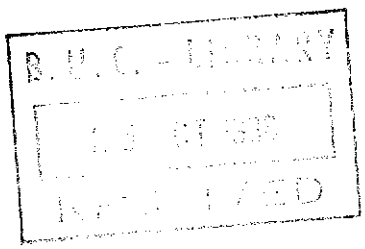
A Comparative Study of Regression Testing Methods

by

Ghinwa S. Baradhi



June 1996

A Comparative Study of Regression Testing Methods

by

Ghinwa S. Baradhi

BS., Beirut University College, 1991

THESIS

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
at the Lebanese American University
June 1996

[redacted]

Dr. Nashat Mansour (Advisor)
Assistant Professor of Computer Science
Lebanese American University

[redacted]

Dr. Ale Hejase
Technical Manager of Future Publishers

[redacted]

Dr. Walid Keirouz
Assistant Professor of Computer Science
Lebanese American University

ABSTRACT

We present a comparative study of popular regression testing algorithms. These algorithms include slicing, incremental, firewall, genetic, and simulated annealing algorithms. The study uses a variety of small-size and medium-size modules along with associated test cases tables, and is based on the following quantitative and qualitative criteria, efficiency, number of retests, precision, inclusiveness, user's parameter setting, global variables, type of maintenance, type of testing, level of testing, and type of approach. The comparison results show that the five algorithms are suitable for different requirements of regression testing. Slicing and adapted firewall algorithms detect the definition-use pairs that are affected by a change, and select the test cases for regression testing based on these definition-use pairs. Incremental algorithm selects the test cases whose outputs may be affected. Genetic and simulated annealing select the minimum number of test cases that provide full testing coverage. In terms of execution time for small-size modules, slicing, incremental, and adapted firewall algorithms exhibit a better behavior comparing to genetic and simulated annealing algorithms. For medium-size modules, the adapted firewall algorithm becomes the slowest. Genetic and simulated annealing algorithms produce the least number of retests, followed by incremental, slicing, and then adapted firewall.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Professor Nashat Mansour, my advisor, for his support, tireless effort and encouragement. I am also grateful to the second readers, Professor Ale Hejase and Professor Walid Keirouz for their help and interest.

Much love and thanks goes to my brother Dr. Nabil Baradhi for his financial support. Special thanks to Dr. Khoury and Dr. Kabani for their personal support and assistance.

Thanks to my sister-in-law Alya Baradhi for her help in typing and valuable comments, and to my fiancé Fayez El Khoury for his understanding and support.

Most of all, I am very grateful to my parents for their devotion to education as a priority in life.

CONTENTS

Chapter

List of Figures

List of Tables

Chapter 1

Introduction

Software maintenance involves changing programs as a result of errors, or alterations in the user requirements [Hartmann and Robson 1989]. During such modifications, new errors may be introduced, causing unintended adverse side effects in the software. Regression testing is the testing process which is applied after a program is modified. It aims at providing confidence that modifications are correct and have not adversely affected other portions of the program. It attempts to revalidate the old functionality inherited from the old version [Agrawal et al. 1993]. The objective is to choose all or only tests that may produce different output to verify that the modified program still behaves the same way as the original program, except where change is expected.

Two types of regression testing have been identified. These are namely progressive and corrective regression testing [Leung and White 1989]. The former involves testing major changes to the specification. The latter is performed on a specification that essentially remains unchanged, so that only minor alterations, that do not affect the overall program structure, require revalidation [Hartmann and Robson 1989].

A number of approaches for regression testing have been proposed. Such approaches are classified as data flow approaches, incremental approaches, slicing approaches, integration approaches, or minimization approaches. Data flow approaches use the data dependencies in a program to guide the selection of the test cases that must be rerun after analysis determines the relationships between definitions of variables and uses of the same variable [Harrold et al. 1993]. Incremental approaches use information saved from previous testing sessions to determine the effects of program modifications. They update the test case requirements to reflect the changed program and identify a subset of the test suite for the regression testing [Agrawal et al. 1993]. Slicing approaches decompose programs by analyzing data flow and control flow graphs. The decomposition yields a method for maintainers to use so that changes can be assured to be completely contained in the modules under consideration, and that there is not undetected "linkages" between the modified and unmodified code [Gallager and Lyle 1988]. Minimization approaches 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 [Fischer and Chruscicki 1981], [Harrold and Rothermal 1994], [Fakih and Mansour 1996].

Regression testing is a significant component of maintenance. Hence, efficient and effective regression testing can reduce the cost of maintenance [Leung and White 1991]. However, the known approaches differ in a number of ways: efficiency; how much they reduce the testing effort; whether they support corrective or perfective maintenance; level of testing to which they apply, i.e. unit, integration, or function, and the ability to regression test global variables.

Comparing and evaluating regression testing techniques has received little attention. In this thesis we present a comparative study of representative regression testing methods. These methods are; *(i)* incremental approach, *(ii)* firewall approach, *(iii)* slicing approach, *(iv)* and the minimization approach. The study uses the following criteria for comparing the method. *(i)* efficiency, *(ii)* number of test cases for regression testing, *(iii)* user's parameter settings, *(iv)* the extent to which global variables complicate the regression testing., *(v)* precision, *(vi)* inclusiveness, *(vii)* level of testing, *(viii)* type of testing, *(ix)* type of maintenance that it supports, *(x)* and type of the approach.

This thesis is organized as follows. Chapter 2 describes the regression testing problem. Chapter 3 includes a survey of regression testing approaches. Chapter 4 addresses the design and implementation of each of the regression testing approaches used in our comparison. In Chapter 5 the comparison criteria are described. In Chapter 6, we report the experimental results, discussion, and comparison. Chapter 7 contains conclusions and suggestions for further research.

Chapter 2

Regression Testing Problem

Regression testing is applied after a program is modified in the maintenance phase. It aims at providing confidence that modifications are correct and have not adversely affected other portions of the program. Regression testing can be corrective, where the software is modified due to error reports or minor alterations. It can also be perfective, where the specifications of the software are changed or enhanced.

2.1. The problem of regression testing

Given a program represented by a control-flow graph with M segments, where a program segment represents a control statement or a continuous sequence of assignment statements. Along with the control-flow graph, the data-flow graph is stored to represent the definitions and uses of the variables for the statements in each segment. Also, assume that the set, $T = \{t_1, t_2, t_3, \ldots\ldots\ldots\ldots, t_n\}$, of test cases used in the initial development of the program is saved and a table of the test cases and the program segments they cover can be determined. Suppose that modification of the program is required. The regression testing problem is to select a subset from T that satisfies certain criteria which are listed in Chapter 5.

2.3.1. Example

Consider the flow graph for a simple program segment given in Figure 2.1. Statements of the form "X := " represent definitions of X, while statements of the form " := X" represent uses of X. Data flow analysis is performed on the program and the required definition-use pairs are identified. Table 2.1 consists of a possible set T of test cases used in the initial development of the program, their execution paths and the definition-use pair(s) satisfied by each of them. A definition-use pair is denoted by an ordered pair where the first coor-

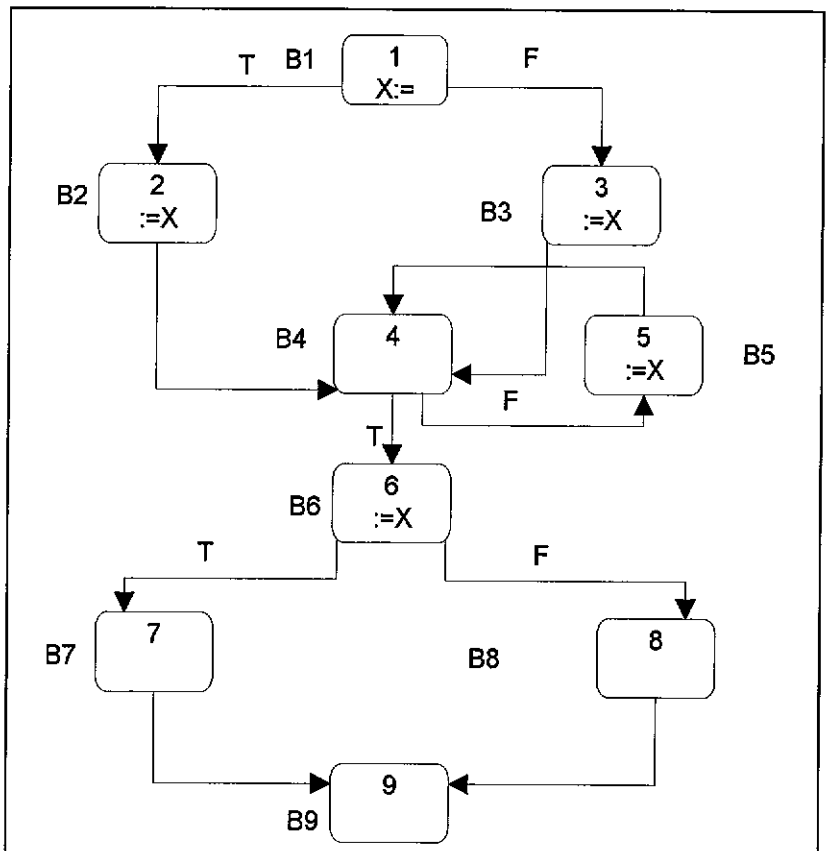


Figure 2.1. Control-Flow/Data-Flow graph for partial example program: "X := " represents a definition of X and ":= X" represents a use of X.

dinate is the node in the control flow graph containing the definition and the second coordinate is the node containing the use. The definition-use pairs for variable X are: (B_1, B_2), (B_1, B_5), (B_1, B_6), (B_1, B_3), (B_3, B_5), and (B_3, B_6).

Table 2.1. Test History Information

test case	execution path	definition-use pair(s)
t_1	$B_1, B_2, B_4, B_6, B_7, B_9$	(B_1, B_2), (B_1, B_6)
t_2	$B_1, B_2, B_4, B_5, B_6, B_7, B_9$	(B_1, B_5)
t_3	$B_1, B_3, B_4, B_6, B_8, B_9$	(B_1, B_3), (B_3, B6)
t_4	$B_1, B_3, B_4, B_5, B_6, B_8, B_9$	(B_3, B_5)

Suppose that the program has been changed by inserting uses of variables X in nodes B7 and B8. Then, the control flow, the definition-use pairs, and new test cases are added and modified accordingly. Figure 2.2 shows the changes that result to the control flow graph. Table 2.2 shows the additions to Table 2.1.

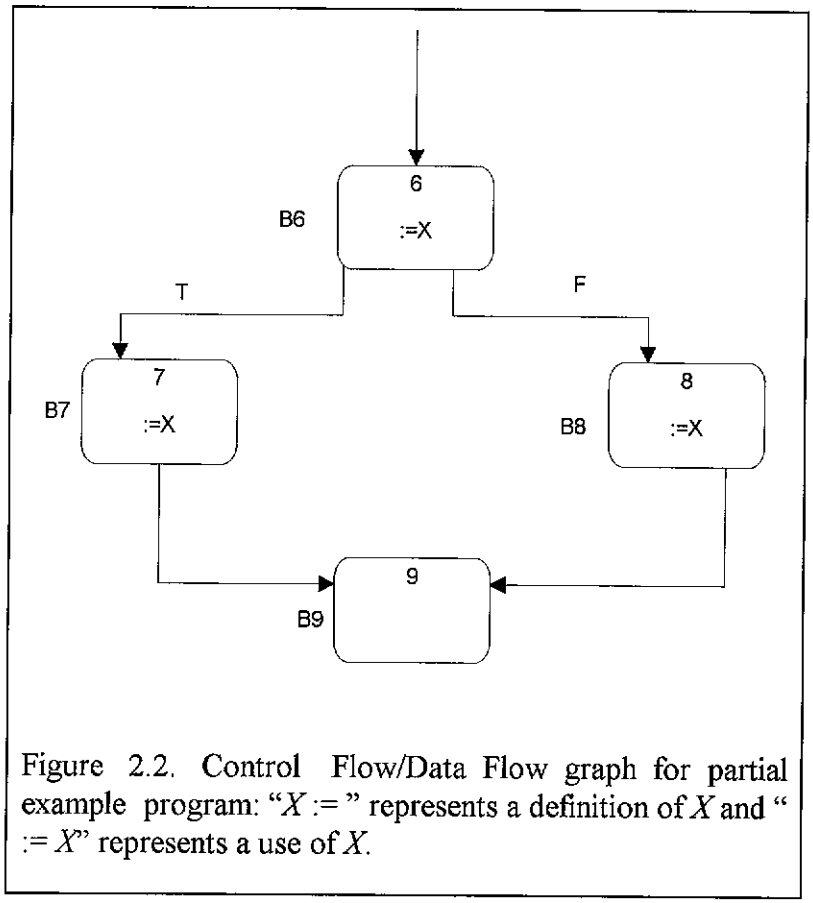


Figure 2.2. Control Flow/Data Flow graph for partial example program: "$X :=$ " represents a definition of X and " $:= X$" represents a use of X.

