207
C.1

# REDUCTION-BASED METHODS AND METRICS FOR SELECTIVE REGRESSION TESTING

by

**RAMI K. BAHSOON**

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

**Thesis Advisor: DR. Nasha't Mansour**

**Department of Computer Science**
**LEBANESE AMERICAN UNIVERITY**
**June 2000**

**LEBANESE AMERICAN UNIVERSITY**

**GRADUATE STUDIES**

We hereby approve the thesis of

**Rami Bahsoon**

Candidate for the *Master of Science* degree*.

Signatures Redacted

Signatures Redacted

Signatures Redacted

date July 5, 2000

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

Signatures Redacted

# REDUCTION-BASED METHODS AND METRICS FOR SELECTIVE REGRESSION TESTING

## ABSTRACT

by

RAMI K. BAHSOON

Selective regression testing attempts to choose an appropriate subset of test cases from among a previously run test suite for a software system, based on information about the changes made to the system to create new versions.

In this thesis, we address two major problems in selective regression testing: the regression test selection problem and the coverage identification problem. To address the former problem, we propose three reduction-based selective regression testing methods that reduce the number of selected test cases for retesting the modified software by omitting redundant tests from the initial test suite. But, one method, referred to as Modification-Based Reduction version 1 (MBR1), selects a reduced number of test cases based on the modification made and its effects in the software. A second method, referred to as Modification-Based Reduction version 2 (MBR2) improves MBR1 by omitting tests that do not reach the modification. A third method, referred to as Precise Reduction (PR), further reduces the number of test cases selected by omitting all non-modification-revealing tests from the initial test suite.

To approach the latter selective retesting problem, we suggest two McCabe-based regression test selection metrics that could be also extended to address the test selection problem. These metrics are the Reachability regression Test selection McCabe-based metric (RTM), and dataflow Slices regression Test McCabe-based metric (STM). The suggested metrics help in monitoring test-coverage adequacy, reveal any shortage or redundancy in the test suite, and assist in identifying where additional tests may be required for retesting.

We empirically compare MBR1, MBR2, and PR with three reduction and precision-oriented methods on 60 test-problems. The results show that PR selects the least number of test cases most of the time and omits non-modification-revealing test cases all the time. We illustrate a typical application of our suggested metrics using the 60 test-problems on two coverage-oriented selective regression testing methods.

**KEYWORDS:** Software maintenance, selective regression testing, test suite reduction, retesting metrics, McCabe's cyclomatic complexity, test coverage.

*To my parents*

# ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Nasha't Mansour for his guidance throughout my M.S. studies. Dr. Mansour, this is the result of your simple, attractive, and intelligent style in introducing any matter that make one curious enough to get involved. Thanks for your extremely gentle and polite character that nourishes a trusty environment, which is vital for any progress and motivation. I can tell that I have learnt a lot from your profound thinking and knowledge.

Thanks is also due to Dr. Ramzi Haraty and Dr. George E. Nasr for being on my Thesis committee and for their valuable comments and suggestions. Specifically, I highly appreciate Dr. Haraty's time in being an extremely patient reader.

I would like to express my sincere gratitude to Dr. A. Kabbani and the LEBANESE AMERICAN UNIVERSITY for helping and supporting in achieving our goals through providing excellent education in a friendly environment.

Also, thanks is due to brothers Bernard Hobeika, Vittorio Re, and Mr. Gabby Dahan of COLLEGE FRERES DU SACRE COEUR for understanding and approving a leave to accomplish this work.

Finally, I would like to thank my family for their long support and for setting education as one of highest priorities.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

The software maintenance phase of the software development life cycle involves activities that are carried out after the development of the software and during its operational life. In this phase, programs are modified as a result of errors, changes in the user requirements, or changes in the software and hardware environments. During such modifications, new errors may be introduced, causing unintended adverse side effects in the software. Thus, necessitating the software to be tested again.

Regression testing can be progressive or corrective (Leung and White 1989). The former involves retesting major changes to the program's specifications. The latter is performed on specifications that essentially remain unchanged, so that only modifications that do not affect the overall program structure require retesting. For corrective regression testing, it might be costly to repeat the whole set of test cases used in the initial development of the program and unreliable to choose a random subset of these test cases. Selective retest techniques attempt to reduce the cost required to retest a modified program by selectively reusing tests and selectively retesting the modified program. These techniques address two major problems (Rothermel and Harrold, 1997): (1) the regression test selection problem- the problem of selecting tests from an existing test suite and (2) the coverage identification problem, the problem of determining where additional tests may be required.

Therefore, if $TS=\{t_1, t_2,\ldots, t_n\}$ is the set of $n$ test cases used in the initial development of a program $P$, the regression test selection problem requires that a subset of test cases $R$ be selected from $TS$ for rerunning on the modified program $P'$ with the objective to provide confidence that no adverse effects have been caused by the modification. The coverage identification problem requires identifying portions of $P'$ that require additional testing.

The test case selection is approached by strategies such as safe, minimization, or coverage. Safe approaches require the selection of every existing test case that exercise any program element that could be possibly affected by a given program change. Minimization approaches attempt to select the smallest set of test cases necessary to tests affected program elements at least once. Coverage approaches attempt to assure that some structural coverage criterion is met by the test cases that are selected. Because in practice a coverage criterion is applied to select a single test case satisfying each coverage requirement induced by the criterion, coverage approaches can be viewed as special types of minimization approaches (Rosenblum and Weyuker, 1997).

The selection of suitable test cases can be made in different ways and a number of selective regression testing methods have been proposed. These methods are based on different objectives and techniques such as: procedure and class firewalls (Leung and White, 1992; Hsia et al., 1997); semantic defferencing (Binkley, 1997); textual differencing (Vokolos and Frankl, 1997); slicing-based dataflow technique (Gupta et al., 1996); and safe algorithm based on program's control graph (Rothermel and Harrold, 1997). Furthermore, some empirical studies of these methods have been reported (Baradhi and Mansour, 1997; Graves et al., 1998) and some software tools based on some of them have been constructed (Vokolos and Frankl, 1997; von Mayrhauser and Zhang, 1999).

In particular, Mansour and El-Fakih (1999) have proposed using an optimization formulation of the selective retesting problem and a Simulated Annealing (SA) algorithm for minimizing the number of selected test cases. Also, Harrold, Gupta, and Soffa (1993) have suggested a methodology for reducing the size of a test suite, which can be used for reducing the number of selected test cases. We refer to this method as the Reduction (RED) algorithm. Furthermore, Agrawal, Horgan, and Krauser (1993) have proposed slicing algorithms (SLI) that select test cases whose output may be affected by the modification made to the program. The first two methods are reduction-oriented: RED aims to reduce the number of selected retests regardless of where the modification is done in the program. SA takes the modification into account and aims to minimize the number of selected retests. The SLI algorithm concentrates on including all the test cases whose dynamic/relevant program slices contain the modified component. This often leads to a large number of selected retests by SLI.

In this thesis, we restrict our attention to two major problems of selective retesting. These are: (1) the problem of selecting a reduced number tests from an existing test suite and (2) the coverage identification problem- the problem of determining where additional tests may be required.

To address the former problem, we propose three reduction-based selective regression testing methods. The first method, referred to as Modification-Based Reduction version 1 (MBR1), improves the RED algorithm by accounting for the location of the modification made in the program and its effects. The second method, referred to as Modification-Based Reduction version 2 (MBR2) improves MBR1 to only select test cases that execute the modification. The third method, referred to as Precise Reduction (PR), uses slicing in a similar way to the SLI algorithm to determine the useful test cases and applies a reduction procedure to reduce the final number of selected retests. We use 60 problems to empirically evaluate MBR1, MBR2, and PR and compare with SA, RED, and SLI.

We approach the latter problem by suggesting two McCabe-based regression test selection metrics that could be also extended to address the test selection problem. These are Reachability regression Test selection McCabe-based metric (RTM), and dataflow Slices regression Test McCabe-based metric (STM). RTM is an upper bound metric that derives its measures from reachability information and provides an upper indication of paths that must be tested for being potentially affected by the modification, and by implication, an upper bound of tests to rerun for exercising these paths at least once. STM is a data flow McCabe-based variable dependent metric; it derives its measure from information related to slices affected by the modification and computes two bounds: an upper and a lower bound of the number of retests to test the affected definition-use pairs. Furthermore, these metrics help in monitoring test-coverage adequacy, revealing any shortage or redundancy in the test suite, and assist in identifying where additional tests may be required. We use 60 problems to try our suggested metrics on two coverage-oriented selective regression testing methods. These are Gupta, Harrold, and Soffa's DataFlow (DF) algorithms (Gupta et al., 1996), and Leung and White's Segment-FireWall (SFW) algorithm (Leung and White, 1992).

This thesis is organized as follows. Chapter 2 describes the regression test selection problem and the program modeling assumptions. Chapter 3 presents the two proposed versions of Modification-Based Reduction. Chapter 4 proposes Precise Reduction. Chapter 5 suggests the McCabe-based regression test selection metrics. Chapter 6 describes the experimental approach and results. Chapter 7 contains our conclusions and suggestions for further work.

# CHAPTER 2

# REGRESSION TEST SELECTION PROBLEM AND PROGRAM MODELS

Software maintenance activities can account for as much as two-thirds of the overall cost of software production (Rosenblum and Weyuker, 1997). One necessary maintenance activity, regression testing, is performed on modified software to provide confidence that the software behaves correctly and that modifications have not adversely impacted the software's quality. Regression testing is expensive; it can account for as much as one-half of the software maintenance (Leung and White 1989). During regression testing, an established suite of tests may be available for reuse. One regression testing strategy reruns all such tests, but this *retest-all* approach may consume inordinate time and resources. *Selective retest* techniques, in contrast, attempt to reduce the time required to retest a modified program by selectively reusing tests and selectively retesting the modifed program.

In this chapter, we present the regression test selection problem, a regression testing problem. In the subsequent sections, we brifely describe the selective regression testing subproblems, steps of a typical selective regression testing technique, and cost effectiveness considerations. We also present the program modeling assumptions.

