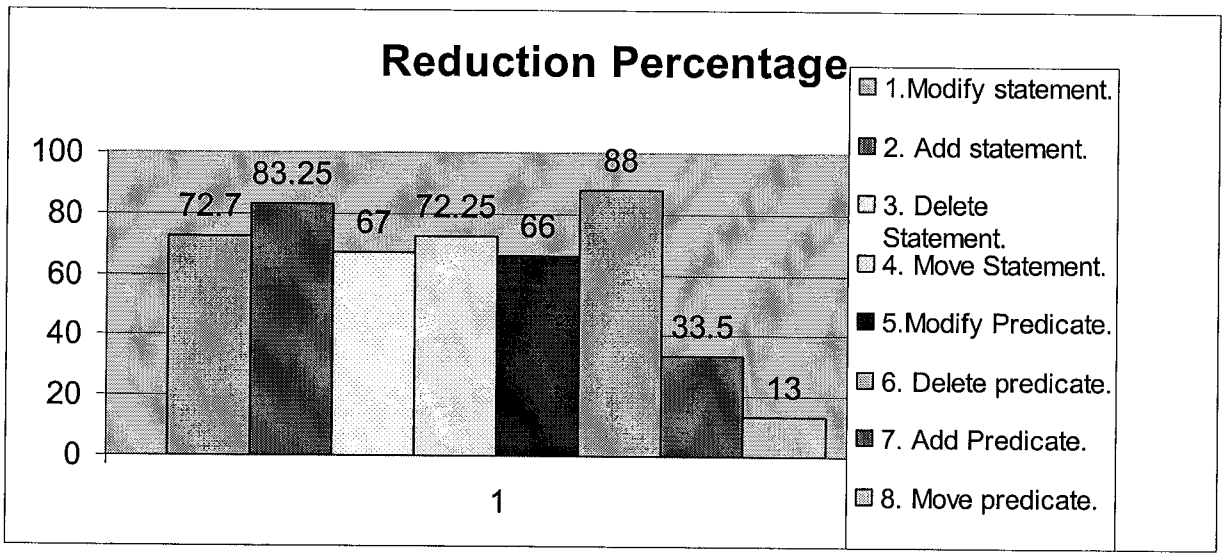


Figure 5.6 Percentage of test case reduction.

5.2.3 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.

Chapter 6

Conclusion and Future Work

6.1) Summary

Throughout this thesis, we demonstrated the need for a new systematic methodogy that can be applied in practive to trusted application to handle both requirement and code modifications. We achieved this by detecting code design, code changes, and deriving the selected tests based on control dependence graph and regression testing. The problem of security regression testing in practice is defined. Existing regression testing methodologies are discussed and criticized. A new technique for trusted application regression testing is presented. This technique is safe and efficient.

Chapters1 and 2 provide the reader with definitions and background information related to regression testing. The problems of regression testing were identified and related work was discussed. Chapter 3 described the details of the new methodology and our enhancements. Our new strategy provides a new objective in regression testing of trusted applications, which has good potential to guide trusted code test selection. In chapter 4, an implementation of our new strategy was discussed. The resulting tool performs regression testing on trusted code, and its modified version, and displays result in a simple and readable way even for less experienced test personnel. In chapter 5, evaluations and discussions were presented based on the case studies. The results of the case studies indicated that this technique was cost-effictive, safe, and efficient in finding only real changes and defects.

In our thesis, we presented a new methodology for re-testing trusted applications, because it is safe unlike previously mentioned methods, since it selects every test than can possibly exhibit different behaviour in a modified trusted code, including tests that cover new or deleted code, and that what it makes it specially tailored for trusted applications.

Also, it is more precise than existing algorithms aimed at achieving safe solutions, because it does not base its selection on whether code is affected, looking instead for changed code. Our algorithm is faster and more space efficient than other algorithms, and does not require information on code modifications.

6.2) Contributions of the Thesis

The contributions of this thesis can be summarized as follows:

- Demonstrate that trusted application regression testing is essential for conducting and managing trusted code regression analysis.
- Provide a new technique for trusted code selective retest that is neither coverage criteria based nor requires complete information on corresponding program components, and can be implemented in a test tool. In addition, it is directed toward finding modifications in trusted applications.
- Introduce a new strategy of combinging select-test algorithm and backward slicing that work on both intra-procedural level and iter-procedural level. It provides a clean and useful focus on the real modifications of trusted code.
- Develop and illustrate the use of our newly developped tool, and show that our strategy can be used to produce quality regression testing suites.