Table 2.2. Addendum to Table 2.1, Reflecting New Test Case Requirements and New Test Cases

test case	execution path	definition-use pair(s)
t_5	$B_1, B_2, B_4, B_6, B_8, B_9$	(B_1, B_8)
t_6	$B_1, B_3, B_4, B_5, B_6, B_7$	(B_3, B_7)

Suppose that a change in the modified example program indicates that all existing definition-use pairs for the definition of X in B_1 must be retested. These definition-use pairs are: (B_1, B_2), (B_1, B_3), (B_1, B_6), (B_1, B_7), and (B_1, B_8). The associated testing sets for these definition-use pairs are listed in Table 2.3.

Table 2.3. Definition-Use Pairs and Their Associated Testing Sets

definition-use pair(s)	associated testing set
(B_1, B_2)	$\{t_2, t_5\}$
(B_1, B_3)	$\{t_3, t_6\}$
(B_1, B_5)	$\{t_2\}$
(B_1, B_6)	$\{t_2, t_5, t_6\}$
(B_1, B_7)	$\{t_2\}$
(B_1, B_8)	$\{t_5\}$

Test cases t_2 and t_5 are added to the representative set and definition-use pairs (B_1, B_2), (B_1, B_3), (B_1, B_6), (B_1, B_7), and (B_1, B_8) are marked. Test case t_3 is added to the representative set since either of test cases t_3 or t_6 satisfy definition-use pair (B_1, B_3). Rerunning the program with test cases t_2, t_3, and t_5 ,out of the six test cases, provides the desired coverage of the changed program.

Chapter 3

A Survey of Regression Testing Approaches

This chapter presents regression testing approaches and their respective properties. Three popular methods used in our experimental work are explained in some more detail in Chapter 4.

3.1. Input partition strategy

Yau and Kishimoto have described a regression testing tool which is based on the input partition testing strategy [Yau and Kishimoto 1987]. This strategy is based on the assumptions that each program is correct, that a suite of test cases exists, and that the program's specification is in the form of a cause-effect graph. It divides the input domain of a module into different classes using both the program specification and code, and requires one test be selected from each input class. The objective of regression testing is to execute each new or changed input partition at least once. Symbolic execution is used to identify those input domains which are not modified and to aid in test generation. This regression testing strategy selects a subset of the previous tests and some new tests to exercise the modified code. Similar to most regression tools, this tool concentrates on unit regression testing.

This revalidation technique has two advantages over other techniques. First, it can handle any kind of modification, including changes to the program's control structure. Second, it is based on an input-partition approach, which is suitable for program revalidation because it generates test cases by reflecting the changes in both the program specification and code.

In practice, Yau and Kishimoto's technique works satisfactorily for programs that perform input classifications (such as classifying a triangle when given three sides) but does not work as well when executing algorithmic processes (such as sorting routines).

Furthermore, Since it is based on cause-effect graphing to represent the program specification and symbolic-execution techniques for test-case execution, these methods can become very complex and produce a large output.

3.2. Path testing strategy

A post-maintenance testing system is described by Benedusi et al. [Benedusi et al. 1988], and also deals with unit regression testing. It implements a path testing strategy by selecting tests to exercise a specific set of paths. The regression testing strategy uses the following two steps: *(i)* identify those paths that have been added, those deleted and those modified; *(ii)* update and re-run the test cases which exercise the modified and new paths. Algebraic expressions are used to represent the program before and after the program modification. By comparing these expressions using

elementary algebraic operations, it is possible to classify paths affected by the modification into new, deleted, modified and unmodified paths. By storing test cases and their associated program paths in a table, it is straightforward to identify those tests needed to be re-run based on the changed code.

3.3. Incremental data flow strategy

Harrold and Soffa have developed an incremental data flow tester which can be used for regression testing purposes [Harrold and Soffa 1988]. This tool combines data flow testing with incremental data flow analysis to aid in unit and integration regression testing. During the initial testing, the system stores the previous data flow analysis results and test cases. After a module is modified, the system analyzes the effects of the changes on the test history and determines new and existing definition-use pairs for retesting. Existing test cases are reused whenever possible. The system identifies those definition-use pairs that have not been exercised. This tool does not help with test generation and uses only a white-box testing method (structural testing based on data-flow coverage).

Harrold and Soffa have extended their technique to include interprocedural testing [Harrold and Soffa 1993]. Data flow testing is performed across procedure boundaries to integrate those procedures. They present a technique to select a representative set of test cases from a test suite that provides the same coverage as the entire test suite. This selection is performed by identifying, and then eliminating, the redundant and obsolete test cases in the test suite. The representative set replaces

the original test suite and thus, potentially produces a smaller test suite. The representative set can also be used to identify those test cases that should be rerun to test the program after it has been changed. The technique is independent of the testing methodology and only requires an association between a testing requirement and the test cases that satisfy the requirement.

3.4. Zero-One (0-1) integer programming model

Fischer and Chruscicki use zero-one integer programming to find a minimum set of tests which can cover all the program segments [Fischer and Chruscicki 1981]. Four tables are used to record the control flow relation between program segments, reachability between segments, test cases which cover the different segments, and variable use and define information within each segment, respectively. A series of inequality constraint expressions is created, one for each program segment, relating those tests which traverse the segment. The objective function relates the cost of running all the tests to individual tests. Given the program segments that have been modified, standard linear-programming techniques are used to solve for the objective function which can be used to derive the minimum number of test cases that must be rerun to validate the modification. Recall that linear program problems usually terminate with a non-integer solution. To arrive at a zero-one solution, additional constraints and iteration may be needed. However, cutting plane methods sometimes work well and sometimes not at all; branch and bound techniques can spend too much time solving linear programs.

Hartmann and Robson have extended Fischer's method to include programs written in the C programming language, programs with several modules, and segments which have multiple use of variables [Hartmann and Robson 1988]. Since this tool also uses zero-one linear programming, it suffers from the same kind of computational problems associated with other zero-one linear programming problems. This tool has the potential to aid in system regression testing by treating each module as a segment for test coverage purposes. It uses a white-box testing technique (segment cover) that cannot be generalized to include black-box testing.

3.5. Zero-One (0-1) integer programming model using Genetic algorithms

Genetic algorithms are based on the mechanics of natural evolution. They mimic natural populations reproduction and selection operations to achieve efficient and robust optimization. The optimal retesting problem consists of finding values for (X1, X2, ...,X_N) that minimize the cost function

$$\mathbf{Z = c_1X_1 + c_2X_2 + ... + c_NX_N} \qquad (1)$$

Subject to the constraints

$$\sum_{j=1}^{N} \mathbf{a_{ij}X_j} \geq \mathbf{b_i\ ;\ i = 1, ... , M}$$

Where X_j indicates the inclusion (or exclusion) of test case *j* in the selected subset of retests; c_j is the cost element for running each test case; a_{ij} is an element of the test-segment coverage table; and b_i indicates whether segment i needs to be covered by the subset of retests.

An outline of a hybrid genetic algorithm for minimizing that cost function Z, is described in Figure 3.1.

```
Random generation of initial population, Size POP;
Evaluates fitness of individuals;
repeat
    Rank individuals and allocate reproduction trials;
    for (i=1 to POP step 2) do
        Randomly select 2 parents from list of
            reproduction trials;
        Apply crossover and mutation;
    endfor
    Evaluate fitness of offsprings;
    Check feasibility of individuals;
    Do feasibilization, penalization and hill-climbing;
until (termination criterion is satisfied)
Solution = Fittest.
```

Figure 3.1. Hybrid Genetic Algorithm for Optimal Retesting

HGAs algorithm is an array of POP individuals. An individual in the population is encoded as an n-element vector $[X_1, X_2, X_3, X_n]$ that corresponds to a candidate solution for the optimal retesting problem. An element (gene) $X_j = 1$ (or 0) indicates the inclusion (or exclusion) of test case j from the selected subset of retests. The initial population of individuals is randomly generated. The fitness of an individual **Z** is evaluated by adding the genes of an individual. Henceforth, the optimal subset of retests corresponds to the minimum fitness **Z** of all feasible individuals.

3.6. Zero-One (0-1) integer programming model using Simulated annealing algorithm

The simulated annealing algorithm simulates the natural annealing phenomenon by a search (perturbations) process in the solution space (energy landscape) optimizing some cost function (energy) [Mansour and Goel 1994]. It starts with some initial solution at a high (artificial) temperature and reduces the temperature gradually to a freezing point. At each temperature, regions in the solution space are searched by the Metropolis algorithm. The algorithm is given in Figure 5.2, where a candidate solution is represented by the configuration $X_{soln} = (X_1, X_2, \ldots\ldots, X_N)$. The energy is given by the cost function Z in equation 1 described above.

```
Initial configuration = random X_soln;
Determine initial temperature T(0);
Determine freezing temperature T_f;
while (T(i) > T_f and not converged) do
    repeat          /*Metropolis Algorithm*/
        perturb (X_soln);
        if (perturbed X_soln is feasible) then
            accept_or_not();
        else
            reject_perturbation();
    until equilibrium
    save_bestsofar();
    check_convergence();
    T(i+1) = α T(i); /*cooling schedule*/
endwhile;
procedure accept_or_not()
    if (ΔZ ≤ 0) then
        update (X_soln); /*accept perturbation*/
    else
        if (random(0, 1) < e^(-ΔZ/T(i))) then
            update (X_soln)
        else
            reject_perturbation();
        endif;
    endif;
```

Figure 3.2. Simulated Annealing Algorithm for Optimal Retesting.

3.7. Algorithm for selective regression testing

Harrold and Rothermel have presented a new technique for selective regression testing [Harrold and Rothermel 1993]. The algorithm constructs control dependence graphs for program versions, and uses these graphs to determine which tests from the existing test suite may exhibit changed behavior on the new version. Unlike most previous techniques for selective retest, the algorithm selects every test from the

original test suite that might expose errors in the modified program, and does this without prior knowledge of program modifications. It handles all language constructs and program modifications, and is easily automated. However this simple algorithm can be imprecise. It considers statement order significant. Thus, a small swap statement between two unrelated assignments could be caught by the algorithm as a real change in behavior where it should not be.

3.8. Retest strategy for corrective regression testing

Leung and White have suggested classifying the initial test cases as reusable, retestable, obsolete, new-structural and new-specification test cases [Leung and White 1989]. A problem facing those conducting maintenance testing is to identify the proper test classes. A Retest Strategy for performing corrective regression testing is proposed, where testing involves test cases from the reusable, retestable, and new-structural classes. The guiding principle of Retest is to view the regression testing problem as composed of two sub-problems: the test selection problem and the test plan update problem, and to structure the retesting process into two phases: the test classification phase and the test plan update phase.

3.9. Global variables regression testing and integration testing

Leung and White extended their work and presented some insights into the problem of testing and regression testing global variables [Leung and White 1990b]. It has

been shown that global variables can be treated as parameters and can be tested accordingly.

Later, Leung and White identified the common errors and faults in combining modules into a working unit [Leung and White 1990a]. They made practical recommendations for regression testing at the integration level. They showed that with emphasis on reusing the previous test cases and retesting only the parts that are modified, one can reduce the testing expenses. They proposed the concept of "firewall" to assist the tester in focusing on that part of the system where new errors may have been introduced by a correction or a design change.

In 1992 they have provided some additional new work on regression testing for function testers and global variables [Leung and White 1992]. They present a new methodology that involves regression testing of modules where dependencies due to both control-flow dependency is modeled as a call graph, and a firewall defined to include all affected modules which must be retested. Global variables are considered as the remaining data-flow dependency to be modeled, an approach to testing and regression testing of these global variables is given, including the definition of a firewall concept for the data-flow aspect of software change.

3.10. Regression testing tool based on the "firewall" concept

The development of a regression-testing tool was undertaken for IBM in order to provide a capability to support regression testing for both developers and function/system testers [White et al. 1993]. This internal regression testing tool is

called Test Manager. The objective of Test Manager is to aid in regression testing at the unit testing, integration testing or function testing levels, and also to produce a reduced regression test set to verify the software changes indicated by the user. For small software changes, one aspect of this support would be to identify small subsets of potentially large regression test buckets in order to obtain a more focused test of the modified areas of the software. Another aspect of this support is the ability to examine the modified procedures (or blocks in the case of developers), to identify all those additional procedures (or blocks) affected by the change, and to observe how the subset of test cases interact with all these procedures (or blocks). Test Manager has been designed assuming that procedures are totally hierarchical, i.e., each procedure belongs to a unique module, and each module belongs to a unique component. Sharing procedures between components or modules is needed. It is required to see if Test Manager must be modified to handle any of these non-hierarchical situations. Given the research which has been accomplished on regression testing global variables and data flow dependencies, the possibility of adding this capability of regression testing variables to Test Manager should be considered.