## 2.1 SELECTIVE REGRESSION TESTING

Selective regression testing attempts to choose an appropriate subset of test cases from a previously run test suite for a software system, based on information about the changes made to the system to create new versions. The intuition is that if, instead of rerunning the entire test suite (the so-called retest all strategy), a systematically–selected subset is chosen to be run, then substantial resources will be saved due to the limited size of test suite (Rosenblum and Weyuker, 1997). Most of the selective retest techniques attempt to reduce the cost of regression testing by identifying portions of the modified program or its specification that should be tested.

Let $P$ be a procedure or a program, $P'$ be a modified version of $P$, and $TS$ be the test suite created to test $P$. A typical selective retest technique proceeds as follows:

(1) Select $R \subseteq TS$, a set of tests to execute on $P'$, the modified version of $P$.

(2) Test $P'$ with $R$, establishing $P'$ 's correctness with respect to $R$.

(3) If necessary create $R'$, a set of new functional or structural tests for $P'$.

(4) Test $P'$ with $R'$, establishing $P'$ 's correctness with respect to $R'$.

(5) Create $R''$, a new test suite and test history for $P'$, from $R$, $R'$, and $R''$.

In performing these steps, a selective retest technique addresses several problems (Rothermel and Harrold, 1997). Step (1) involves the *regression test selection problem*. Step (3) addresses the *coverage identification problem*: the problem of identifying portions of $P'$ that require additional testing. Steps (2) and (4) address the *test suite execution problem*: the problem of efficiently executing tests and checking test results for correctness. Step (5) addresses the *test suite maintenance problem*: the problem of updating and storing test information.

Although each of these problems is significant, we restrict our attention to two major problems: (1) the problem of selecting tests from an existing test suite and (2) the problem of determining where additional tests may be required.

The first problem, referred to as the test selection problem, assumes that a test suite, *TS*, used in the initial development of the program; a set of test case requirements $r_1$, $r_2$, …$r_n$; and subsets of *TS*, $T_1$, $T_2$,…, $T_n$, one associated with each of the $r_i$s such that any one of the test cases $t_j$ belonging to the $T_i$ can be used to test $r_i$ are saved and that a table of test case-requirement coverage can be determined. After a program is modified, the regression test selection problem requires finding a subset *R* of test cases from *TS* that test for the $r_i$s impacted by the modification and satisfy criteria such as minimum-cardinality of the obtained subset or testing different data effects.

The second problem, refered to as the coverage identification problem, addresses the problem of identifying portions of the modified program that require additional testing. This, if necessary, requires creating $R'$, a set of new functional or structural tests for $P'$, and updating *TS*.

The former is addressed in chapters 3 and 4 by proposing reduction-based selective retesting methods. An approach to the former and the latter based on McCabe cyclomatic complexity is suggested in chapter 5.

## 2.2 COST-EFFECTIVENESS OF SELECTIVE RETEST TECHNIQUES

Leung and White (1991) show that a selective retest technique is more economical than the retest-all technique only if the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that the selective retest technique omits. The various factors affecting the cost include, but are not limited to, CPU time, disk space, effort of testing personnel, the cost of business opportunities gained or lost through increased or reduced testing.

## 2.3 PROGRAM MODELING USING CONTROL FLOW

We assume that a program under test is modeled by a control flow graph with *n* nodes, where each node represents a program segment, which corresponds to a control statement or to a contiguous sequence of assignment statements. Those *n* nodes constitute a set of requirements $\{r_1, r_2,...,r_n\}$. Also it is assumed that a set of *n* test-requirements *TS*, $T_1, T_2,\ldots,T_n$ , one associated with each of the $r_i$s such that any one of the test cases $t_j$ belonging to the $T_i$ can be used to test $r_i$, is determined and saved.

## 2.4 PROGRAM MODELING USING DATA AND CONTROL FLOW

For selective regression testing techniques that require slicing data flow-based analysis, we assume that the program model presented in the previous section is extended to keep data flow information for relevant segments/nodes. This includes the definitions and uses of the variables in the program statements within segments. Uses are classified as either computation uses (c-uses) or predicate uses (p-uses) according to whether a variable is used in a computation or in a predicate statement.

## 2.5 REACHABILITY INFORMATION

The control flow graph representation enables us to derive information about the reachability of the program segments from other ones to determine the impact of change. Such information is needed to determine all requirements that might be potentially affected by the modification. For a program with $n$ program segments, the segment reachability matrix, $S$, generated from the control flow graph, is a 0-1 $nxn$ matrix that describes both the direct and indirect interconnections between the various program segments. $S$[i, j]=1 indicates that segment $S_j$ is directly/indirectly reachable from $S_i$, where 0 indicates otherwise. Given the segment $k$ is modified, segments potentially affected by the modification are obtained by performing logical OR operation on the $k$-th row and the $k$-th column of the modified segment in reachability matrix.

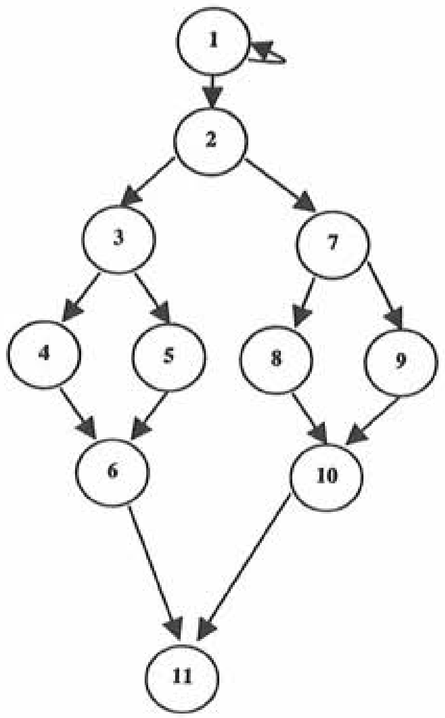

*Figure 2.1. Control flow graph G used to derive reachability matrix S*

For example, the control flow graph G of Figure 2.1 translates into $S$, the reachability information matrix depicted in Table 2.1. Upon modifying $S_4$, logical OR operation on entries of the fourth row and fourth column gives that besides $S_4$, segments $S_1$, $S_2$, $S_3$, $S_6$, and $S_{11}$ are potentially affected.

*Table 2.1. Reachability information matrix*

| *S* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

# CHAPTER 3

# MODIFICATION-BASED REDUCTION

In this chapter, we propose two selective retest methods that orient the reduction technique, RED, presented in (Harrold et al., 1993) to address the regression test selection problem by including modification information. The two methods are referred to as MBR1 (Modification-based Reduction version 1) and MBR2 (Modification-based Reduction version 2). MBR1 improves the RED algorithm by using reachability information to only account for the modification and its impact in the reduction test selection process. MBR2 uses reduction to only select test cases that execute the modified requirement.

## 3.1 BACKGROUND

The reduction technique, RED, is a test suite management technique that is capable to manage and control the size of the test suite by eliminating both redundant and obselete tests from the test suite with the objective of obtaining a minimum representative set of test cases.

The algorithm requires an association between the initial test cases (in *TS*) and the requirements of the program. For a particular program, a test selection criterion translates into a set of requirements whose satisfaction provides the desired measure of coverage with respect to that criterion. The RED algorithm aims to construct a minimum subset of test cases, *R*, from *TS* that can still satisfy all the given requirements of the considered criterion.

Given *TS*, a set of requirement $\{r_1, r_2, ..., r_n\}$, and subsets of *T*, $T_1, T_2, \ldots, T_n$ such that the test cases in $T_i$ cover the requirement $r_i$. To select the test cases for *R*, the RED algorithm first considers all test cases that occur in single element $T_i$s and includes them in *R*. Then, it marks all the $T_i$s containing these selected tests cases. Next, all unmarked $T_i$'s of cardinality two are considered. Repeatedly, the test cases that occur in the maximum number of such subsets are chosen and added to *R*. Again, all unmarked $T_i$s that contain the added test case are marked. This process is repeated for all unmarked $T_i$'s of cardinality 3, 4, ..., Max-Card, where Max-Card is the maximum cardinality of the $T_i$'s.

The algorithm can provide a reduction in the number of test cases to rerun to validate a changed program. However, RED is modification independent; it does not explicitly target the modification and count for its effects in the reduction process. In the subsequent sections, we orient reduction to count for such and address the test selection problem. Chapter 4 also works on this problem.

## 3.2 MODIFICATION-BASED REDUCTION

MBR1 and MBR2 revise the test selection process of RED to account for modification and its impact on the reduction process. Upon modifying a particular requirement(s), the change does not necessarily affect all other requirements. This implies that it may be unnecessary and costly to consider the unaffected requirements and include tests that exercise those requirements. The two versions assume that the program under test is modeled by control flow graph discussed in section 2.3.

### 3.2.1 Modification-Based Reduction version 1 (MBR1)

Our approach to determine the requirements that might be impacted by the change uses reachability information derived from the control-graph discussed in section 2.5. A requirement that is not reachable from/reaches the modified one(s) is not potentially affected by the modification and need not be tested. The MBR1 method uses the derived reachability information to select a subset *R* of test cases from *TS* that satisfies $r_i$'s reachable from/reach the modified requirement. The selected set *R*, with test cases that satisfy the affected $r_i$'s, must contain at least one test case from each $T_i$ associated with the affected $r_i$ to ensure that an adequate coverage of the changed and potentially affected requirements is attained. Such approach aims to potentially reduce the selected regression test suite by eliminating tests covering almost unaffected parts along with redundant tests. A redundant test case is one that covers the same structural or functional requirement as another test case in the test suite.

#### 3.2.1.1 MBR1 input/output

The MBR1 inputs are: the set of requirements $\{r_1, r_2,...,r_n\}$; a set of affected requirements *AR* subset of $\{r_1, r_2,...,r_n\}$, such that any requirement $r_j$ in *AR* is said to be reachable from/reaches the modified requirement(s); and subsets of

*TS*, $T_1, T_2, .., T_n$ such that the test cases $t_j$ in $T_i$ cover the requirement $r_i$. MBR1 outputs *R*, a selective set of tests that covers affected requirements in *AR*.