6.3) Future Work

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. We are looking into utilizing statistical analysis tool such as SPSS to aid in determining the effectiveness of our technique in practice.

In below, we briefly expand on some of these issues:

- Implement the technique in production test environment: compared to computer processing time, the time of software engineers is much more costly. Hence, automation is essential to the usability of security regression testing methodology. Our tool for trusted application regression testing can be implemented in a production re-test environment.

- Use statistical analysis tools to determine the effectiveness of our strategy in practice: when we evaluated our technique, we found that the future use of a statistical analysis tool can be powerful in measuring test selections, and improvements on test selections.

From our experience, incomplete and out-of date documentation exists throughout project development and always causes big problems. Because requirement capturing and system design are often done in an informal way, requirement and design documentation is written manually. That causes more serious problems in trusted application re-testing since the document is error-prone. As a part of testing, documentation testing is not done efficiently. Formal regression testing methodologies still have practical problems and not tailored specifically for trusted applications. This thesis has opened new research topics related to secure application regression testing.

References

[1] A.B. Taha, S.M. Thebaut, and S.S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," Proceedings of the 13th Annual International Computer Software and Applications Conference, pp. 527-34, September, 1989.

[2] B. Korel, "The program dependence graph in static program testing," Information Processing Letters, vol. 24, pp. 103-108, January 1987.

[3] B. Sherlund and B. Korel, "Modification oriented software testing," Conference Proceedings: Quality Week 1991, pp. 1-17, 1991.

[4] Benedusi, A. Cimitile, and U. De Carlini. Postmaintenance testing based on path change analysis. In Proceedings of the Conference on Software Maintenance - 1988, pages 352-61, October 1988.

[5] Binkley, "Using semantic differencing to reduce the cost of regression testing," Proceedings of the Conference on Software Maintenance '92, pp. 41-50, November 1992.

[6] E. Duesterwald, R. Gupta and M. L. Soffa, "Rigorous data flow testing through output influences," Proceedings of the 2nd Irvine Software Symposium (ISS'92), pp. 131-145, March 1992.

[7] E. Schatz and B. G. Ryder, "Directed tracing to detect race conditions," LCSR-TR-176, Laboratory for Computer Science Research, Rutgers University, February 1992.

[8] E.F. Miller. Exploitation of software test technology. In Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA), page 159, June 1993.

[9] G. Rothermel and M.J. Harrold, "A comparison of regression test selection techniques" Technical Report 114, Clemson University, Clemson, SC, April 1993.

[10] G. Rothermel and M.J. Harrold, "A safe, efficient regression test selection technique" ACM transactions on software engeneering and methodology, Vol. 6, No. 2 , April 1997.

[11] Gupta and M. L. Soffa, "Automatic generation of a compact test suite," Proceedings of the Twelfth IFIP.

[12] H. Agrawal and J. Horgan, "Dynamic program slicing," Proceedings of ACM SIGPLAN '90 Symposium on Programming Language Design and Implementation, pp. 246-256, June 1990.

[13] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental Regression Testing. In Proceedings of the Conference on Software Maintenance - 1993, pages 348-357, September 1993.

[14] H. Agrawal, R. DeMillo and E. Spafford, "Dynamic slicing in the presence of unconstrained pointers," Proceedings of the Symposium on Testing, Analysis and Verification, pp.60-73, October 1991.

[15] H. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations in C programs. In Proceedings of the Fourth ACM Symposium on Testing, Analysis and Verification (TAV4), October 1991.

[16] H.K.N. Leung and L.J. White, "A cost model to compare regression test strategies," Proceedings of the Conference on Software Maintenance, 1991, pp. 201-8, October, 1991.

[17] H.K.N. Leung and L.J. White, "A study of integration testing and software regression at the integration level." Proceedings of the Conference on Software Maintenance, 1990, pp. 290-300, November, 1990.

[18] J Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization," ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319-349, July 1987.

[19] J. Hartmann and D.J. Robson, "Techniques for selective revalidation," IEEE Software, Vol. 16(1), pp. 31-8, January, 1990.

[20] J. Laski and W. Szemer. Identification of program modifications and its applications in software maintenance. In proceedings of the conference on software maintenance – 1992, apges 282-90, November 1992.