3.11. Data flow technique based on slicing

Gallager and Lyle have presented a data flow technique based on program slicing that can be used to form the direct sum decomposition for software systems [Gallager and Lyle 1991]. The decomposition yields a method and guidelines for "software surgeons" (maintainers) to use. Under suitable conditions, the surgeon is able to

modify existing code cleanly, in the sense that the changes can be assured to be completely contained in the modules under consideration and that no unseen "linkage" with the modified code is infecting other modules. Also, This decomposition can be used to limit the amount of testing required to assure that the change is correct.

Chapter 4

Implemented Algorithms

In this chapter we present the implementations on which we have based our comparative studies. The first implementation [Harrold et al. 1992] is based on slicing. The second [Agrawal et al. 1993] is based on incremental regression testing. The last is based on the firewall concept [Leung and White 1990a].

4.1. Slicing algorithm

Harrold, Gupta, and Soffa have presented a novel approach to data flow based regression testing [Harrold et al. 1992]. This approach uses slicing type algorithms to explicitly detect definition-use pairs that are affected by a program change. The advantage of this approach is that no data flow history is needed, nor is the recomputation of data flow for the entire program required to detect affected definition-use pairs. Another advantage is that the technique achieves the same testing coverage as a complete retest of the program without maintaining a test suite.

4.1.1 Program slice

A program slice consists of all statements, including conditionals in the program, that might affect the value of variable V at point p. Slicing algorithms are used to identify all definition-use pairs that may be directly or indirectly affected by a program change.

4.1.2. Definition-use pairs

The program is represented by a control flow graph where each node in the graph corresponds to a statement and each edge represents the flow of control between statements. Definitions and uses of variables are attached to nodes in the control flow graph. Data flow analysis determines the relationships between definitions of variables and uses of the same variable. Definitions of variables occur in statements where a variable gets a value, such as assignment statements and input statements. Uses of variables occur where a variable's value is fetched, such as output statements, conditional statements and the right-hand side of assignment statements. Uses are classified as either computation uses (c-uses) or predicate uses (p-uses). A c-use occurs whenever a variable is used in a computation statement. A p-use occurs whenever a variable is used in a computational statement. A definition that reaches a use forms a definition-use pair.

4.1.3. Slicing algorithms

The technique uses two slicing type analysis algorithms to determine directly the affected definition-use pairs and the indirectly affected definition-use pairs.

4.1.3.1. Backward walk algorithm

This algorithm is a backward walk through the program from the point of the edit that searches for definitions related to the changed statement. It identifies the definitions of a set of variables ***U*** that reach a program point ***s***. An outline of the algorithm is given in Figure 4.1.

```
BackwardWalk(s, U)
Begin
        OUT =all variables of the successors of s whose definitions have
                not been encountered
        IN  =all variables of the predecessors of s whose definitions have
                not encountered
        For all nodes n ∈ predecessors of s
                queue = n + reverse-depth-first (queue)
        While queue not empty
                Get n from head of queue
                if a new successor has been encountered
                        recompute OUT[n]
                        if n defines a variable u ∈ U then
                                add n to definition set
                                stop searching for definitions of u
                        else propagate
                        if there exist a predecessor in IN not encountered
                                For all nodes x ∈ predecessors of n
                                    queue=x+reverse-depth-first (queue)
        Return definition set
End
```

Figure 4.1. Outline of Backward algorithm.

The algorithm traverses the control flow graph from the point s in the backward direction until definitions of all variables of U are encountered along each path. Sets of variables are maintained for relevant nodes in the control flow graph. One set contains the variables of the successors of s whose definitions have not been encountered. And the other set contains the variables of the predecessors of s whose definitions have not been encountered. A queue is maintained based on a reverse depth first ordering of nodes in the control flow graph. The queue consists of those nodes that must be visited and examined for definitions of variables in U. A statement is added to the queue if the set consisting of its successor is not empty. When the queue is empty, all definitions of all variables in U have been encountered along all backward paths from s and the algorithm terminates.

4.1.3.2. Forward walk algorithm

This algorithm is a forward walk through the program from the point of the edit that detects the uses and the subsequent definitions and uses that are affected by a definition that is changed as a result of the program edit. Any definition-use pairs that depend on a changed predicate are identified. Also, the algorithm identifies definition-use pairs that are control dependent on an affected predicate, even though the value computed by the definition is unaffected. The algorithm is outlined in Figure 4.2.

It identifies the uses of values, that are directly or indirectly affected by a change in either a value of v at point s or a change in a predicate. ValueUseTriples have the form of triples (s,u,v) indicating that the value of variable v at statement s, affected by the change, is used by statement u. The returned ValueUseTriples are used to

compute the affected definition-use pairs. The algorithm inputs a set of pairs representing the definitions whose uses are to be found, along with a boolean that indicates whether the change is only a predicate change. For each statement node *n*, two sets are maintained containing the definitions whose uses are to be found, since their values are affected by the edit. One set contains the values just before *n* whose uses are to be found. The other set contains the values just after *n* whose uses are to be found. A queue is maintained based on a depth first ordering of nodes in the control flow graph. The queue consists of those nodes that must be visited and examined for c-uses and p-uses of the definitions of variables in the set consisting of the values just before *n*, along with definitions in statements that are control dependent on a changed or affected predicate. When the queue is empty, all ValueUseTriples that are affected by a change are identified and the algorithm terminates.

```
ForwardWalk(Pairs, PredOnly)
Begin
        For all nodes consisting of (s, v) ∈ Pairs
                For all nodes n ∈ successors of s
                        queue = n + depth-first (queue)
        OUT[n] = all values just after n whose uses are to be found
        IN[n]   = all values just before n whose uses are to be found
                not encountered
        While queue not empty
                Get n from head of queue
                if a new predecessor has been encountered
                        recompute IN[n]
                        compute control-dependencies of n
                        if n has a c-use of variable v ∈ IN[n]
                                update ValueUseTriples
                                update OUT[n]
                        else
                        if n has a p-use of variable v ∈ IN[n]
                                update ValueUseTriples
                                Backward on n to get the definitions of v
                                update IN[n]
                                update OUT[n]
                        if there exist a successor in OUT not encountered
                                For all nodes x ∈ successors of n
                                    queue=x + depth-first (queue)
        Return ValueUseTriples
End
```

Figure 4.2. Outline of Forward algorithm.

4.1.4. Example

The control flow of an example program is given in Figure 4.3. The program should compute the square root for input variable X. Table 4.1 gives the def-use pairs for the program that are required to satisfy the "all-uses" data flow testing criterion. The program has an error in statement 8; it should read "X1:=X3". Correcting the error requires two actions.

(1) Delete the use of X2 in statement 8.

ForwardWalk on X2 in statement 8 first finds the affected definitions of X1 in statement 8. Then, uses of X1 in statement 4, 5, and 6 are detected. Thus, detected def-use pairs for X1 are (8,(4,6)), (8,(4,4)), (8,4), (8,(6,7)) and (8,(6,8)). Since definition of X3 in statement 4 may be affected, def-use pairs for X3,(4,(6,7)), (4,(6,8)) and (4,7) are detected. The use of X3 in statement 7 may affect the definition of X2 and def-use pairs of X2 (7,(4,4)), (7,(4,10)) and (7,4) are identified.

(2) Insert the use of X3 in statement 8.

BackwardWalk on statement 8 for X3 finds the reaching definitions of X3 in statement 4 and returns the def-use pair (4,8) for X3. Since ForwardWalk on X3 in statement 8 is identical to the ForwardWalk performed above, the same def-use pairs are returned..

Table 4.2 lists the def-use pairs that are affected after the program change and must be tested.

Table 4.1. Def-use Information

node	definition	def-use pairs
1	X	(1,3),(1,6)
2	X1	(2,(4,4)),(2,(4,10)),(2,4) (2,(6,7)),(2,(6,8))
3	X2	(3,(4,4)),(3,(4,10)),(3,4)
4	X3	(4,(6,7)), (4,(6,8)),(4,7),(4,8)
7	X2	(7,(4,4)), (7,(4,10)),(7,4)
8	X1	(8,(4,6)), (8,(4,10)),(8,4) (8,(6,7)),(8,(6,8))

Table 4.2. Def-use Pairs for Regression Testing

node	definition	def-use pairs
5	X3	(4,(6,7)), (4,(6,8)),(4,7),(4,8)
7	X2	(7,(4,4)), (7,(4,10)),(7,4)
8	X1	(8,(4,6)), (8,(4,10)),(8,4) (8,(6,7)),(8,(6,8))

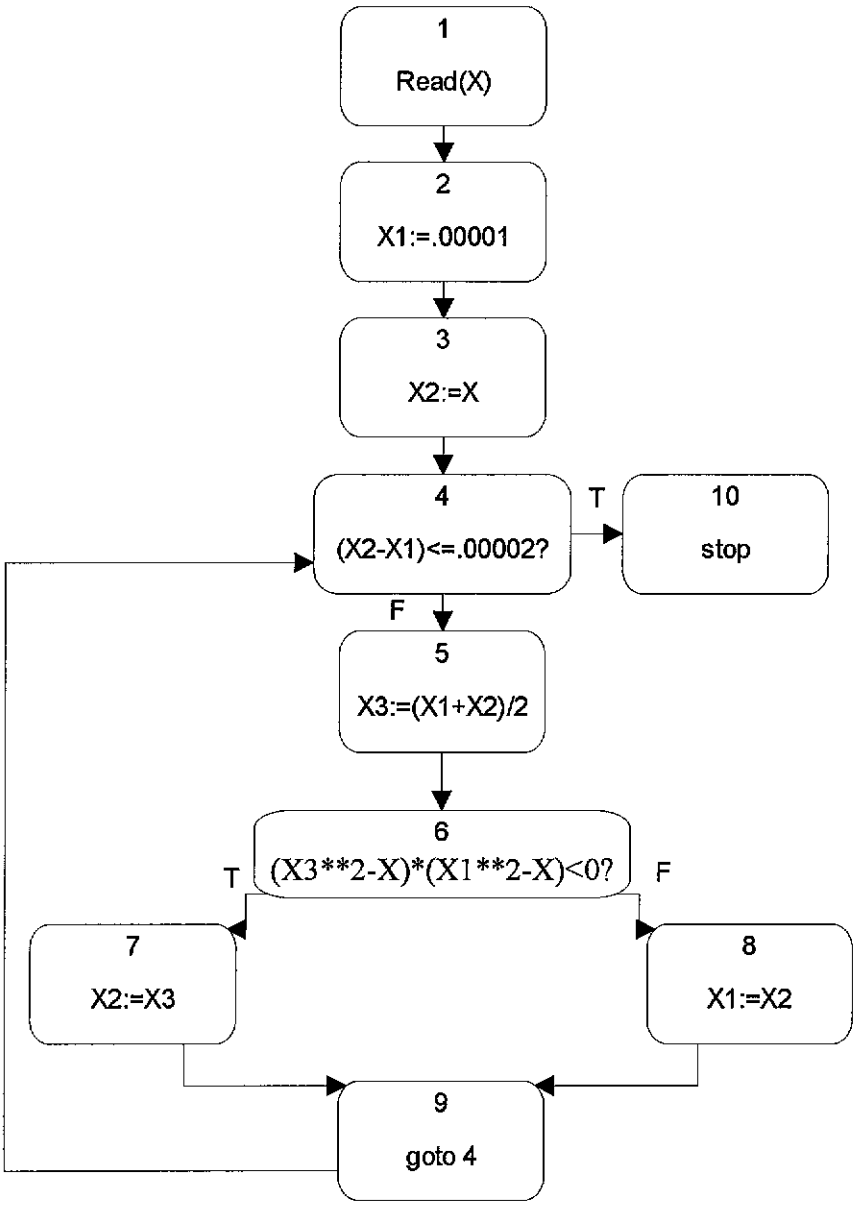


Figure 4.3. Program to compute the square root of X with an error in statement 8.

4.2. Incremental algorithm

Agrawal, Horgan, and Krauser present efficient methods of selecting small subsets of regression test sets that may be used to ensure that bug fixes and new functionality introduced in a new version of a software do not adversely affect the correct functionality inherited from the previous version [Agrawal et al. 1993]. Using these methods, the test cases in the regression test suite whose outputs may be affected by the changes to the program may be identified automatically.

The problem of determining the test cases in a regression test suite on which the modified program may differ from the original program is referred to as the incremental regression testing problem.

4.2.1. Incremental regression testing techniques

The techniques implemented in this approach are based on the following observations:

1) If a statement is not executed under a test case, it can not affect the program output for that test case.
2) Not all statements in the program are executed under all test cases.
3) Even if a statement is executed under a test case, it does not necessarily affect the program output for that test case.
4) Every statement does not necessarily affect every part of he program output.

4.2.1.1. The execution slice technique

This technique is based on observation 1 and 2 above. Execution slices are composed of all statements executed under a test case. If a test case does not execute any of the modified statements, it does not need to be rerun. Execution slices are computed during off-line processing. After the program is modified, the new program is rerun on only those test cases whose execution slices contain a modified statement. However execution slices fail to identify an error in a predicate statement, thus another technique is used to solve this problem. For this algorithm, it has been assumed that the execution slices for the test cases are given.