### 3.2.1.2 MBR1 steps

To construct *R*, MBR1 initiates the reduction process by neglecting and marking as unaffected every requirement $r_k$ not in *AR* where k in [1…*n*]. The associated tests $T_k$'s of $r_k$'s are excluded. For the unmarked requirements, the MBR1 then selects and includes in *R* all the singletons $T_i$'s that test for the affected requirements and marks all $T_i$'s containing those test cases as satisfied. Next, all unmarked $T_i$'s of cardinality two are considered. The test case occurring in the maximum number of $T_i$ is chosen and added to the *R* set. Again, all unmarked $T_i$'s containing these test cases are marked. This process is repeated for $T_i$'s of cardinality 3, 4, …, Max-Card, where Max-Card is the maximum cardinality of the $T_i$'s. When examining the $T_i$'s of size *n*, there may be a tie because several test cases covering the affected requirements occur in the maximum number of $T_i$'s of that size. In this case, $T_i$'s with cardinality (*n*+1) are examined. The test case that occurs in the maximum number of $T_i$'s of cardinality (*n*+1) is chosen. If a decision can not be made, the $T_i$'s with greater cardinality are examined and finally a random choice is made. Figures 3.1 and 3.2 outline the MBR1 steps and algorithm respectively.

```
Step 1: Determine affected requirements using reachability information
        Neglect as unaffected every requirement $r_k$ not in AR; k in [1…n]
        Mark $T_k$'s of $r_k$'s

Step 2: Select tests from singletons associated with affected requirements and
        mark affected requirements satisfied with selection

        For unmarked Ti's of cardinality =1 Do
           Select and include in R singletons $T_i$'s
           Mark $r_i$'s associated with singletons $T_i$'s as satisfied
           For unmarked $T_j$'s of various cardinalities Do
              If $T_i \cap T_j \neq \emptyset$ Then
                 Mark $r_j$ as satisfied

Step 3: Select tests from unmarked suites with higher cardinality and mark
        affected requirements satisfied with the selection

        While n < Max-Card Do
           Get $t_j$'s in the maximum of unmarked $T_i$'s of cardinality equal to n
           If the number of returned $t_j$'s greater than 1 Then
              Set Tie to true
              While Tie Do
                 Get $t_j$ the maximum of tied tests in unmarked $T_i$'s of
                    cardinality(n+1)
                 If the number of returned tests equal 1 Then
                    Set Tie to false
              EndWhile

        Select and include $t_j$ in R
           For unmarked $T_j$'s of various cardinalities Do
              If $T_i \cap T_j \neq \emptyset$ Then
                 Mark $r_j$ as satisfied
           Increment n
        EndWhile
```

*Figure 3.1. High level description of MBR1 steps*

```
Input    $T_1, T_2, \ldots, T_n$: associated testing sets for $r_1, r_2, \ldots, r_n$ respectively,
         containing test cases from $t_1, t_2, \ldots, t_n$
         AR: $r_k, r_j, r_m$ where $r_k, r_j, r_m$, are potentially affected requirements
Output   R: {Tk,Tj, Tm...}subset {$T_1, T_2, \ldots, T_n$}
Declare  MAX_CARD, CUR_CARD:1...nt; LIST: list of t's; NEXT_TEST:one of $t_1, t_2, \ldots, t_n$
         MARKED: array[1...n] of boolean, initially false; MAY_REDUCE: Boolean
         Max( ):return the maximum of a set numbers; Card( ):returns the cardinality of a set
Begin
          /* Step 1: initialization*/
   For any requiremnt $r_i$ such that $r_i$ in {$r_1, r_2, \ldots, r_n$} and $r_i$ not in AR{$r_k, r_j, r_m \ldots$} Do
     MARKED[i]: =true  /* neglects Tests associated with Unaffected requirements*/
     MAX_CARD:= Max(Card(Ti)) such that MARKED[i]=false
 /* Get the max cardinality of $T_i$ in affected requirements associated tests*/
     If Card ($T_i$) =1 and (MARKED[i]: =false) Then
        R: =Union $T_i$'s   /*take union of all $T_i$'s */
     Else  MARKED [i]: =true
     For each $T_i$ such that ($T_i \cap R \neq$ empty) do MARKED [i]: =true
                                        /*Mark all $T_i$ containing elements in R*/
        CUR_CARD: =1       /*consider single element first*/
 /* Step 2 : compute R according for sets of higher cardinality*/
   Loop
       CURD_CARD: =CUR_CARD+1
       While there are $T_i$ such that (Card ($T_i$)=CUR_CARD) and not MARKED [i] Do
              /*process all unmarked sets of higher cardinality*/
         LIST:=all $t_j$ in $T_i$ (where Card($T_i$)=CUR_CARD) and (not MARKED[i])
             /*all $t_j$ in $T_i$ of size CUR_CARD */

         NEXT_TEST:= SelectTest(CUR_CARD,LIST)  /*get another $t_j$ to include in R*/
         R:= Union(R, {NEXT_TEST} ) /*add the test to R*/
         MAY_REDUCE:=false
         For each $T_i$ where NEXT_TEST belong to $T_i$ Do
             MARKED [i]:=true  /*mark $t_i$ containing next NEXT_TEST*/
             If Card (Ti)=MAX_CARD Then MAY_REDUCE:=1
         Endfor
        If MAY_REDUCE Then
            MAX_CARD: =MAX (Card ($T_i$)), for all i where MARKED[i]=false
     Endwhile
   Until CUR_CARD=MAX_CARD
End ReduceTestsuite

Function SelectTest(Size,List)    /* this function selects the next $t_i$ to be included in R*/
Declare COUNT:array[1...n]
Begin

   For each $t_i$ in LIST Do compute COUNT[$t_i$],the number of unmarked $T_j$'s of
       cardinality SIZE containig $t_i$, Construct TESTLIST consisting of tests from LIST for
            which COUNT[i] is the maximum
   If Card(TESTLIST)=1 then return(the test case in TESTLIST)
   Elseif SIZE=MAX_CARD then return (any test case in TESTLIST)
   Else return(SelectTest(SIZE+1,TestLIST))
End SelectTest
```

*Figure 3.2. Algorithm MBR1 for finding regression set R from TS*

#### 3.2.1.3 MBR1 algorithmic complexity

To analyze the worst case run-time of MBR1, let *k* denote the number of associated testing sets $T_i$ in *AR*, $S_c = \sum$ Card {$T_i$'s} denote the number of test cases $t_i$'s testing for AR, and Max-Card, the maximum cardinality of the groups of sets affected at a time. MBR1 involves two main steps: (1) computing the number of occurrences of various test cases in sets of varying cardinality, and (2) selecting the next test case to add to the *R* set. These steps are performed repeatedly until a representative set is found. Computing the number of occurrences of various test cases in sets of varying cardinality takes O (k*Max-Card) time since there are k sets of affected requirements and all elements of these sets are examined once. Selecting the next test case to be included in the *R* set requires examining the counts associated with each test case. This step takes at most O ($S_c$ * Max-Card) times. Selecting a test case and recomputing the counts is repeated at most k times. Therefore, the overall run-time of $MBRv_1$ is O (k (k+$S_c$) Max-Card).

#### 3.2.1.4 MBR1 example

To illustrate MBR1, consider the program *P* and its corresponding control-flow graph in Figure 3.3. The program reads the lengths of three sides of a triangle, classifies the triangle as either scalene, isosceles, right, or an equilateral, computes its area using a formula based on the class, and finally, outputs the class and the area computed. The code is segmented without loss of generality; $S_1,\ldots, S_{11}$ correspond to those segments. For this example, we assume that the combined conditions are handled in one predicate segment for the sake of simplicity in exposition. Table 3.1 gives the test cases used in testing the program. Table 3.2 gives the test-segment traversal information. Table establishes the test–requirement based on information given in Table 3.3.

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

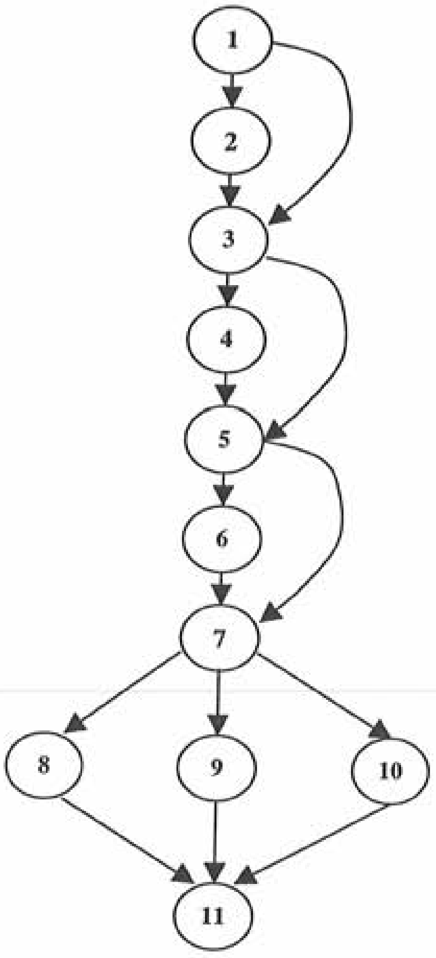

*Figure 3.3. Segmented program P and its corresponding control flow*

*Table 3.1. Tests $t_i$'s used in testing P*

| *Test Case* | *Input* | | | *Output* | |
|---|---|---|---|---|---|
| | a | b | c | Class | area |
| $t_1$ | 2 | 2 | 2 | Equilateral | 1.73 |
| $t_2$ | 4 | 4 | 3 | Isosceles | 5.56 |
| $t_3$ | 5 | 4 | 3 | Right | 6.00 |
| $t_4$ | 6 | 5 | 4 | Scalene | 9.92 |
| $t_5$ | 3 | 3 | 3 | Equilateral | 2.60 |
| $t_6$ | 4 | 3 | 3 | Isosceles | 4.47 |