[21] K.F. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," Proceedings of the National Telecommunications Conference, Vol. B-6-3, pp. 1-6, November, 1981.

[22] L.J. White and H.K.N. Leung, "A firewall concept for both control-flow and data-flow in regression integration testing," Proceedings of the Conference on Software Maintenance, 1992, pp. 262-70, November, 1992.

[23] M. Davis and E. Weyuker. Computability, Complexity, and Languages. Academic Press, Boston, MA, 1993.

[24] M.J. Harrold and B. A. Malloy, "A unified interprocedural program representation for a maintenance environment," IEEE Transactions on Software Engineering, to appear.

[25] M.J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," Proceedings of the Conference on Software Maintenance '92, pp. 272281, November 1992.

[26] M.J. Harrold and B. A. Malloy, "Performing data flow analysis on the PDG", Technical Report 92-108, Clemson University, March 1992.

[27] M.J. Harrold, B. A. Malloy and G. Rothermel, "Efficient construction of program dependence graphs," Technical Report 92-128 Clemson University, December 1992.

[28] M.J. Harrold and M.L. Soffa, "An incremental approach to unit testing during maintenance," Proceedings of the Conference on Software Maintenance, 1988, pp. 362-7, October, 1988.

[29] M.J. Harrold and M.L. Soffa, "Interprocedural data flow testing," Proceedings of the Third Testing, Analysis and Verification Symposium, pp. 158-67, December, 1989.

[30] M.J. Harrold, B.A. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," Proceedings of the International Symposium on Software Testing and Analysis 93 (ISSTA93), pp. 160-70, June, 1993.

[31] M. Weiser, "Program slicing," IEEE Transactions on Software Engineering, vol. 5E-10, no. 4, pp. 352-357, July 1884.

[32] P. Benedusi, A. Cimitile, and U. De Carlini, "Postmaintenance testing based on path change analysis," Proceedings of the Conference on Software Maintenance, 1988, pp. 352-61, October, 1988.

[33] R. Ballance and B. Maccabe, "Program dependence graphs for the rest of us," Technical Report, University of New Mexico, November 1992.

[34] R. Cytron, J. Ferrante, B. Rosen and M. Wegman, "Efficiently computing static single assignment form and the control dependence graph," ACM Transactions on Programming Languages and Systems, vol. 13, no. 4, pp. 451-490, October 1991.

[35] R. Gupta, M. J. Harrold and M. L. Soffa, "An approach to regression testing using slicing", Proceedings of the Conference on Software Maintenance '92,pp. 299-308, November 1992.

[36] R. M. Stallman, "Using and porting GNU CC," Free Software Foundation, Inc., Cambridge MA, pp. 73-77, February 1990.

[37] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," Annual ACM Symposium on Principles of Programming Languages, January, 1993.

[38] S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and Systems, v. 12, no. 1, pp. 26-60, January 1990.

[39] Schach, Software Engineering, Aksen Associates, Boston, MA, 1990.

[40] S.S. Yau and Z. Kishimoto, "A method for revalidating modified programs in the maintenance phase," COMPSAC '87: the Eleventh Annual International Computer Software and Applications Conference, pp. 272-7, October, 1987.

[41] T.J. Ostrand and E.J. Weyuker, "Using dataflow analysis for regression testing," Sixth Annual Pacific Northwest Software Quality Conference, pp. 233-47, September, 1988.

[42] U. Linnenkugel and M. Mullerburg, "Test data selection criteria for (software) integration testing," Systems Integration '90. Proceedings of the First International Conference on Systems Integration, pp. 709-17, April, 1990.

[43] W. Yang, "Identifying syntactic differences between two programs," Software-Practice and Experience, Vol. 21(7), pp. 739-55, July, 1991.

Appendix

User Manual:

The tool developped is in fact very easy to operate even for unexperienced personnel. It has a single interface, which contains 2 fill-in fileds, 3 browsers, and two buttons. The first fill-in field should be filled by test personnel which should contain the name and full path of the original trusted application.

The second fill-in field should aslo filled by test personnel which should contain the name and full path of the modified trusted application.

Next, the "OK" Button is selected, and the whole process runs in the background internally.

The result will be displayed on 3 browsers:

- Browser one contains the CDG table of the first trusted application.
- Browser two contains the CDG table of the modified version.
- Browser Three contains full information on the location of the modified, deleted, or added code, plus the region(s) that are directly affected.