4.2.1.2. The dynamic slice technique

This technique is based on observation 3. It suggests that not every statement that is executed under a test case has an effect on the program output for that test case. Dynamic program slices are computed to determine for each test case the statements that have an effect on the program output. A dynamic program slice is obtained by recursively traversing the data and control dependence edges in the dynamic dependence graph of the program for the given test case. All slices are computed during off-line processing. Then, after the program is modified, the new program is rerun on only those test cases whose dynamic program slices contain a modified statement. Figure 4.4 outlines the algorithm for computing the dynamic slices.

However, this technique fails to identify test cases that if executed will affect the output if they are evaluated differently. The relevant slice technique will solve the problem.

```
For each test case
    Recursively DO
    For each statement S in the test case
        if a p-use is found
            search for its successor(s) in the dynamic slice
                if found
                    add S to the dynamic slice
        else if a definition is found
            if the dynamic slice statements does not contain
               any of its successor
                    include S in the dynamic slice
    endfor;
endfor;
```

Figure 4.4. . Dynamic Slice Algorithm.

4.2.1.3. The relevant slice technique

Relevant slices are composed of those statements that if executed do not affect the output, but would have if they are evaluated differently. Those statements should be identified because if changes are made to them they may evaluate differently and change the program output. These slices are obtained by identifying the predicates on which the statements in the dynamic slices are potentially dependent, as well as the closure of data and potential dependencies of these predicates. The set of statements corresponding to these predicates including those in the dynamic slice, gives the desired relevant slice. After the program is modified, the new program is rerun on only those test cases whose relevant slices contain a modified statement. Figure 4.5 depicts the algorithm for computing the potential dependencies of a variable, *var*, at a

location, *loc*, in a given execution history. The relevant slices are composed of the union set of the potential dependences and the dynamic slice for each test case in the test suite.

```
static-defs = static reaching definitions of var at loc;
dynamic-defs = the dynamic reaching definition of var at loc;
control-nodes = the closure of the static control dependences
                    of the statements in static-defs;
initialize potential-deps to null;
mark all nodes in the dynamic dependence graph between loc and
        dynamic-def as unvisited;
for each node, n, in the dynamic dependence graph starting at
        loc and going back up to dynamic-def do
    if n is marked as visited then
        continue; /*skip this node*/
    mark n as visited;
    if n belongs to control-nodes then
        add n to potential-deps;
        mark all the dynamic control dependences of n
            between n and dynamic-defs as visited;
    endif;
endfor;
return potential-deps;
```

Figure 4.5. Algorithm to compute the potential dependences of a variable, *var*, at a location, *loc*, in a given execution history

4.2.2 Example

Consider the program in Figure 4.6. It reads the lengths of the three sides of a triangle, classifies the triangle as one of a scalene, isosceles, right, or an equilateral triangle, computes its area, and outputs the class and the area computed. Table 4.3 shows the test cases using which the program is tested.

```
S1:  read(a,b,c);
S2:  class := scalene;
S3:  if a = b or b = a
S4:          class := isosceles;
S5:  if a * a = b * b + c * c
S6:          class := right;
S7:  if a = b and b = c
S8:          class := equilateral;
S9:  case class of
S10:                 right          : area := b * c / 2;
S11:                 equilateral    : area := a *2 * sqrt(3) / 4;
S12:                 otherwise      : s := (a + b + c) / 2;
S13:                                    area:=    sqrt(s*(s-a)*(s-
b)*(s-c));
     end;
S14:         write(class, area);
```

Figure 4.6: An example program

Table 4.3. An example test suite

Testcase	Input			Output	
Variables	a	b	c	class	area
T1	2	2	2	equilateral	1.73
T2	4	4	3	isosceles	4.46
T3	4	4	3	right	6.00
T4	6	4	4	scalene	9.92
T5	3	3	3	equilateral	2.60
T6	4	3	3	scalene	4.47

Consider the test case T4, the program correctly classifies the triangle as an equilateral triangle, but incorrectly computes its area as 2.6 instead of the correct value, 3.9. Debugging reveals that there is a fault in statement S11. It uses the expression a * 2 instead of a * a. After the fault is corrected, the regression testing consists in this case of executing the new program on the test cases whose execution slices contain the S11 statement. After applying the execution slice technique, two test cases, T1 and T5, out of the initially five test cases should be rerun. Suppose that

T6 is a new test cases added to the test suite table. This test case classifies the triangle as a scalene instead of isosceles. After debugging the error is corrected in statement S3. It should be b = c instead of b = a. The execution slice technique implies that all test cases should be rerun, since they all pass by S3. However, test cases that classify the triangle as right or equilateral will not affect the output. This problem is solved by the dynamic slice technique. The dynamic program slices consist only of those statements that are actually executed under a test case. Figure 4.7 and Figure 4.8 show the shaded statements of the corresponding dynamic slices.

```
S1:  read(a,b,c);
S2:  class := scalene;
S3:  if a = b or b = a
S4:          class := isosceles;
S5:  if a * a = b * b + c * c
S6:          class := right;
S7:  if a = b and b = c
S8:          class := equilateral;
S9:  case class of
S10:                 right          : area := b * c / 2;
S11:                 equilateral    : area := a *2 * sqrt(3) / 4;
S12:                 otherwise      : s := (a + b + c) / 2;
S13:                                    area:=   sqrt(s*(s-a)*(s-
b)*(s-c));
     end;
S14:         write(class, area);
```

Figure 4.7: The Dynamic Program Slice for T1.

However. the dynamic slice technique depends on which statements were changed and not on how they were changed. For example, changing the statement in S3 to b = b instead of b = c, will still imply that case T4 should not be rerun. Thus this technique fail to identify such test cases that would produce different output. This problem is

resolved using the relevant slice technique. Figure 4.9 shows the relevant program slice for the test case T4.

```
S1:  read(a,b,c);
S2:  class := scalene;
S3:  if a = b or b = a
S4:          class := isosceles;
S5:  if a * a = b * b + c * c
S6:          class := right;
S7:  if a = b and b = c
S8:          class := equilateral;
S9:  case class of
S10:                 right          : area := b * c / 2;
S11:                 equilateral    : area := a *2 * sqrt(3) / 4;
S12:                 otherwise      : s := (a + b + c) / 2;
S13:                                  area:=   sqrt(s*(s-a)*(s-
b)*(s-c));
     end;
S14:         write(class, area);
```

Figure 4.8: The Dynamic Program Slice for T4.

```
S1:  read(a,b,c);
S2:  class := scalene;
S3:  if a = b or b = a  (the changed statement belongs to
S4:          class := isosceles;     the relevant slice)
S5:  if a * a = b * b + c * c
S6:          class := right;
S7:  if a = b and b = c
S8:          class := equilateral;
S9:  case class of
S10:                 right          : area := b * c / 2;
S11:                 equilateral    : area := a *2 * sqrt(3) / 4;
S12:                 otherwise      : s := (a + b + c) / 2;
S13:                                  area:=   sqrt(s*(s-a)*(s-
b)*(s-c));
     end;
S14:         write(class, area);
```

Figure 4.9: The Relevant Program Slice for T4.

4.3. Firewall concept

Integration testing is an important phase of the testing process[Leung and White 1990a]. In their study, Leung and White identify the common errors and faults in combining modules into a working unit. They propose the concept of "firewall" to assist the tester in focusing on that part of the system where new errors may have been introduced by a correction or a design change. For our study purpose, we implemented the "firewall" concept at the unit level, where we call it an adapted firewall algorithm.

4.3.1. Adapted firewall algorithm

Leung and White (1990a) have proposed building "firewalls" to confine integration testing to a small set of modules rather than allowing it to spread to many other modules. The construction of a "firewall" involves the modules that are modified and their direct ascendants and direct descendants.

The adapted firewall is based on the control-flow graph where each node is a segment and the arcs represent connectivity between segments. Associated with the control-flow are a connectivity matrix and a reachability matrix that depict the connectivity of segments to each other and the reachability of segments from each other respectively. Additionally, a test case cross reference matrix is associated with the control-flow graph that identifies segments executed by test cases during the testing phase of software development.

The adapted firewall algorithm proposed encloses the set of segments that must be unit-tested after a modification is made to the program. A "firewall" is built around the modified segments and some related segments. Only those segments within the "firewall" need to be re-unit and regression tested.

The construction of the "firewall" involves consideration of all segments which are not modified and not affected but interact with the modified or affected segments. These are :

i) the direct ascendants of definitions that are affected by a program edit,

ii) and the direct descendants where no uses are affected by the program edit.

The algorithm to construct the "firewall" at the unit level is described in Figure 4.10.

a. Let E be the subset of arcs in the control-flow graph
that defines the "firewall"
Let **W** be the set of all modified segments
Initially **E** is empty
b. **For** each modified segment in **W**,
perform a backward walk from the point of edit in
the control-flow graph until a definition, ***d***,
related to the edit is found
for each segment **A** that is connected to the definition ***d*** do:
add **A** and the arc from **A** to the definition ***d*** to **W**
add all successors and predecessors of **A** not connected
to the definition to **E**
c. From definition ***d***, perform a depth-first-search procedure until either :
a c-use occurs then,
add this segment with all its predecessors to **W**,
or,
a leaf node with no c-use is found: then,
add this segment with its predecessors to **E**
d. after steps (b) and (c) recursively described have been considered,
the "firewall" **E** is complete.

Figure 4.10. "Firewall" Concept Algorithm at the Unit Level.

4.3.2. Example

Consider the control flow graph of Figure 4.11. Suppose that a modification on variable **X** has been done at segment **7**. Applying the algorithm leads the following: the backward walk identifies segment **4** where a definition of variable **X** is found. Then, segments **2** and **3** with their arcs, **(2, 4)**, **(3, 4)**, to the definition in segment **4** are added to **X**. Segment **1** with its arcs **(1, 2)**, **(1, 3)**, are added to the firewall **E**. Next, a depth-first-search procedure is applied leading to the addition of segment 7 where a c-use of variable **X** is found to **W**. Also, predecessor of segment 7, segment **5** is added to **W**. As a result, the firewall is composed of the components **(1, 2)** and **(1,**

3). And **W** is the component composed of **(2, 4)**, **(3, 4)**, and **(5, 7)** for which every test case that passes through these pairs must be retested.

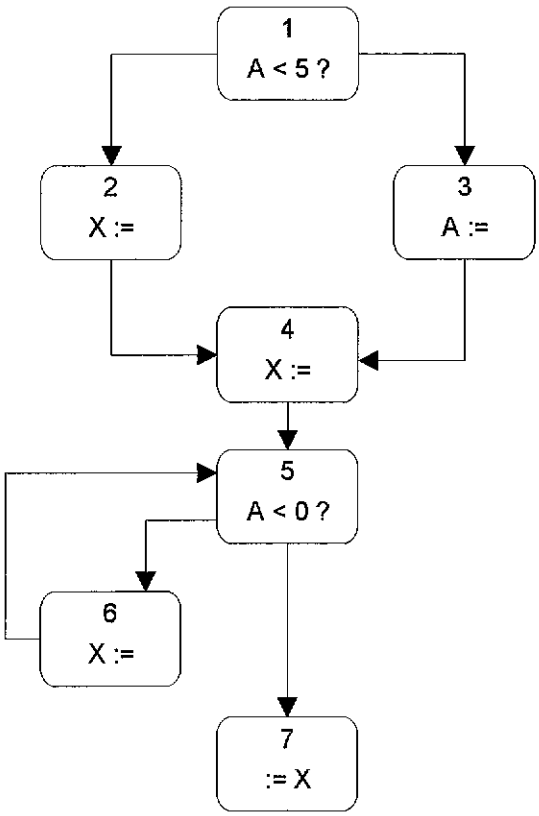


Figure 4.11. Control flow/Data flow Program Example.

Chapter 5

Comparison Criteria

In this chapter we present the criteria used for evaluating and comparing regression testing approaches. The criteria are divided into two categories: quantitative and qualitative. The quantitative criteria are efficiency, quality, precision, and inclusiveness. The qualitative criteria are user's parameter setting, global variables, type of maintenance, type of testing, level of testing, and type of approach.

5.1. Quantitative criteria

5.1.1. Efficiency

The efficiency of a regression testing method is measured in terms of its space and time requirements [Harrold and Rothermel 1995]. Space efficiency is determined by the test history and program analysis information a method must store. The efficiency is determined by the execution time of a regression testing algorithm as well as the time needed for the generation of data flow and control flow graphs. However, methods where information on program modifications is needed, may require more computational resources.

5.1.2. Number of test cases for regression testing

The number of test cases for regression testing refers to the number of test cases selected to be rerun.

5.1.3. Precision

Before defining precision, it is necessary to define some keywords.