*Table 3.2. Test-segment traversal information of P*

| *Test Case* | *Sequence of Segments Traversed* |
|---|---|
| $t_1$ | $S_1, S_2, S_3, S_5, S_6, S_7, S_9, S_{11}$ |
| $t_2$ | $S_1, S_2, S_3, S_5, S_7, S_{10}, S_{11}$ |
| $t_3$ | $S_1, S_3, S_4, S_5, S_7, S_8, S_{11}$ |
| $t_4$ | $S_1, S_3, S_5, S_7, S_{10}, S_{11}$ |
| $t_5$ | $S_1, S_2, S_3, S_5, S_6, S_7, S_9, S_{11}$ |
| $t_6$ | $S_1, S_2, S_3, S_5, S_7, S_{10}, S_{11}$ |

*Table 3.3. Test suite TS, testing–requirements $S_i$ , and associated tests Ti's*

| *i* | $R_i$ | $T_i$ |
|---|---|---|
| 1 | $S_1$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 2 | $S_2$ | $\{t_1,t_2,t_5,t_6\}$ |
| 3 | $S_3$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 4 | $S_4$ | $\{t_3\}$ |
| 5 | $S_5$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 6 | $S_6$ | $\{t_1,t_5\}$ |
| 7 | $S_7$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 8 | $S_8$ | $\{t_3\}$ |
| 9 | $S_9$ | $\{t_1,t_5\}$ |
| 10 | $S_{10}$ | $\{t_2,t_4,t_6\}$ |
| 11 | $S_{11}$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |

Upon modifying $S_8$, reachability analysis shows that the change in $S_8$ might potentially affect all segments except $S_9$ and $S_{10}$. The set of affected requirements, $AR = \{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_{11}\}$, is then used by MBR1 to find the set *R* of regression tests that satisfy all potentially affected requirements. Using Table 3.3, the algorithm first marks segments $S_9$ and $S_{10}$ as unaffected and neglect their associated $T_i$'s (i.e., $T_9$ and $T_{10}$ from the reduction process). MBR1 then examines all potentially affected singletons: test $t_3$ in $T_4$ and $T_8$ is chosen and included in *R*. Since test $t_3$ covers segments $S_1$, $S_3$, $S_5$, $S_7$, and $S_{11}$, the corresponding $T_i$'s of those tests are marked as being satisfied. Then, we consider unmarked $T_i$'s of cardinality two (i.e., $T_6$). Examining $T_6$ shows a tie for test cases $t_1$ and $t_5$ for the maximum. Thus, we continue processing the unmarked $T_i$'s for the next higher cardinality. $T_2$ of cardinality 4 is considered next. We only use the test cases involved in the tie to compute the maximum of cardinality 4. Again, there is a tie between $t_1$ and $t_5$ for the maximum. Since the highest cardinality is reached, a random choice can be made to include either $t_1$ or $t_5$ in the *R* set. Selecting $t_1$ to include in *R* leads to mark $T_2$ and $T_6$ as satisfied since $t_1$ exercises their corresponding requirements. Thus the resulting regression set *R* is $\{t_1, t_3\}$.

### 3.2.1.5 MBR1 testing coverage

MBR1 does not attempt to provide a complete testing coverage for all the program requirements. However, an adequate coverage of all the requirements that are impacted by the change(s) is sought. The coverage with respect to all requirements and the degree of test suite selection are dependent on the location of the modified requirement and varies with degree of reachability it exhibits with other requirements. If most of the requirements are said to be reachable from/reach the modified one, then MBR1 tends to test for all requirements. In some cases, testing for potentially affected requirements, whose tests do not reach the modification, might drive MBR1 to select such tests that could happen to cover the unaffected part. This is could be simply referred to the fact that no other tests in *TS* could satisfy that requirement. MBR2 focuses on this problem.

### 3.2.2 Modification-Based Reduction version 2 (MBR2)

MBR2 is based on the following simple observations concluded from MBR1: (i) not every test $t_k \in T_i$ exercising a requirement reachable from/reaches the modified requirement does necessarily reach the modification; (ii) if a requirement, say the modified one, is not executed under a test case, it can not affect the program output for that test; (iii) a requirement reachable from/reaches the modified segment/requirement does not necessarily affect the program output of the modification-related part.

To verify these observations through the illustrated MBR1 example: even though the modification of $S_{10}$ characterizes $S_4$ and $S_6$ as potentially affected, neither of their associated tests reach $S_{10}$. Thus, these tests fail to test for the modification even though the inclusion of tests from $T_4$ and $T_6$ is necessary according to the previous version. Further, the inclusion of tests from those

$T_i$'s might drive some requirements identified as unreachable to be executed; hence, not affecting the output of the modification-related part.

MBR2 exploits these observations by eliminating tests that do not reach the modification due to a control evaluation. The adapted observations are then used towards orienting MBR2 to only select tests executing the modified requirement. Our approach assumes that adequate tests responsible for exercising every control evaluation in the program are provided.

### 3.2.2.1 MBR2 steps

Modifying requirement $r_m$, $t_i$'s belonging to its associated test $T_m$ constitutes the set of tests that are said to execute the modified requirement. To construct *R*, MBR2 initiates the reduction process by neglecting and marking as unaffected every requirement $r_k$ not in *AR* where k in [1…*n*]. The associated test cases $T_k$'s of $r_k$'s are excluded. For the unmarked requirements, MBR2 first examines all test cases that occur in single element $T_i$'s in the test suite. A singleton test is selected and included in the *R* set only if it executes the modified requirement; i.e., it belongs to $T_m$. Requirements that are covered by the selected test are marked as satisfied. Then, all unmarked $T_i$'s of cardinality two are considered. The test case executing the modified requirement and occurring in the maximum number of $T_i$'s of the considered cardinality is chosen and added to *R*. Again, all unmarked $T_i$'s containing the selected test are marked. This process is repeated for $T_i$'s of cardinality 3, 4, …, Max-Card, where Max-Card is the maximum cardinality of the $T_i$'s. When examining the

$T_i$'s of size *n*, there may be a tie because several test cases exercising the modified requirement occur in the maximum number of $T_i$'s of that size. In this case, MBR2 examines the unmarked $T_i$'s with cardinality ($n$+1) for those test cases that were involved in the tie. The test case that occurs in the maximum number of $T_i$'s of cardinality ($n$+1) is chosen. If a decision can not be made, the $T_i$'s with greater cardinality are examined and finally a random choice is made. At any time, if the considered $T_i$ includes no test cases that execute the modified requirement, the $T_i$ is neglected and its associated requirement is marked. Notice that such neglected tests might be exercising some requirements that are reachable from/reaches the modified one(s); however, these tests do not reach the considered requirement modification and need not be run. Figures 3.4 and 3.5 outline the MBR2 steps and algorithm respectively.

```
Step 1: Determine affected requirements using reachability information
        Neglect as unaffected every requirement $r_k$ not in $AR$; k in [1…n]
        Mark $T_k$'s of $r_k$'s

Step 2: Select tests from Singletons that executes
        $R_{mod}$ and mark requirements satisfied with selection

        For unmarked $T_i$'s of cardinality =1 Do
           If ($t_j \varepsilon T_i$) executes $R_{mod}$ Then
              Select and include $t_j$ in $R$
           Mark $r_i$'s associated with singleton $T_i$'s as satisfied
           For $T_j$'s of various cardinalities Do
              If $R \cap T_j \neq \emptyset$ Then
                 Mark $r_j$ as satisfied

Step 3: Select tests from unmarked suites with higher cardinality
 That executes $R_{mod}$ and mark requirements satisfied with the selection.

        While n < Max-Card Do

           Insert in List all ($t_j \varepsilon T_i$) executing $R_{mod}$ (i.e $T_i \cap T_m \neq \emptyset$)such
                                       that $T_i$ is unmarked of cardinality n
           Get $t_j$'s such that $t_j$ occurs in the maximum of List

           If the number of returned $t_j$'s greater than 1 Then
              Tie = true
              While Tie Do
                 Get $t_j$ the maximum of tied tests in unmarked $T_i$'s of cardinality (n+1)
                 If the number of returned tests equal 1 Then
                    Tie = false
              EndWhile
           Select and include $t_j$ in $R$
           For unmarked $T_i$'s of various cardinalities Do
              If ($t_j \varepsilon T_i$) Then
                 Mark $r_j$ as satisfied

           Set List to empty
           Increment n
        EndWhile
```

*Figure 3.4. High level description of MBR2*

```
Input   $T_1, T_2, \ldots, T_n$: associated testing sets for $r_1, r_2, \ldots, r_n$ respectively,
        containing test cases from $t_1, t_2, \ldots, t_n$;
        $R_{mod}$: modified requirement. AR: $r_k, r_j, r_m$ where $r_k, r_j, r_m$, are potentially affected
        requirements
Output  R: {Tk,Tj, Tm…}subset $\{T_1, T_2, \ldots, T_n\}$
Declare MAX_CARD, CUR_CARD:1...nt; LIST: list of t's; NEXT_TEST:one of $t_1, t_2, \ldots, t_n$
        MARKED: array[1..n] of Boolean, initially false; MAY_REDUCE: Boolean
        Max( ):return the maximum of a set numbers; Card( ):returns the cardinality of a set
      Begin
      /* Step 1: initialization*/
 For any requiremnt $r_i$ such that $r_i$ in $\{r_1, r_2, \ldots, r_n\}$ and $r_i$ not in $AR\{r_k, r_j, r_m \ldots\}$ Do
     MARKED [i]: =true  /* neglects Tests associated with Unaffected requirements*/
     MAX_CARD:= Max(Card(Ti)) such that MARKED[i]=false
/* Get the max cardinality of $T_i$ in affected requirements associated tests*/
    If Card ($T_i$) =1 and ($T_i \cap T_m \neq$ empty) Then
        R:=Union $T_i$'s   /*take union of all $T_i$'s executing $R_{mod}$ */
    Else  MARKED [i]: =true
    For each $T_i$ such that ($T_i \cap R \neq$ empty) do MARKED [i]: =true
           /*Mark all $T_i$ containing elements in R*/
           CUR_CARD: =1  /*consider single element first*/
/* Step 2 : compute R according for sets of higher cardinality*/
  Loop
      CURD_CARD: =CUR_CARD+1
      While there are $T_i$ such that (Card ($T_i$)=CUR_CARD) and not MARKED [i] Do
             /*process all unmarked sets of higher cardinality*/
      LIST:=all $t_j$ in $T_i$ (where Card ($T_i$)=CUR_CARD) and (not MARKED[i]) and  $t_j$ in $T_m$
              /*all $t_j$ in $T_i$ of size CUR_CARD  executing $R_{mod}$ */
      If (Card(Ti)=CUR_CARD) and (not MARKED[i]) and ($T_i \cap T_m$ is empty) Then
          MARKED [i]: =true  /*Neglect  $T_i$'s doesn't execute $R_{mod}$*/
       NEXT_TEST:= SelectTest(CUR_CARD,LIST)  /*get another $t_j$ to include in R*/
       R:= Union(R, {NEXT_TEST} ) /*add the test to R*/
       MAY_REDUCE:=false
       For each $T_i$ where NEXT_TEST belong to $T_i$  Do
          MARKED [i]:=true  /*mark $t_i$ containing next NEXT_TEST*/
          If Card (Ti)=MAX_CARD Then MAY_REDUCE:=1
      EndFor
      If MAY_REDUCE Then
         MAX_CARD: =MAX (Card ($T_i$)), for all i where MARKED [i]=false
  Endwhile
 Until CUR_CARD=MAX_CARD
End ReduceTestsuite
```

```
Function SelectTest (Size, List)  /* this function selects the next $t_i$ to be included in R*/
Declare COUNT:array[1…n]
Begin

   For each ti in LIST Do compute COUNT[$t_i$],the number of unmarked $T_j$'s of
      cardinality SIZE containig $t_i$, Construct TESTLIST consisting of tests from LIST for
          which COUNT[i] is the maximum
   If  Card(TESTLIST)=1 then return(the test case in TESTLIST)
   Elseif  SIZE=MAX_CARD then return (any test case in TESTLIST)
   Else return(SelectTest(SIZE+1,TestLIST))
End SelectTest
```

*Figure 3.5. Algorithm MBR2 for finding regression set R from TS*

### 3.2.2.2 MBR 2 algorithmic complexity

To analyze the worst case run-time of MBR2, let $n$ denote the number of associated testing sets $T_i$, $C_m$ = Cardinality{$T_m$} denote the number of tests in $T_m$ that traverse the modified requirement $r_m$, and Max-Card, the maximum cardinality of the groups of sets. MBR2 performs two main steps: (1) computing the number of occurrences of test cases that traverse a modified requirement(s) against $T_m$ in sets of varying cardinality and (2) selecting the next test case to add to the *R* set. These steps are performed repeatedly until a representative set is found. Computing the number of occurrences of test cases in sets of varying cardinality takes $O$ ($n$*$C_m$*Max-Card) time since there are $n$ sets and all elements of these sets are examined once against $T_m$. Selecting the next test case to add to the *R* set includes examining the counts associated with each test that is said to traverse the modified requirement. This step takes at most $O$ ($C_m$*Max-Card) time. Selecting a test case and recomputing the counts is repeated at most n times. Therefore, the overall run-time of MBR2 is $O$ ($n$ ($n$+1)(Cm*Max-Card)).

### 3.2.2.3 MBR2 example

Upon modifying $S_{10}$, reachability analysis shows that the change in $S_{10}$ might potentially affect all segments except $S_8$ and $S_9$. This forms the set of potentially affected requirements, AR = {$S_1$, $S_2$, $S_3$, $S_4$, $S_5$, $S_6$, $S_7$, $S_{10}$, $S_{11}$}. Tests in $T_{10}$ = {$t_2$, $t_4$, $t_6$} are tests that execute the modified segment $S_{10}$. To find the set of regression tests *R* that execute $S_{10}$ and might satisfy the potentially affected requirements with control reaching the modification, MBR2 starts the reduction process by marking segments $S_8$ and $S_9$ as unaffected and neglect their associated $T_i$'s (i.e., $T_8$ and $T_9$). MBR2 then examines all the singletons of the potentially affected requirements: test $t_3$ of $T_4$ is not selected and included in the representative set for $t_3$ doesn't execute $S_{10}$. $T_4$ is marked as neglected, the control in $S_4$ does not reach $S_{10}$. Then, we consider unmarked $T_i$'s of cardinality two (i.e., $T_6$). Examining $T_6$ shows that neither test cases $t_1$ nor $t_5$ execute $S_{10}$. None of those tests needs to be selected. Again, $S_6$ and its associated test $T_6$ are marked as neglected. The unmarked $T_{10}$ of cardinality 3 is considered next. Test cases $t_2$, $t_4$, and $t_6$ in $T_{10}$ traverses the modified segments, $S_{10}$. Since there is a tie between test cases $t_2$, $t_4$, and $t_6$ for the maximum, we continue processing the unmarked $T_i$'s for the next higher cardinality. Thus, $T_2$ of cardinality 4 is considered next. We only use the test cases involved in the tie to compute the maximum of cardinality 4. Again, there is a tie between $t_2$ and $t_6$ for the maximum causing the $T_i$'s of cardinality 6 to be examined. Examining the unmarked $T_1$, $T_3$, $T_5$, $T_7$, and $T_{11}$ of cardinality 6 shows that $t_2$ and $t_6$ still lie in the tie for the maximum of those $T_i$'s. Since the highest cardinality is reached, a random choice can be made to include either $t_2$ or $t_6$ in the representative set. Selecting $t_2$ to include in the representative set allows $T_1$, $T_2$, $T_3$, $T_5$, $T_7$, $T_{10}$ and $T_{11}$ to be marked as satisfied since $t_2$ exercises their corresponding requirements. Thus the regression set *R* is reduced to $\{t_2\}$. Even though each of $t_4$ and $t_6$ executes the modified segments, those tests are excluded since they are redundant tests that cover the same requirements. Furthermore, the eliminated tests, $t_1$, $t_3$ and $t_5$ exercise potentially affected requirements with control evaluation leading to the execution of portions unreachable from the modified segment; hence, unaffecting the program output of the modification-related part.

# CHAPTER 4

# PRECISE REDUCTION

In this chapter, we modify the reduction technique, RED, presented in (Harrold et al., 1993) to address the regression test selection problem using slicing-based analysis of the program under test. We refer to proposed method as Precise Reduction (PR).

## 4.1 BACKGROUND

In this section, we give background information on the dynamic and relevant slicing techniques suggested in (Agrawal et al., 1993). In the subsequent sections, we describe how we modified RED to address the test selection problem using slicing-based analysis.

### 4.1.1 Dynamic slice

The dynamic slice technique is based on the observation that not every statement/segment that is executed under a test case has an effect on the program output for that test case. The dynamic program slice with respect to the program output for a given test case gives the statements/segments that were executed and had 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. However, this technique fails to identify test cases that if executed will affect the output if they are evaluated differently. The relevant slice technique extends the dynamic slicing technique to solve this problem.

### 4.1.2 Relevant slice technique

Relevant slices with respect are composed of those statements that, if modified, may alter the program output for a given test case. 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 dependency of these predicates. The set of statements corresponding to these predicates including those in the dynamic slice gives the desired relevant slice. An algorithm to compute the potential dependencies of a variable, *var*, at a location, *loc*, in a given execution history is given in Figure 4.1.

```
Static_defs=static reaching definitions of var at loc;
Dynamic_def=the dynamic reaching definition of var at loc;
Control_nodes=the closure of the static control dependeces
                    of the statements in static_defs;
intialize 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; /* i.e., 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_def as visited;
    EndIf
EndFor
return potential_deps;
```

*Figure 4.1. An algorithm to compute the potential dependencies of a variable, var, at a location, loc, in a given execution history*

## 4.2 PRECISE REDUCTION

The approach presented in (Harrold et al., 1993) for managing and reducing the size of the test suite is adapted and extended to address the regression test selection problem through introducing modification(s) information that exploits the relevant slicing analysis approach and observations presented in (Agrawal et al., 1993). The proposed algorithm, Precise Reduction (PR), is motivated by the following observations that are revised to fit the requirement-based testing nature of the reduction technique: (i) not all requirements in the program are executed under a test case; (ii) if a requirement is not executed under a test case, it can not affect the program output for that test case; and (iii) even if a requirement is executed under a test case, it does not necessarily affect the program output for that test case. The adapted observations are then used towards orienting the reduction process to only select tests with relevant slices contain modified requirement(s) and affect the output for the considered modification(s). Such proposed orientation ensures that only modification-revealing tests are the selected for revalidation; all non-modification-revealing tests are omitted and henceforth referred as precise (Rothermel and Harrold, 1996). PR also attempts to eliminate all redundant tests by removing tests that exercise the same requirement as another in the test suite. Eliminating non-modification-revealing along with redundant tests potentially reduces the *R* set of tests to be used for revalidation, an objective that is sought by PR.

PR assumes that the program under test extends the model of section 2.4 to keep potential dependencies information for the various program variables in the relevant segments/nodes. Such information enables us to compute the program dynamic and relevant slices with respect to test cases.

## 4.3 PR INPUT/OUTPUT

The PR inputs are: the set of requirements $\{r_1, r_2, ..., r_n\}$; and subsets of *TS*, $T_1$, $T_2$, …, $T_n$ such that the test cases $t_j$ in $T_i$ cover the requirement $r_i$. The approach assumes that relevant program slices with respect to the program output for all test cases in the regression test suite are computed. After a program is modified, tests $t_j$'s in $T_i$ whose relevant slices containing the modified requirement and affect the output for the considered modification are determined and forms the set $RST \subseteq TS$. If the number of $t_i$'s in *RST* exceeds 1, then PR is applied to select *R*, a reduced set of *RST* from *TS*.