(i) modification-revealing tests

a test T_i belonging to the test suite T is modification-revealing, if it produces different outputs in the original program as well as in the modified version [Harrold and Soffa 1994].

(ii) modification-traversing tests

tests that execute modified code are said to be modification-traversing.

However, although modification-revealing tests are necessarily modification-traversing, not all modification-traversing tests are modification-revealing. For example, consider the program example AVG in Figure 5.1 with its history information in Table 5.1. Suppose statement S1 count = 0, is modified. In this case, test T2 is modification-traversing because it executes the modified version of S1 in the new version of the procedure. However, test T2 is not modification-revealing:

it traverses no statement that uses the value computed in S1 and thus cannot produce different output in the new version of AVG [Harrold and Soffa 1994].

```
S1:  count = 0
S2:  fread(fileptr, n)
S3:  while (not EOF) do
S4:          if (n < 0)
S5:                  return(error)
             else
S6:                  numarray[count] = 0
S7:                  count++
             endif
S8:          fread(fileptr, n)
     endwhile
S9:  avg = calcavg(numarray, count)
S10:         return(avg)
```

Figure 5.1. AVG Program Example.

Table 5.1. Test history information for AVG program

test number	input	output	execution history
T1	empty file	0	S1,S2,S3,S9,S10
T2	-1	error	S1,S2,S3,S4,S5
T3	1 2 3	2	S1,S2,S3,S4,S6,S7,S8,S3,.....,S9,S10

Precision measures the extent to which a regression testing method omits tests that are non-modification-revealing [Harrold and Rothermel 1995]. It is the ability of a method to avoid choosing these tests that will not cause the modified program to produce different output. A test would produce different output for the modified program, if it executes modified code. We can compare the precision of methods by showing how much they can select the modification-revealing tests only. Suppose the test suite T contains n non-modification-revealing tests, and a regression testing approach, S, selects m of these tests, then the precision of S

relative to the original program, the modified program, and the test suite ***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 the modified program, and ***S*** omits 33 of these 44 tests, then ***S*** is 75% precise relative to the original program, the modified program, and the test suite ***T***.

5.1.4. Inclusiveness

Inclusiveness measures the extent to which a regression testing method chooses tests that will cause the modified program to produce different output. It is the ability of a method to specifically select all tests that cover affected statements. We can compare the inclusiveness of a method by showing how much it can account for the effects of new and/or deleted statements. Also, conclusions about inclusiveness can be drawn by analyzing the test suite available and checking for a modification-revealing test that is missed out by the regression testing method. Suppose the test suite ***T*** contains ***n*** modification-revealing tests, and a regression testing approach, ***S***, selects ***m*** of these tests, then the inclusiveness of ***S*** relative to the original program, the modified program, and the test suite ***T***, is the percentage calculated by the expression **((*m*/*n*) * 100)**.

For example, if ***T*** contains 50 tests, 8 of which are modification-revealing with respect to the modified program, and ***S*** selects only 2 of the 8 modification-

revealing tests, then S is 25% inclusive relative to the original program, the modified program, and the test suite T.

5.2. Qualitative criteria

5.2.1. User's parameter setting

Regression testing methods might need some parameters to be set by the user.

5.2.2. Type of maintenance

Usually, software maintenance is divided into four activities; *(i)* adaptive maintenance is the modification of a system because of some change to its external environment, *(ii)* corrective maintenance is the modification of a system to correct an error, *(iii)* perfective maintenance is the modification of a system to enhance its functionality, *(iv)* preventive maintenance is the modification of a system to ease future maintenance. In our study, we compare the regression testing methods according to their use for corrective and/or perfective type of maintenance.

5.2.3. Type of testing

Regression testing methods can be based on functional testing and/or structural testing.

Functional testing considers the program as a black-box and is not concerned with its internal structure and behavior. It is concerned with determining whether there are instances in which the program does not conform to its specifications. However, structural testing uses knowledge of the program's construction. The testing is based on the internal structure and logic of the program.

5.2.4. Level of testing

The level of testing is determined by the levels of abstraction a regression testing method can handle. We study the ability of an approach to handle various levels of testing. Unit testing level, where testing is applied to each individual procedure/module. Integration testing level, where testing involves sets of procedures, ideally in an incremental fashion, so that one or more procedures are added to those already integration tested. Function testing level, in which testing is applied to the entire software system, using the software functional specifications.

5.2.5. Type of approach

Regression testing methods can be classified as coverage, minimization, or safe methods. Methods that select only modification-traversing tests are called coverage methods. Methods that return small test sets and thus reduce the regression testing time are called minimization methods. Methods that always select all modification-revealing tests are called safe methods.

5.2.6. Global variables

The ability to regression test global variables should be considered. It is important not to overlook testing these variables. Using global variables affects not only the understandability, readability and maintainability of the software, but also its testability. Global variables create data-flow dependencies between modules which are not directly callable and may be well separated in the call graph [Leung and White 1990b].

Chapter 6

Experimental Results, Discussion, and Comparison

In this chapter, we compare the quantitative and qualitative properties of slicing, incremental, adapted firewall, genetic, and simulated annealing algorithms described in Chapters 3 and 4.

6.1. Quantitative criteria results

In this section, we present the results for the efficiency, number of test cases to rerun, precision and inclusiveness of the above mentioned algorithms. The experiments were done on a PC with an INTEL CPU486 DX4 100 MHz. Twenty different programs (m1 _ m20) were used, for which the codes were written and the graphs were generated manually along with the test cases tables. These programs are of small and medium sizes. With each program are associated: the number of segments, M, the number of the modified segment, S_{mod}, the control-flow and data-flow graphs of M segments, a table of N test cases and their segment coverage information.

Table 6.2 gives the running times, texec, in seconds, of slicing, incremental, adapted firewall, genetic, and simulated annealing algorithms for small-size modules (m1 _ m14) and medium-size modules (m15 _ m20). Also, the table gives the number of test cases that should be rerun after modifications. The results are also depicted in Figures 6.1

through 6.8 for small-size and medium-size modules. Table 6.1 explains the symbols used in Table 6.2.

M	Number of segments
N	Number of test cases
S_{mod}	Number of the modified segment
#T	Number of test cases to rerun
texec	Running time of the algorithm

Table 6.1. Parameters used in Tables 6.2.

				Slicing Algorithm		Incremental Algorithm		Adapted Firewall Algo		Genetic Algorithm		Simulated Annealing	
	M	N	S_{mod}	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
m1	8	6	3	0.3	5	0.3	4	0.3	5	8.0	2	0.3	2
m2	8	32	8	0.8	26	0.8	8	0.9	23	6.7	1	2.1	1
m3	10	15	8	0.3	5	0.6	5	1.3	5	4.9	1	1.0	1
m4	10	27	10	0.5	8	1.0	8	1.2	8	1.4	1	1.8	1
m5	14	24	13	1.4	6	1.4	6	1.8	8	4.2	2	1.7	1
m6	14	32	14	1.5	32	0.7	8	1.6	32	4.0	1	2.3	1
m7	21	9	12	1.3	5	1.3	5	1.7	9	3.5	2	0.5	2
m8	21	18	19	2.0	11	2.0	7	2.4	10	7.4	1	1.3	1
m9	24	18	9	0.9	5	0.9	3	1.3	8	9.2	3	1.1	3
m10	24	23	12	1.3	5	1.3	3	1.6	10	9.8	2	2.0	3
m11	34	58	29	5.6	12	3.1	8	6.0	15	9.9	2	4.3	2
m12	34	126	32	7.0	29	5.2	8	8.0	31	9.1	1	10.6	1
m13	40	20	21	2.3	7	2.3	3	4.1	9	4.6	3	1.4	3
m14	40	28	39	4.6	9	4.2	3	6.9	9	5.3	1	2.0	1
m15	45	32	27	2.9	4	2.9	4	4.6	8	7.3	4	2.5	3
m16	45	96	43	7.0	16	7.0	8	11.8	20	6.3	4	8.2	3
m17	56	34	41	5.5	8	6.7	9	8.2	8	5.7	3	2.8	3
m18	56	108	50	9.4	22	8.2	5	15.0	23	5.9	2	9.6	2
m19	60	90	30	4.9	15	4.9	3	7.8	17	7.9	4	7.6	3
m20	60	145	46	9.2	22	6.5	2	13.4	25	6.4	3	12.9	3

Table 6.2. The running times in seconds, and the number of tests to rerun of Slicing, Incremental, Adapted Firewall, Genetic, and Simulated Annealing algorithms for modules m1 to m20.

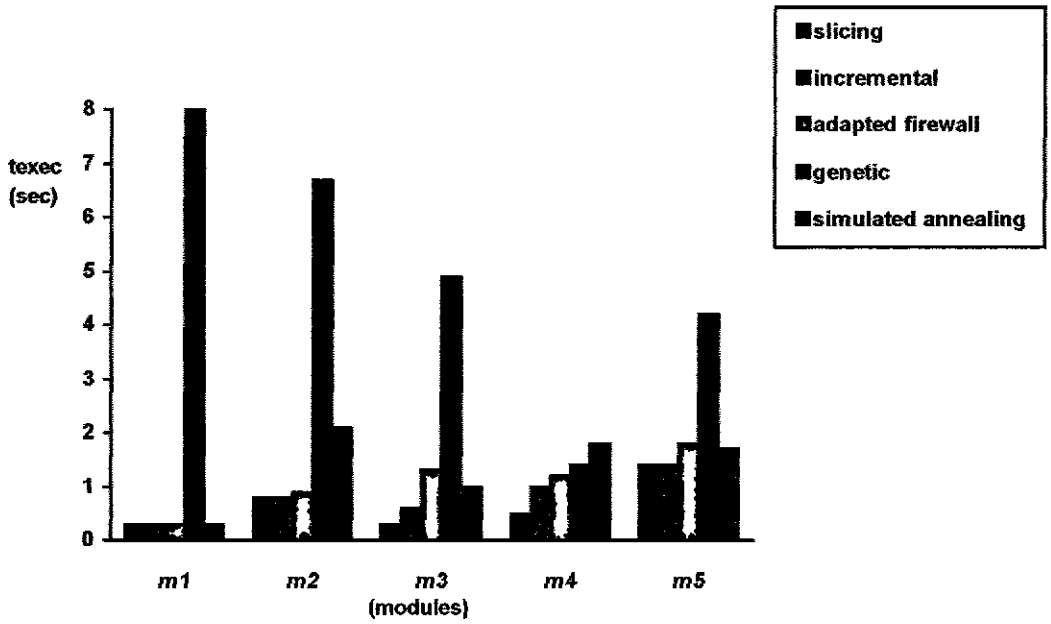


Figure 6.1. The running times of Slicing, Incremental, Adapted Firewall, Genetic, and Simulated Annealing algorithms for modules (m1_m5).

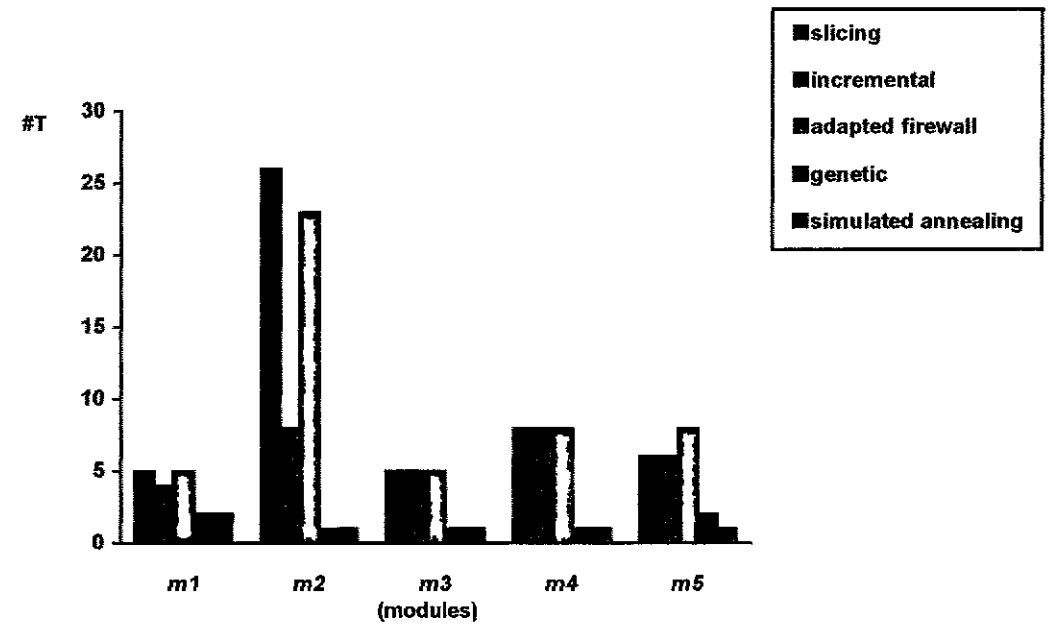


Figure 6.2. The number of test cases selected by Slicing, Incremental, Adapted Firewall, Genetic, and Simulated Annealing algorithms for modules (m1_m5).