## 4.4 PR ALGORITHMIC STEPS

To construct *R*, the PR algorithm first examines all test cases that occur in single element $T_i$'s in the test suite. A singleton test is selected and included in the *R* set only if it belongs to *RST*; that is, its computed relevant slice covers the modified requirement(s) and affect the output for the considered modification. Requirements that are covered by the selected test are marked as satisfied. Then, all unmarked $T_i$'s of cardinality two are considered. The test case, whose computed relevant slice covering the modified requirement and affecting the output for the considered modification(s) (belongs to *RST*), is chosen and added to the *R* set if it happens to occur in the maximum number of $T_i$. Again, all unmarked $T_i$'s containing the selected test are marked. This process is repeated for $T_i$'s of cardinality 3, 4, …, Max-Card, where Max-Card is the maximum cardinality of the $T_i$'s. When examining the $T_i$'s of size n, there may be a tie because several test cases with relevant slices containing the modified requirement and affecting the output for the considered modification happen to occur in the maximum number of $T_i$'s of that size. In this case, PR examines the unmarked $T_i$'s with cardinality $(n+1)$ for those test cases that were involved in the tie. The test case that occurs in the maximum number of $T_i$'s of cardinality $(n+1)$ is chosen. If a decision can not be made, the $T_i$'s with greater cardinality are examined and finally a random choice is made. At any time, if the considered $T_i$ includes no test cases with relevant slices containing the modified requirement and affecting the output for the considered modification, the $T_i$ is neglected and marked simply because these tests exercise $r_i$'s that do not alter the program output upon the considered requirement modification (revised observation i and ii). The reason for the inclusion of such tests in the *R* set is unnecessary and might be costly. The PR steps and algorithm are outlined in Figures 4.2 and 4.3 respectively.

```
Step 1: During off line Processing:
        Compute dynamic/relevant slices
        Determine RST, the set with tests whose relevant slices
                    contain a modified requirement ($R_{mod}$)

Step 2: Select from singletons tests whose relevant slices contain
        $R_{mod}$ and mark requirements satisfied with selection

        For unmarked $T_i$'s of cardinality =1 Do
           If ($t_j \varepsilon T_i$) and ($t_j \varepsilon$ RST) Then
              Select and include $t_j$ in $R$
          Mark $r_i$'s associated with singleton $T_i$'s as satisfied
          For $T_j$'s of various cardinalities Do
             If $R \cap T_j \neq \emptyset$ Then
                Mark $r_j$ as satisfied
           EndFor
        EndFor

Step 3: Select from unmarked suites with higher cardinality tests
        whose relevant slices contain $R_{mod}$ and mark requirements
        satisfied with the selection

        Mark as neglected all Ti's that do not contain any test with relevant slice contain
        $R_{mod}$

        While $n$ <= Max-Card Do

          Insert in List all ($t_j \varepsilon T_i \cap$RST) such
                      that $T_i$ is unmarked of cardinality $n$
          Get $t_j$'s such that $t_j$ occurs in the maximum of List

         If the number of returned $t_j$'s greater than 1 Then
            Set Tie to true
            While Tie Do
               Get $t_j$ the maximum of tied tests in unmarked $T_i$'s of cardinality ($n$+1)
               If the number of returned tests equal 1 Then
                  Set Tie to false
            EndWhile
         Select and include $t_j$ in $R$
         For unmarked $T_i$'s of various cardinalities Do
            If ($t_j \varepsilon T_i$) Then
               Mark $r_j$ as satisfied
           ElseIf ($n$ = Max-Card) Then
               Mark $r_j$ as neglected
         EndFor

         Set List to empty
         Increment $n$
        EndWhile
```

*Figure 4.2. High level Description of PR steps*

```
Input    $T_1, T_2, \ldots, T_n$: associated testing sets for $r_1, r_2, \ldots, r_n$ respectively,
         containing test cases from $t_1, t_2, \ldots, t_n$;
         $R_{mod}$: modified requirements.
         AR: $t_k, t_j, \ldots, t_m$ where $t_k, t_j, t_m$, are tests whose relevant slices contain $R_{mod}$
Output   R: {Tk,Tj,Tm…}subset {$T_1, T_2, \ldots, T_n$}
Declare  MAX_CARD, CUR_CARD:1...nt; LIST: list of t's; NEXT_TEST:one of $t_1, t_2, \ldots, t_n$
         MARKED: array[1..n] of Boolean, initially false; MAY_REDUCE: Boolean
         Max( ):return the maximum of a set numbers; Card( ):returns the cardinality of a set
    Begin
    /* Step 1: initialization*/
  MAX_CARD:= Max(Card(Ti)) /* Get the max cardinality of $T_i$ in affected
                                        requirements associated tests*/
  If Card ($T_i$)=1 and ($T_i \cap RST \neq \emptyset$) Then
      R: =Union $T_i$'s   /*take union of all $T_i$'s with relevant slices contain $R_{mod}$ */
  Else  MARKED [i]: =true   /*neglect single element $T_i$'s with relevant slices not
                                        containing $R_{mod}$ */
  For each $T_i$ such that ($T_i \cap R \neq$ empty) do MARKED[i]: =true
    /*Mark all $T_i$ containing elements in R*/
        CUR_CARD: =1      /*consider single element first*/
  /* Step 2 : compute R according for sets of higher cardinality*/
  Loop
    CURD_CARD: = CUR_CARD+1
    While there are $T_i$ such that (Card ($T_i$)=CUR_CARD) and not MARKED[i] Do
          /*process all unmarked sets of higher cardinality*/
    LIST:=all $t_j$ in $T_i$ (where Card($T_i$)=CUR_CARD) and (not MARKED[i]) and $t_j$ inRST
        /*all $t_j$ in $T_i$ of size CUR_CARD whose relevant slice contain $R_{mod}$ */
    If (Card(Ti)=CUR_CARD) and (not MARKED[i]) and ($T_i \cap RST= \emptyset$) Then
        MARKED [i]: =true  /*Neglect $T_i$'s with relevant slice not containing $R_{mod}$*/
     NEXT_TEST:= SelectTest(CUR_CARD,LIST)  /*get another $t_j$ to include in R*/
     R:= Union(R, {NEXT_TEST} ) /*add the test to R*/
     MAY_REDUCE:=false
     For each $T_i$ where NEXT_TEST belong to $T_i$ Do
        MARKED [i]:=true  /*mark $t_i$ containing next NEXT_TEST*/
        If Card (Ti)=MAX_CARD Then MAY_REDUCE:=1
     Endfor
     If MAY_REDUCE Then
        MAX_CARD: =MAX (Card ($T_i$)), for all i where MARKED[i]=false
   Endwhile
 Until CUR_CARD=MAX_CARD
End ReduceTestsuite
```

```
Function SelectTest(Size,List)   /* this function selects the next $t_i$ to be included in R*/
Declare COUNT:array[1...n]
Begin
   For each ti in LIST Do compute COUNT[$t_i$],the number of unmarked $T_j$'s of
      cardinality SIZE containig $t_i$, Construct TESTLIST consisting of tests from LIST for
         which COUNT[i] is the maximum
   If Card(TESTLIST)=1 then return(the test case in TESTLIST)
   Elseif SIZE=MAX_CARD then return (any test case in TESTLIST)
   Else return (SelectTest(SIZE+1,TestLIST))
End SelectTest
```

*Figure 4.3. Algorithm PR for finding regression set R from TS*

## 4.5 PR ALGORITHMIC COMPLEXITY

To analyze the worst case run-time of PR, let *n* denote the number of associated testing sets $T_i$, $C_s$ = Cardinality {RST} denote the number of tests in RST, and Max-Card, the maximum cardinality of the groups of sets. PR involves two main steps: (1) computing the number of occurrences of test cases with relevant slices containing a modified requirement(s) in sets of varying cardinality and (2) selecting the next test case to add to the representative set. These steps are performed repeatedly until the *R* set is found. Computing the number of occurrences of test cases in sets of varying cardinality takes O ($n$*$C_s$*Max-Card) time since there are *n* sets and all elements of these sets are examined once against RST. Selecting the next test case to add to the representative set includes examining the counts associated with each test with relevant slices containing a modified requirement(s). This step takes at most *O* ($C_s$*Max-Card) time. Selecting a test case and recomputing the counts is repeated at most n times. Therefore, the overall run-time of PR is *O* ($n$ ($n$+1)($C_s$*Max-Card)).

## 4.6 PR EXAMPLE

We use program *P* and its associated tests explained in section 3.2.1.4 to illustrate our proposed PR technique. Table 4.1 gives the test cases used in testing the program *P*. Table 4.2 gives *TS* that includes test–requirement information. Figures 4.4 and 4.5 show the relevant slices with respect to the program output for $t_2$, $t_4$, and $t_6$.

*Table 4.1. Tests $t_i$'s used in testing P*

| *Test Case* | *Input* | | | *Output* | |
|---|---|---|---|---|---|
| | a | b | c | Class | area |
| $t_1$ | 2 | 2 | 2 | Equilateral | 1.73 |
| $t_2$ | 4 | 4 | 3 | Isosceles | 5.56 |
| $t_3$ | 5 | 4 | 3 | Right | 6.00 |
| $t_4$ | 6 | 5 | 4 | Scalene | 9.92 |
| $t_5$ | 3 | 3 | 3 | Equilateral | 2.60 |
| $t_6$ | 4 | 3 | 3 | Isosceles | 4.47 |

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

*Figure 4.4. The relevant program slice for $t_4$*

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

*Figure 4.5. The relevant program slice for $t_2$ and $t_6$*

*Table 4.2. Test suite TS, testing–requirements $S_i$ , and associated tests Ti's*

| $i$ | $R_i$ | $T_i$ |
|---|---|---|
| 1 | $S_1$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 2 | $S_2$ | $\{t_1,t_2,t_5,t_6\}$ |
| 3 | $S_3$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 4 | $S_4$ | $\{t_3\}$ |
| 5 | $S_5$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 6 | $S_6$ | $\{t_1,t_5\}$ |
| 7 | $S_7$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |
| 8 | $S_8$ | $\{t_3\}$ |
| 9 | $S_9$ | $\{t_1,t_5\}$ |
| 10 | $S_{10}$ | $\{t_2,t_4,t_6\}$ |
| 11 | $S_{11}$ | $\{t_1,t_2,t_3,t_4,t_5,t_6\}$ |

```
read(a,b,c);
class:=scalene;
if a=b or b=c                                      S1

   class:=isosceles;                               S2

if a*a=b*b+c*c                                     S3

   class:=right;                                   S4

if a=b and b=c                                     S5

   class :=equilateral;                            S6

case class of                                      S7

   right        :area:=b*c/2;                      S8

   equilateral: area:=a*2*sqrt(3/4);               S9

   otherwise : s:=(a+b+c)/2;
               area:=sqrt(s*(s-a)*(s-b)*(s-c));
                                                   S10
end;
write(class, area);                                S11
```

*Figure 4.6. The relevant program slice for* $t_3$

Upon modifying $S_{10}$, dynamic/relevant-slicing analysis shows that the change in $S_{10}$ may affect the program output only for $t_2$, $t_4$, and $t_6$ for this example. Note that the modified segment $S_{10}$ belongs to the relevant slice with respect to the program output of these tests. However, the relevant program slices with respect to other test cases do not include the modified $S_{10}$. For example, Figure 4.6 shows that the modified $S_{10}$ does not belong to the relevant slice with respect to the program output for $t_3$. Tests $t_2$, $t_4$, and $t_6$ then constitute the *RST* to be used in the reduction test selection process.

Using Table 4.2, the PR algorithm first examines the singleton $T_i$'s: tests in $T_4$ and $T_8$ can not be included in the *R* set since $t_3$ does not belong to *RST*; i.e., the computed relevant slice for $t_3$ does not contain $S_{10}$. $T_4$ and $T_8$ are marked as being neglected. Then, we consider unmarked $T_i$'s of cardinality two (i.e., $T_6$ and $T_9$). Test cases $t_1$ and $t_5$ do not belong to *RST*. $T_6$ and $T_9$ are also marked as neglected. The unmarked $T_{10}$ of cardinality 3 is considered next. Test cases $t_2$, $t_4$, and $t_6$ in $T_{10}$ belong to *RST*; i.e., they affect or potentially affect the output for the considered change. Since there is a tie between test cases $t_2$, $t_4$, and $t_6$ for the maximum, we continue processing the unmarked $T_i$'s for the next higher cardinality. Thus, $T_2$ of cardinality 4 is considered next. We only use the test cases involved in the tie to compute the maximum of cardinality 4. Again, there is a tie between $t_2$ and $t_6$ for the maximum causing the $T_i$'s of cardinality 6 to be examined. Examining the unmarked $T_1$, $T_3$, $T_5$, $T_7$, and $T_{11}$ of cardinality 6 shows that $t_2$ and $t_6$ still lie in the tie for the maximum of those $T_i$'s. Since the highest cardinality is reached, a random choice can be made to include either $t_2$ or $t_6$ in the representative set. Selecting $t_2$ to include in the representative set allows $T_1$, $T_2$, $T_3$, $T_5$, $T_7$, $T_{10}$ and $T_{11}$ to be marked as satisfied since $t_2$ exercises their corresponding requirements. Thus the resulting regression test *R* is $\{t_2\}$. Note that the modification-revealing tests $t_4$ and $t_6$ are excluded since they are redundant tests that cover same requirements and have the same relevant slices as the selected $t_2$. The eliminated tests, $t_1$, $t_3$ and $t_5$, are non-modification-revealing tests; they don't affect the output for the considered change.

# CHAPTER 5

# McCABE-BASED REGRESSION TEST SELECTION METRICS

In this chapter, we address a second major problem in selective regression testing: the coverage identification problem, the problem of determining where additional tests may be required. We approach this problem by proposing two selective McCabe cyclomatic-based test selection metrics. These are the Reachability regression Test selection McCabe-based metric (RTM), and the dataflow Slices regression Test McCabe-based metric (STM). These metrics monitor testing coverage adequacy, suggest bounds on the number of tests to rerun, help in revealing any shortage or redundancy in the test suite, and assist in identifying where additional tests may be required. In the following subsequent sections, we explain our objectives and approaches in finding these metrics.

## 5.1 McCABE CYCLOMATIC COMPLEXITY

McCabe Cyclometic complexity is a software metric that provides a quantitative measure of the logical complexity of the program (McCabe, 1976). When McCabe metric is used in the context of path testing, the value computed for the cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of independent paths that compromise the basis set, and by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program segments at least once. To know how many paths to look for, the computation of the cyclomatic complexity gives us the answer. The cyclomatic number V(G) of a flow graph G with $n$ nodes/segments, $e$ edges, and P predicate nodes/segments calculates as follows:

V (G)= $e$-$n$+2 where 2 is a constant; or by using the count of the predicates in G given by V(G)=P+1.

## 5.2 OBJECTIVES OF McCABE-BASED REGRESSION TEST SELECTION METRICS

The proposed McCabe-based regression testing metrics are coverage regression test selection metrics that visualize the affected component of the graph to quantify the complexity and the retesting effort, assist in static generations of tests to retest for the change, and find upper and lower bounds of selective regression tests that guarantee an adequate coverage of the modified program components. Such metrics can be used as a guide to provide a deterministic measure of the number of selected tests for revalidation. It also attempts to provide a logical measure of the modification complexity, the complexity of the part affected by the modification.

More specifically, the two McCabe-based regression independent test selection metrics are: Reachability regression Test selection McCabe-based metric (RTM), and dataflow Slices regression Test McCabe-based metric (STM). RTM is an upper bound metric that derives its measures from reachability information and provides an upper indication of paths that must be tested for being potentially affected by the modification, and by implication, an upper bound of tests to rerun for exercising these paths at least once. STM, a data flow McCabe-based variable dependent metric, derives its measures from information related to slices affected by the modification and computes two bounds: an upper and a lower bound.

## 5.3 REACHABILITY REGRESSION TEST SELECTION McCABE-BASED METRIC(RTM)

The Reachability regression Test selection McCabe-based metric, denoted by RTM, attempts to provide an upper bound indication of the number of selected regression tests that guarantee the coverage of requirements potentially affected by the modification at least once. The metric is independent of the nature of modification.

### 5.3.1 RTM computation

To calculate the upper bound, our approach uses segment reachability information explained in section 2.4 of chapter 2. Given a control flow graph G with *n* segments, and that segment *l* has been modified, the reachability computation gives the segments that are reachable/reach the modification and classified as potentially affected. We denote the upper bound upon modifying segment *l* in graph G as $RU_G(l)$. To compute $RU_G(l)$, our approach first removes all control constructs that are not reachable from/reach the modified segment/node *l* to obtain a reduced flow graph $G_1$. $G_1$ must contain a unique entry and exit nodes; we consider that the entry and exit points are those of G.

Computing the cyclomatic complexity of $G_1$, $V(G_1)$, gives us $RU_G(l)$. $RU_G(l)=V(G_1)$ provides an upper bound of the linearly independent paths that must be retested and, by implication, an upper bound on the number of tests that must be rerun to guarantee coverage of all segments potentially affected by the change in *l* at least once. Note that the reachability information will include all segments that might at most be affected by the change in *l*; henceforth, the calculated complexity of $G_1$, $V(G_1)$, is considered an upper bound.

### 5.3.2 RTM example

To illustrate the computation of our proposed RTM upper bound, Figure 5.1 shows that upon modifying node 3, reachability information identifies nodes 1, 3, 7, 8, 9, and 10 as potentially affected. From node 1, the entry node, to node 10, the exit node, we remove all control constructs not reaching the potentially affected nodes to obtain the reduced graph $G_1$. The testing effort is reduced to $RU_G(3)=V(G_1)=2$. This a suggested upper bound on tests to rerun for revalidating all potentially affected nodes.

Upon modifying node 4, reachability information identifies nodes 1, 2, 4, 6 and 10 as potentially affected. From node 1, the entry node, to node 10, the exit node, we remove all control constructs not reaching the potentially affected nodes to obtain the reduced graph $G_2$. The testing effort is reduced to $RU_G(4)=V(G_2)=1$. This is a suggested upper bound on tests to rerun for revalidating all potentially affected nodes.

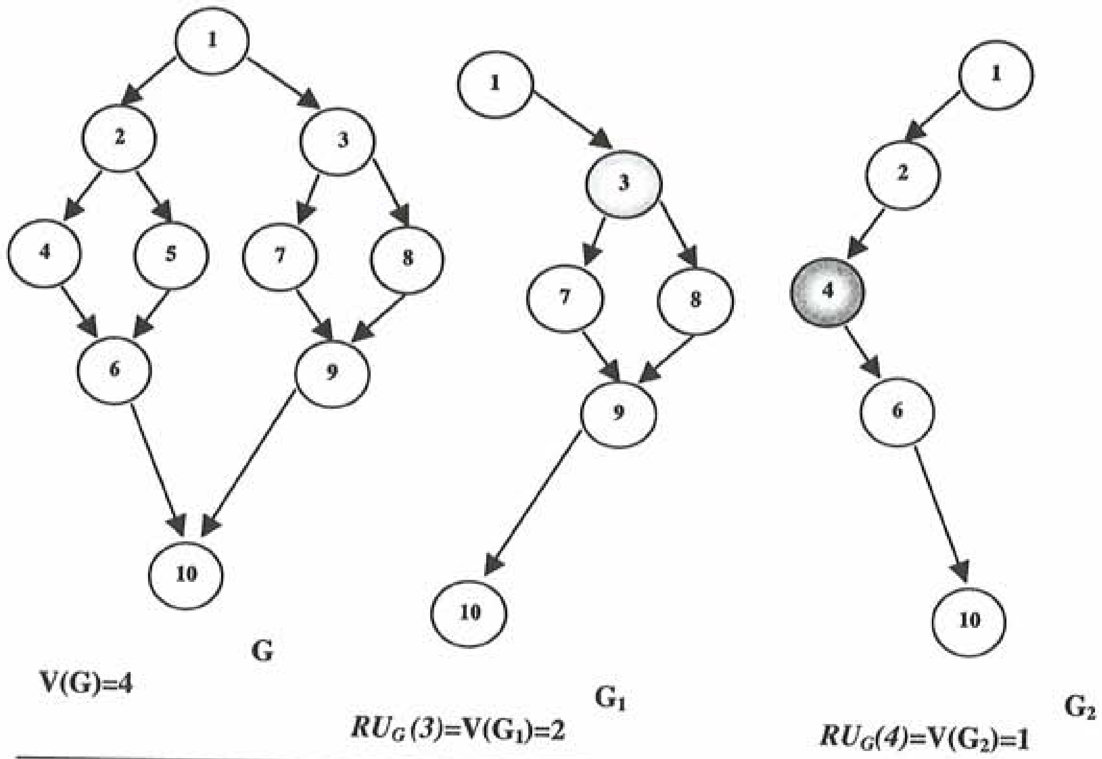

*Figure 5.1. RTM computations upon modifying location 3 and 4*

## 5.3.3 RTM and the program structure

The upper bound is affected by the structure of the program under regression testing and varies with the location of the modified segment and the degree of reachability it exhibits with other ones. If most of the segments are said to be reachable from/reach the modified one, then the upper bound tends to approach the program cyclomatic complexity; hence, slight simplification of the regression test selection problem is attained.