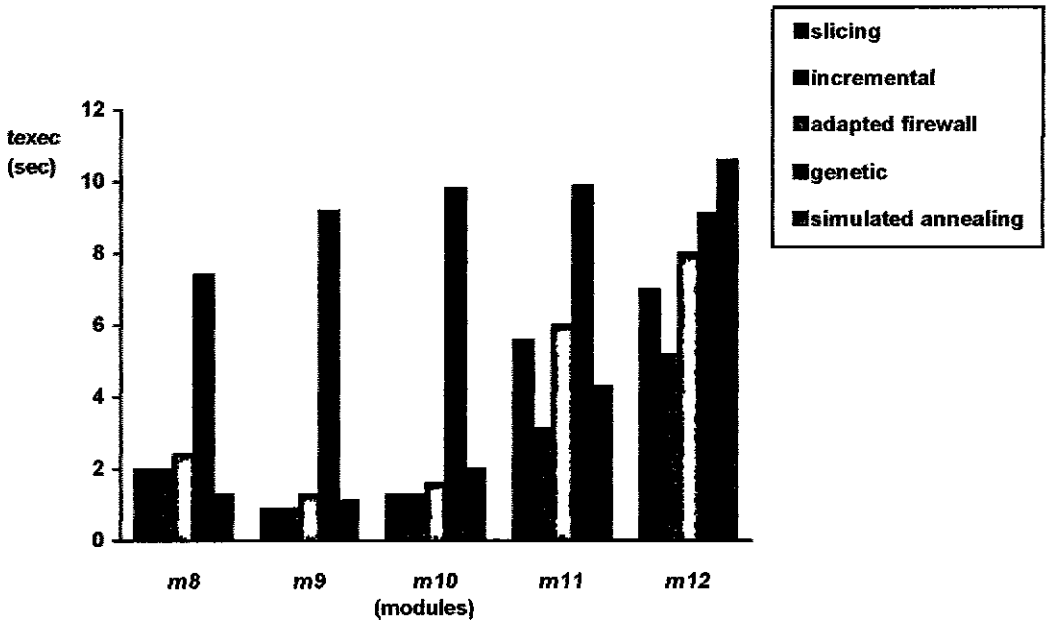


Figure 6.3. The running times of Slicing, Incremental, Adapted Firewall, Genetic, and Simulated Annealing algorithms for modules (m8_m12).

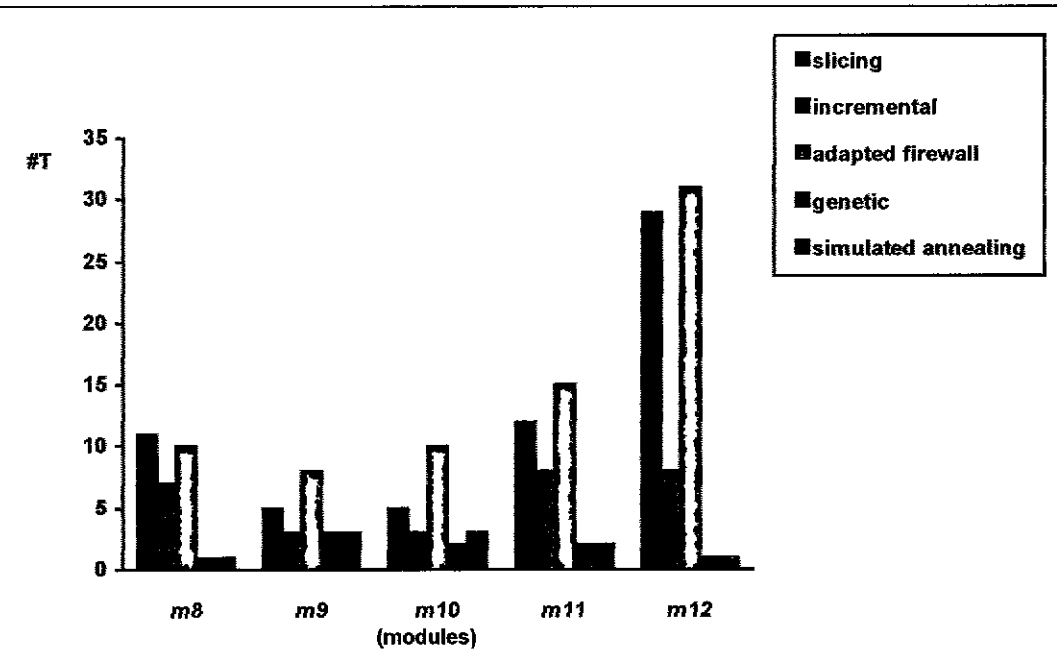


Figure 6.4. The number of test cases selected by Slicing, Incremental, Adapted Firewall, Genetic, and Simulated Annealing algorithms for modules (m8-m12).

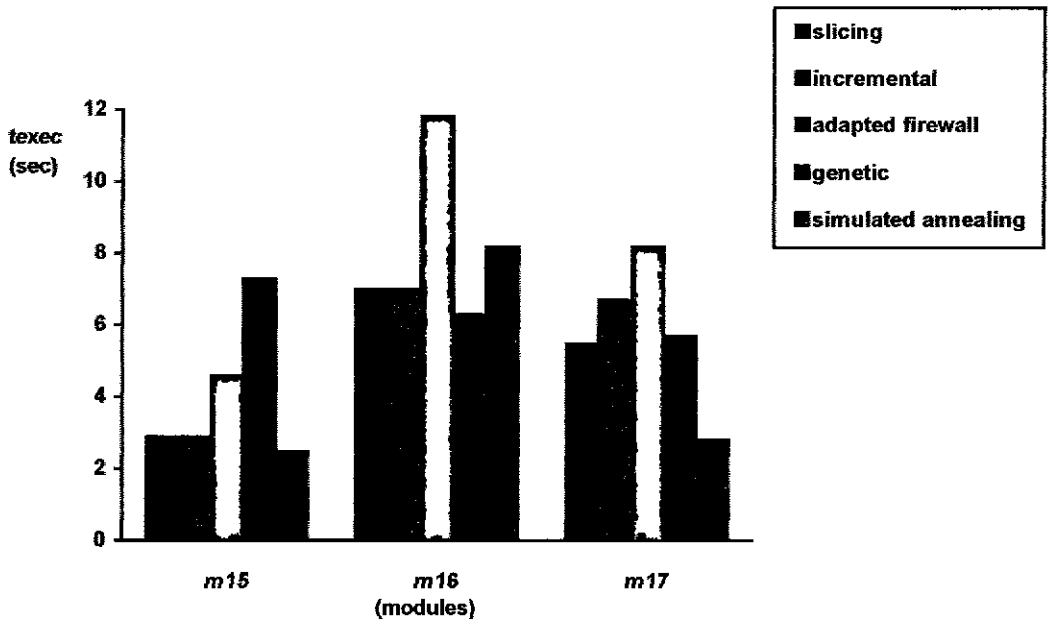


Figure 6.5. The running times of Slicing, Adapted firewall, Incremental, Genetic, and Simulated Annealing algorithms for modules (m15_m17).

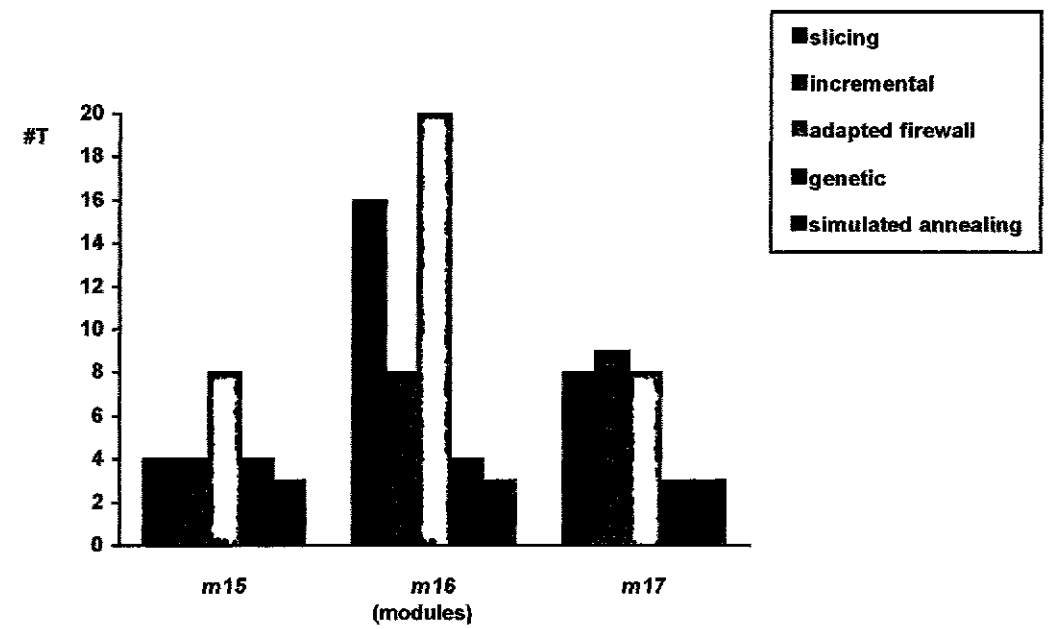


Figure 6.6. The number of test cases selected by Slicing, Adapted firewall, Incremental, Genetic, and Simulated Annealing algorithms for modules (m15_m17).

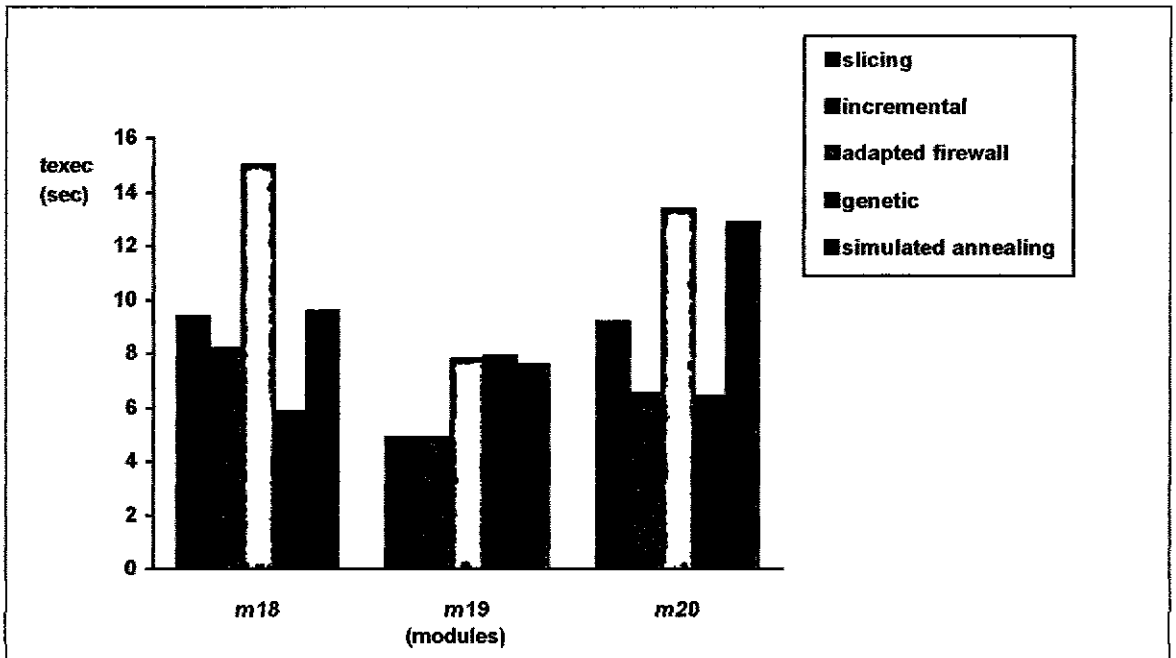


Figure 6.7. The running times of Slicing, Adapted firewall, Incremental, Genetic, and Simulated Annealing algorithms for modules (m18_m20).

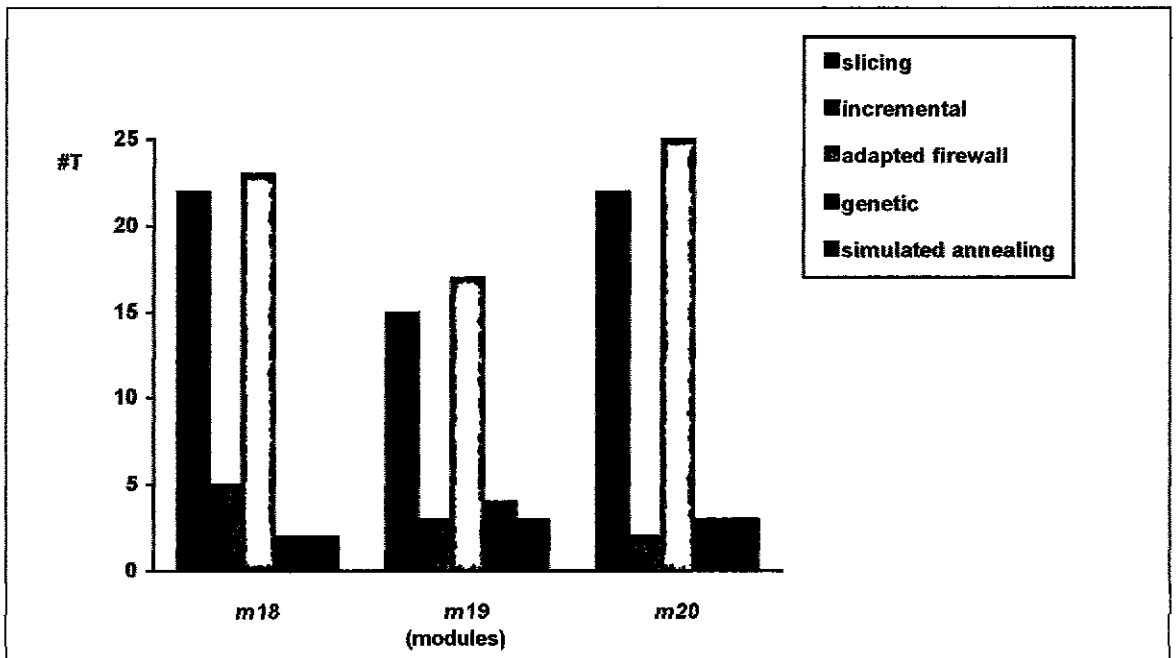


Figure 6.8. The number of test cases selected by Slicing, Adapted firewall, Incremental, Genetic, and Simulated Annealing algorithms for modules (m18_m20).

6.1.1. Efficiency and number of test cases to rerun

Table 6.2 shows that for small-size modules (m1 _ m14) , the slicing and adapted firewall algorithms exhibit almost a similar behavior in terms of execution time and the number of test cases to rerun. For some modules, they give a number of test cases to rerun close to the retest-all strategy. The incremental algorithm shows a better number of retests for similar execution times. The genetic and simulated annealing algorithms give the best number of test cases that must be rerun, although they are clearly slower.

For medium-size modules (m15 _ m20), the adapted firewall algorithm takes the largest time with a number of test cases to rerun close to that of the slicing algorithm. The incremental algorithm gives better results in the number of retests, at an execution time similar to that of the slicing algorithm. The best number of retests, is still offered by the genetic and simulated annealing algorithms, although at a slower execution.

The slicing algorithm and the adapted firewall algorithm at the unit level, are efficient in terms of memory and computational cost. The data-flow information needed in the two algorithms takes time in $O(n^2)$, where n is the number of nodes in the control-flow graph. However, there is no need to completely recompute data-flow information. As explained in the implementation in Chapter 4, only recomputation of the partial data-flow driven by the program changes is needed. On the other hand, the slicing algorithm makes use of the control-flow graph that should be calculated prior to the algorithms used. This would add some additional cost in terms of memory and cost depending on n.

The incremental algorithm uses a data-flow graph derived from the control-flow graph. Thus the time it takes to construct these graphs is at most $O(n^2)$ which is acceptable. The hard part is in the computation of static data dependencies. Unfortunately, precise static data dependencies becomes hard to compute when the program makes use of pointers, arrays, and dynamic memory allocations.

The genetic and simulated annealing algorithms may be data and computation intensive on large programs due to the calculations required for solving systems of linear equations.

On the other hand, all algorithms require test histories on a per statement basis, and most of them require the data-flow graph, thus the space requirement is much based on the program size and test suite size.

6.1.2. Precision and inclusiveness

Tables 6.4, 6.5, 6.6, 6.7 and 6.8 present the results obtained for precision and inclusiveness of the implemented algorithms. Table 6.3 explains the symbols used in these tables. Results of the tables are also shown in Figures 6.9 and 6.10.

M	Number of segments
N	Number of test cases
S_{mod}	Number of the modified segment
#T	Number of test cases to rerun
mr	Modification-revealing test cases
nmr	Non-modification-revealing test cases
m	Modification-revealing test cases selected
n	Non-modification-revealing test cases omitted
Prec%	Precision percentage
Inc%	Inclusiveness percentage

Table 6.3. Parameters used in Tables 6.4, 6.5, 6.6, 6.7, and 6.8.

Table 6.4 reveals results of precision and inclusiveness of the slicing algorithm.

	M	N	S_{mod}	#T	mr	nmr	m	n	Prec%	Inc%
m1	8	6	3	5	4	2	4	1	50	100
m5	14	24	13	6	10	14	4	12	86	40
m8	21	18	19	11	12	6	8	3	50	67
m9	24	18	9	5	10	8	5	8	100	50
m11	34	58	29	12	8	50	8	46	92	100
m13	40	20	21	7	10	10	5	8	80	50
m17	56	34	41	8	13	21	4	17	81	31
m19	60	90	30	15	35	55	10	50	91	28

Table 6.4. Precision and inclusiveness Results for the Slicing Approach.

Table 6.5 reveals results of precision and inclusiveness of the incremental algorithm.

	M	N	S_{mod}	#T	mr	nmr	m	n	Prec%	Inc%
m1	8	6	3	4	4	2	4	2	100	100
m5	14	24	13	6	10	14	6	14	100	60
m8	21	18	19	7	12	6	7	6	100	58
m9	24	18	9	3	10	8	3	8	100	30
m11	34	58	29	8	8	50	8	50	100	100
m13	40	20	21	3	10	10	3	10	100	30
m17	56	34	41	9	13	21	9	21	100	69
m19	60	90	30	3	35	55	3	55	100	28

Table 6.5. Precision and inclusiveness Results for the Incremental Algorithm.

Table 6.6 reveals results of precision and inclusiveness of the "adapted firewall" algorithm.

	M	N	S_{mod}	#T	mr	nmr	m	n	Prec%	Inc%
m1	8	6	3	4	4	2	4	2	100	100
m5	14	24	13	6	10	14	8	14	100	80
m8	21	18	19	7	12	6	8	4	66	66
m9	24	18	9	3	10	8	6	6	75	60
m11	34	58	29	8	8	50	8	43	86	100
m13	40	20	21	3	10	10	9	10	100	90
m17	56	34	41	9	13	21	7	20	95	53
m19	60	90	30	3	35	55	17	55	100	48

Table 6.6. Precision and inclusiveness Results for the Adapted firewall Algorithm.

Table 6.7 reveals results of precision and inclusiveness of the genetic algorithm.

	M	N	S_{mod}	#T	mr	nmr	m	n	Prec%	Inc%
m1	8	6	3	4	4	2	2	2	100	50
m5	14	24	13	6	10	14	2	14	100	20
m8	21	18	19	7	12	6	1	6	100	8
m9	24	18	9	3	10	8	3	8	100	30
m11	34	58	29	8	8	50	2	50	100	25
m13	40	20	21	3	10	10	3	10	100	30
m17	56	34	41	9	13	21	3	21	100	23
m19	60	90	30	3	35	55	4	55	100	11

Table 6.7. Precision and inclusiveness Results for the Genetic Algorithm.

Table 6.8 reveals results of precision and inclusiveness of the simulated annealing algorithm.

	M	N	S_{med}	#T	mr	nmr	m	n	Prec%	Inc%
m1	8	6	3	4	4	2	2	2	100	50
m5	14	24	13	6	10	14	1	14	100	10
m8	21	18	19	7	12	6	1	6	100	8
m9	24	18	9	3	10	8	3	8	100	30
m11	34	58	29	8	8	50	2	50	100	25
m13	40	20	21	3	10	10	3	10	100	30
m17	56	34	41	9	13	21	3	21	100	23
m19	60	90	30	3	35	55	4	55	100	8

Table 6.8. Precision and inclusiveness Results for the Simulated Annealing Algorithm

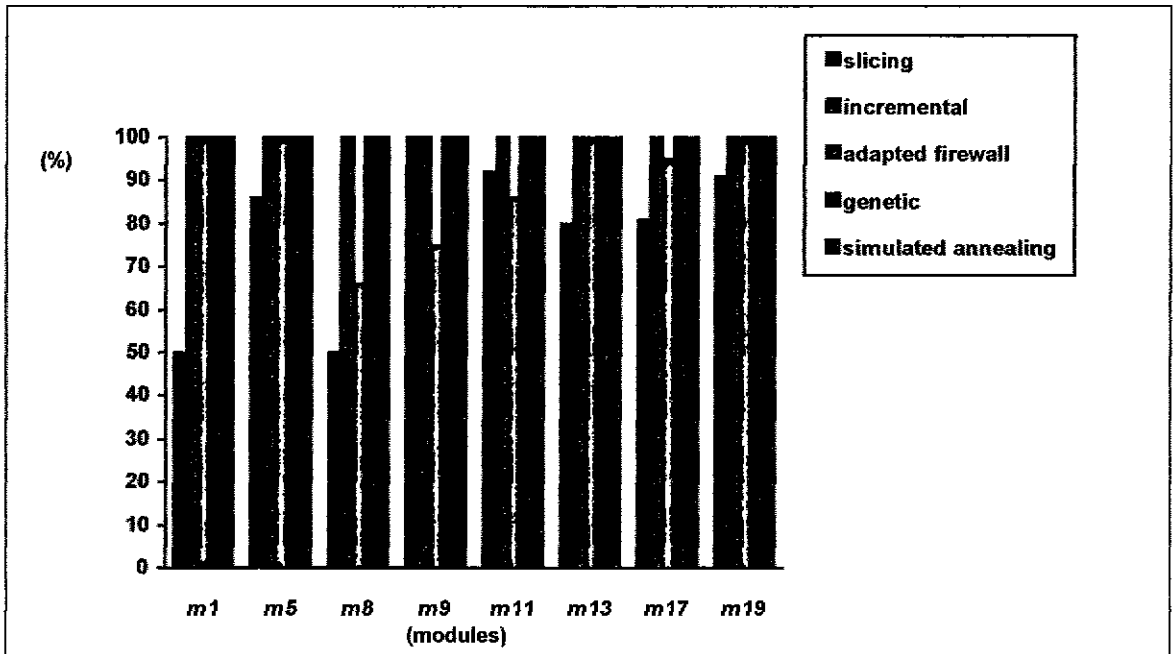


Figure 6.9. The *precision* of Slicing, Incremental, Adapted firewall, Genetic, and Simulated Annealing algorithms for modules presented in Tables 6.4 through 6.8.

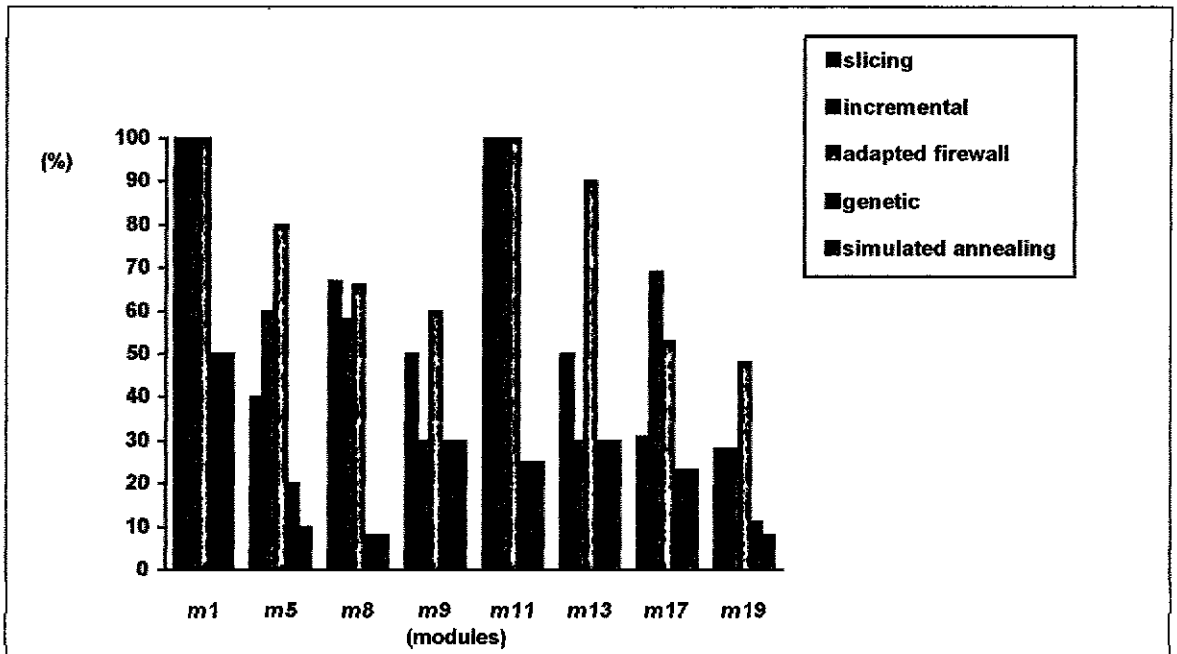


Figure 6.10. The *inclusiveness* of Slicing, Incremental, Adapted firewall, Genetic, and Simulated Annealing algorithms for modules presented in Tables 6.4 through 6.8.

The slicing and the adapted firewall algorithms give a good precision percentage. For some modules we can see that the precision percentage is not very high. This is due to the fact that these algorithms define also all the def-use pairs that are affected by the modification. This would result in selecting other test cases that are modification-revealing to the affected def-use pairs.

The incremental, genetic, and simulated annealing algorithms result in a very high percentage of precision as it is shown from the graphs. Tests that will not cause the modified program to produce different output are avoided.

The slicing and the adapted firewall algorithm select only modification-traversing tests by selecting tests that exercise definition-use pairs. However, the methods may fail to identify tests that execute modified output statements that contain no variable uses, although these statements may cause the program to produce different outputs. Also, the methods cannot detect tests that include deleted statements of a modified program. Thus, depending on the modification the inclusiveness percentage varies for the selected modules.

The precision percentage of the incremental algorithm also varies for the selected modules. The algorithm uses sets of techniques to determine the test cases in the regression test suite on which the new and the old programs may produce different outputs. Only these test cases need to be rerun. However, not all of these test cases are

selected. If several tests exercise a particular affected statement, only some of them are selected depending on the control-dependencies of the related statements.

On the other hand, genetic and simulated annealing algorithms exhibit less inclusiveness than the previous algorithms. If several tests exercise a particular affected statement, only one such test is selected. Some of the tests that are omitted may produce different output if executed.

6.2. Qualitative criteria results

6.2.1. User's parameter setting

In the slicing algorithm, the user is required to specify whether the change is a predicate change or a computation change.

The genetic algorithm requires parameters to be set by the user, such as the ranking range for reproduction, convergence threshold, and crossover and mutation rates. The annealing algorithm requires the parameters: convergence threshold, and initial probability for accepting perturbations.

6.2.2. Type of maintenance

All unit regression testing methods evaluated in this chapter are corrective regression testing where modifications are done only to correct an error report. On the other hand,

the firewall concept at the integration level supports the corrective regression testing and the perfective regression testing.

However, the slicing algorithm can be perfective too. Actions for different types of edits to enhance functionality have been provided. However, this will involve the need of the new and the old control flow graphs to define the definition-use pairs involving the definitions that are present after the program change. Thus this algorithm can be a perfective type of maintenance with an overhead of storage and computational cost in terms of new control flow graph and updates of data flow dependencies.

The minimization algorithms can be used as perfective maintenance, if actions such as, handling and storing the old and new control-flow graphs, are provided.

Additional actions and modifications are needed in order to support perfective maintenance in the adapted firewall algorithm at the unit level. The new control-flow graph along with the old are needed in order to identify the newly created definition-use pairs.

On the other hand, the incremental algorithm is a corrective and a perfective type of maintenance. Actions such as deletion and additions to enhance functionality at the unit level, can be handled.

6.2.3. Type of testing

The models used in the slicing, adapted firewall, genetic, and simulated annealing algorithms are based on the internal structure and logic of the program. The model presented in the incremental regression testing is structure and function based.

The firewall algorithm merges one module at a time to the set of previously tested modules. Functional and structural tests are used in this algorithm.

6.2.4. Level of testing

The slicing, incremental, and adapted firewall algorithms are applied at the unit level. Integration testing cannot be applied using slicing since it is based on identifying the definition-use pairs at the unit level. In incremental regression testing, computation of the execution slices can be computed at the function or module level. Then, we need to execute the new program on a test case only if a modified function or module was invoked during the original program's execution on that test case.

The firewall algorithm is an integration level of testing.

In the minimization approaches, segments can be replaced by modules and the genetic and simulated annealing algorithms can be applied at the program level [Fakih and Mansour 1996].

6.2.5. Type of approach

The slicing and the adapted firewall algorithms, belong to the set of coverage methods. They select tests that cover affected pairs.

The incremental algorithm is a safe regression test selection method. It aims at selecting tests that will cause the modified program to produce different output than the original program.

Genetic and simulated algorithms belong to the minimization regression test selection method. If several tests exercise a particular modified statement, only one such statement is selected.

The firewall algorithm belongs to the coverage algorithms that rely on coverage criteria. It selects all tests that should be integration-tested.

6.2.6. Global variables

Leung and White, showed that global variables can be treated as extra parameters. They extended this idea and applied the firewall algorithm to testing global variables in [Leung and White 1992].

For all algorithms discussed in this chapter, the set/use matrix that identifies the status of all local and global variables along with the arguments within the remote should be used. Moreover, as all unit testing algorithms rely solely on the data-flow dependencies, and the global variables create such dependencies between modules, the set/use matrix can be applied to the algorithms in order to study the effects of such variables. However, the correct testing of such variables is usually performed during integration testing when they are actually being used. So, using this matrix allows us to identify the global variables affected. Then, a global variable set/use matrix at the modules level should be used in order to identify which modules are affected by a change.

6.3. Summary of results

Table 6.9 summarizes the comparison results for the quantitative and qualitative criteria. We note that our assessment is based on the following considerations:

(a) Larger size modules are more important for conclusions, since they are more realistic.

(b) Since the test cases were manually developed, it was not possible to run experiments that were statistically sound, especially for the execution time.

(c) Execution time results are based on an overall assessment, although the results are not consistent for all cases. The terms used for algorithms' speed are based on comparing these algorithms with each other.

Algorithm		Slicing	Incremental	Adapted firewall	Firewall at Integration	Genetic	Simulated Annealing
Execution time for small-size modules		fast	fast	fast	----------	accep-table	accep-table
Execution time for medium-size modules		accep-table	fast	slow	----------	accep-table	slow
Number of test cases to rerun		accep-table	good	accep-table	----------	very good	very good
Precision		high	very high	high	----------	very high	very high
Inclusiveness		high	high	high	----------	low	low
User's parameter setting		required	not required	not requ-ired	not requ-ired	required	required
Type of mainte-nance	corre-ctive	yes	yes	yes	yes	yes	yes
	perfe-ctive	can be w/ modifica-tions	yes	can be w/ additions	yes	can be w/ additions	can be w/ additions
Type of testing		structural	structural functional	structural	structural functional	structural	structural
Level of testing	Unit	yes	yes	yes	----------	yes	yes
	Integ-ration	cannot be applied	can be applied w/ modification	cannot be applied	yes	can be applied w/ addition	can be applied w/ addition
Type of algorithm		coverage	safe	coverage	coverage	minimi-zation	minimi-zation
Global variables		could be identified	could be identified	could be identified	could be tested	could be identified	could be identified

Table 6.9. Classifications of the algorithms discussed.

Chapter 7

Conclusions and Further Work

A comparative study, based on quantitative and qualitative criteria, for slicing, incremental adapted firewall, and minimization regression testing methods have been presented. These methods have been quantitatively evaluated based on efficiency, number of test cases that must be rerun, precision, and inclusiveness using a set of small and medium size modules for which the codes were written and the graphs were generated manually along with the test cases tables. They have been qualitatively compared based on the type of testing, level of testing, and type of maintenance, an algorithm can handle. Also, algorithms have been compared based on user's parameter setting requirements, and the ability to regression test global variables.

For small-size modules the slicing incremental, and adapted firewall algorithms exhibit a better behavior in terms of execution time compared to the genetic and simulated annealing algorithms. For medium-size modules, the least execution time is offered by the slicing and incremental algorithms, while the adapted firewall algorithm takes the longest time.

The comparison among these algorithms shows that genetic and simulated annealing yield the best results in terms of number of test cases to rerun, followed by incremental, slicing, and then adapted firewall.

The choice among the five algorithms is dependent on the regression testers' requirements. To test all affected definition-use pairs, despite spending more time on regression testing activity, the algorithms to choose are slicing and adapted firewall. To choose the minimum number of test cases to provide full testing coverage, the selection should be based on genetic or simulated annealing, although they require more running time. The incremental algorithm is the best choice among these algorithms for selecting a number of test cases whose outputs may be affected. Also, the incremental algorithm selects these test cases at a fast execution time.

Furthermore, all algorithms presented in this thesis apply to corrective maintenance and structural type testing. However, modifications where functional type of testing is needed can only be regression tested with the incremental algorithm at the unit level and with the firewall algorithm at the integration level.

Based on the work presented in this thesis, further research tasks can be pursued. Firstly, additional actions can be added or modified to the algorithms in order to handle perfective type of maintenance. Secondly, modifications may be made to the algorithms so they can be applied at the integration level. Thirdly, the quantitative criteria experiments need to be done for large-size modules. This could be accomplished using tools that can generate the control-flow and data flow graphs and the associated test cases tables for a given large-size module, in order to be used as input for the algorithms. Fourthly, the global variable set/use matrix can be used to monitor the triple effects of modifications by reflecting the status of each global variable, argument, parameter, and

local variable. The global variables would be included in the matrix to facilitate the identification of modules in which the global variables are used. Then, applying the algorithms at the integration level, regression test global variables can be done using the set/use matrix.

Finally, the comparative study based on qualitative criteria can be done to the algorithms for perfective type of maintenance, for integration level of testing, and for global variables regression testing.

References

Agrawal H., Horgan J.R., and Krauser E.W. 1993. Incremental Regression Testing. *Proceedings of the Conference on Software Maintenance,* September, 348-357.

Benedusi P., Cimitile A., and De Carlini U. 1988. Post-maintenance testing based on path change analysis. *IEEE Conference on Software Maintenance*, 352-361.

Chen Y., Rosenblum D.S., and Vo K. 1994. TestTube: A system for selective regression testing. *Int. Conference Software Eng.,* 211-220.

Fakih K., and Mansour N. 1996. A genetic algorithm for corrective retesting. *Lebanese Scientific Research Reports,* Vol. 1, No. 1, January, 5-19.

Fischer K., and Chruscicki A. 1981. A methodology for retesting modified software. *National Telecommunications Conf.*, November, B6.3.1-B6.3.6.

Friedman T., Kirschenbaum M., Narayanswamy V., Oha M., Piwowarski P., and White L. 1993. Test manager : a regression testing tool. *Proc. Conference Software Maintenance,* 338-347.

Gallager K.B., and Lyle J.R. 1988. Using program decomposition to guide modifications. *Proc. IEEE Conf. on Software Maintenance*, 265-269.

Gallager K.B., and Lyle J.R. 1991. Using program slicing in software maintenance. *IEEE Trans. on Software Eng.*, 751-761.

Harrold M.J., Gupta R., and Soffa M.L. 1992. An approach to regression testing using slicing. *Proc. IEEE Conf. Software Eng.,* 299-308.

Harrold M.J., Gupta R., and Soffa M.L. 1993. A methodology for controlling the size of a test suite. *ACM Transaction on Software Engineering and Methodology*, July, 270-285.

Harrold M.J., and Soffa M.L. 1988. An incremental approach to unit testing during maintenance. *Proc. IEEE Conf. on Software Maintenance*, October, 362-367.

Hartmann J., and Robson D.J. 1988. Approaches to regression testing. *Proc. IEEE Conf. on Software Maintenance(CSM-88)*, October, 368-372.

Hartmann J., and Robson D.J. 1989. Revalidation during the software maintenance phase. *Proc. IEEE Conf. on Software Maintenance*, 70-79.

Hartmann J., and Robson D.J. 1990. Techniques for selective revalidation. *IEEE Software*, January, 31-38.

Leung H.K.N., and White L. 1989. Insights into regression testing. *Proc. IEEE Conf. on Software Maintenance*, 60-69.

Leung H.K.N., and White L. 1990a. A study of integration testing and software regressin at the integration level. *Proc. IEEE Conf. on Software Maintenance,* 290-300.

Leung H.K.N., and White L. 1990b. Insights into testing and regression testing global variables. *Proc. IEEE Conf. on Software Maintenance*, 209-221.

Leung H.K.N., and White L. 1991. A cost model to compare regression test strategies. *Proceedings of the Conference on Software Maintenance,* 201-208.

Leung H.K.N., and White L. 1992. A firewall concept for both control-flow and data-flow in regression integration testing. *Proc. Conference Software Maintenance,* November, 262-271.

Mansour N., and Goel A.L. 1994. Simulated annealing for optimal regression testing. *Science Week,* November, 4-11.

Rothermel G., and Harrold M.J. 1993. A safe, efficient algorithm for regression test selection. *Proc. Conference Software Maintenance,* September, 358-367.

Rothermel G., and Harrold M.J. 1994. A framework for evaluating regression test selection techniques. *Int. Conference Software Eng.,* 201-210.

Yau S., and Kishimoto Z. 1987. A method for revalidation modified programs in the maintenance phase. *Proc. COMPSAC*, 272-277.