## 5.4 DATAFLOW SLICES REGRESSION TEST McCABE-BASED METRIC (STM)

In this section, we extend McCabe complexity to deal with variable changes. The Dataflow Slices regression Test McCabe-based metric, denoted by STM, computes two bounds. The STM bounds indicate an upper and lower number of regression tests to rerun to exercise the affected definition-use pairs due to the considered variable(s) change.

### 5.4.1 STM assumptions

To determine STM bounds, our approach extends McCabe complexity to deal with variable changes. Our approach assumes that definitions and uses of variables are attached to nodes in the flow graph. Upon modifying a program, data flow analysis is performed to determine the affected def-use pairs. 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 conditional statement. To identify the definitions and uses that are affected by the program edit, we use the Backward/Forward algorithms suggested in (Gupta et al., 1992).

### 5.4.2 STM computations

To compute the data-flow test bounds, we assume that a control flow graph G with $n$ segments is given, and that the sets of affected definitions-use pairs have been identified due to the modification of variable $x$ at location $l$. From

G, we first remove all paths with control constructs not leading to the identified affected def-use pairs to obtain a reduced graph $G_1$. To be valid, $G_1$ must have a unique entry segment/node and terminate with a unique exit node/segment. We consider that the unique entry and exit points are those of the program or the module under regression test.

### 5.4.3 STM upper bound

To compute the STM upper bound upon the modification of variable $x$ at location $l$, denoted by $SU_G(x/l)$, we compute $SU_G(x/l)=V(G_1)$. $V(G_1)$, the cyclomatic complexity of $G_1$, gives the number of all possible independent paths from the unique entry to the exit point of the program/module and, by implication, the number of tests that must be rerun to ensure the coverage of the affected def-use pairs through the independent paths of G upon the modification of variable $x$ at location $l$. This bound indicates all independent possible paths from the entry to the exit point that traverse the affected def-use pairs; hence, it is referred to as upper bound.

#### 5.4.3.1 STM upper bound example

To illustrate the computation of our proposed STM upper bound, Figure 5.2 shows that upon modifying node 1, forward walk on variable $x$, identifies a use of $x$ in nodes 7. From node 1, the entry node, to node 10, the exit node, we remove all control constructs not reaching the affected def-use pair (1,7) to obtain a reduced graph $G_1$. The testing effort is reduced to $SU_G(x/1)=V(G_1)=1$.

Upon modifying node 10, backward walk on variable y, identifies a use of y in nodes 2. From node 1, the entry node, to node 10, the exit node, we remove all control constructs not reaching the affected def-use pair (2,10) to obtain a reduced graph $G_2$. The testing effort is reduced to $SU_G(y/10)$=V($G_2$)=2.

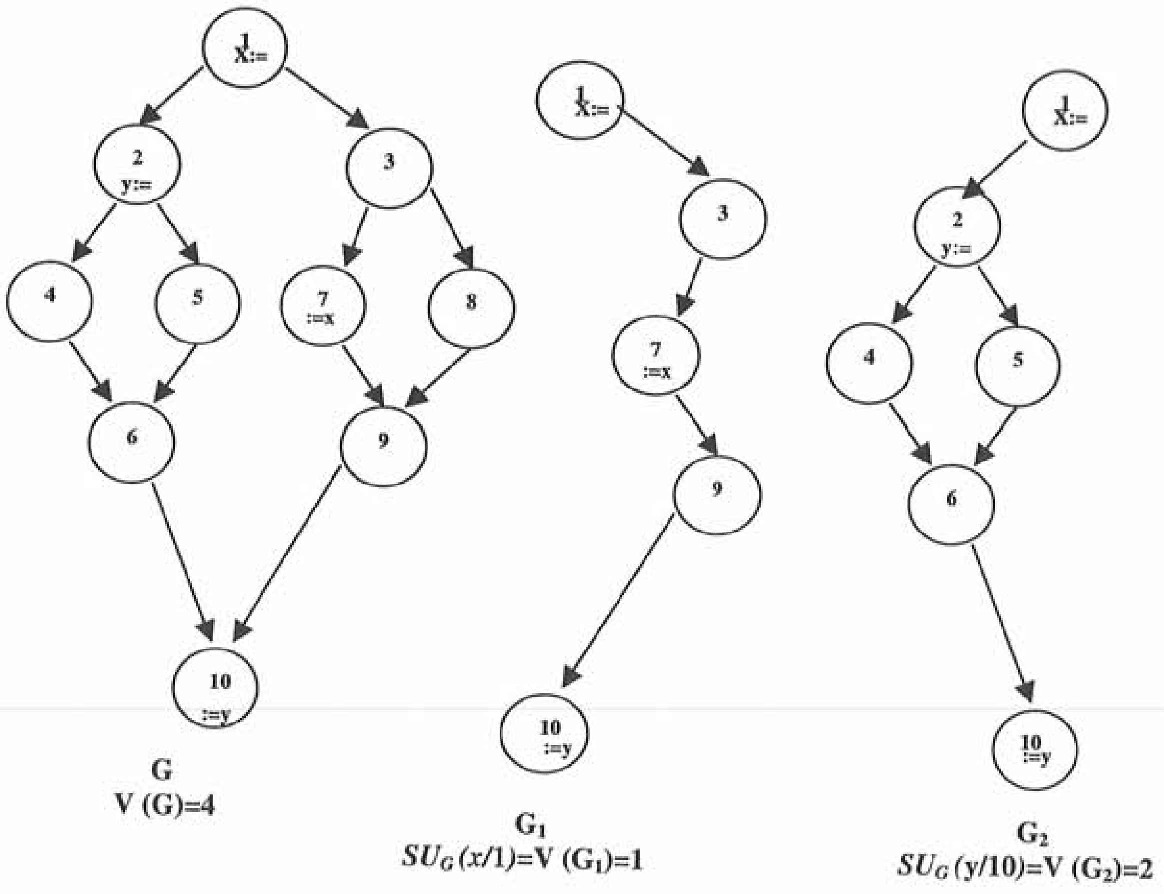

*Figure 5.2. STM computations upon modifying variable x at location 1, and variable y at 10*

### 5.4.4 STM lower bound-with adjusted entry and exit points

The bound described in the previous section provides an upper bound of all independent possible paths that traverse the affected def-use pairs from the entry to the exit point of the program or the module under test. To find the lower bound, we aim to find the least number of linearly independent paths that traverse the affected def-use pairs. Clearly, adjusting the entry and exit points, without the loss of affected def-use testing coverage, might provide a solution.

#### 5.4.4.1 STM lower bound-Forward Adjustment

When a definition variable $x$ is modified at location $l$, we set the node containing the modified variable as an adjusted virtual entry point. ForwardWalk (Gupta et al., 1992) is then performed to locate the uses of the variables $x$ modified at $l$ along each paths. To find an adjusted virtual exit point, we examine the number of returned affected def-use pairs of the modified variable $x$ edited at $l$. If a single affected def-use pair is returned by the walk, then we can set the node containing the only use as an adjusted virtual exit point. However, in case of the obtained def-use pairs exceeds one, we must find a node through which the obtained uses converge. To determine such a node, all the obtained uses are marked. Then, depth search is performed from the marked uses until the first node with all its direct or indirect predecessors are the marked uses is reached. This point is marked as the adujsted virtual exit point. In Figure 5.3, we outline the algorithm Forward

Adjustment for finding the adjusted entry and exit points upon the modification of a definition variable *x* at location *l*.

```
When a definition variable x is modified at location l,
   Set the node at l as an adjusted virtual entry point
   Perform Forward Walk  /* to loacte affected  def-uses */
    Mark the uses
   If a single affected def-use pair is returned Then
      Set the mark use  as a virtual exit point
   Else
      Perform depth search from the marked uses until a node EX with direct
           or indirect predecessors are the marked uses is reached
      Set EX as the adjusted virtual exit point

Return(adjusted virtual entry point, adjusted virtual exit point)
```

*Figure 5.3. Algorithm Forward Adjustment for finding adjusted virtual entry and exit points*

### 5.4.4.2 STM lower bound-Backward Adjustment

When a use of variable *x* is modified at location *l*, we set the node containing the modified variable as an adjusted virtual exit point. BackwardWalk (Gupta et al., 1992) is then performed to locate the definitions of the variables *x* modified at *l* along each paths. To find an adjusted virtual entry point, we examine the number of returned affected def-use pairs of the modified variable *x* edited at *l*. If a single affected def-use pair is returned by the walk, then we can set the node containing the only definition as an adjusted virtual entry point. However, in case of the obtained def-use pairs exceeds one, we must find a node through which the obtained definitions converge. To determine such a node, all the obtained nodes containing the definitions are marked. Then, reverse-depth search is performed from the marked definitions until the first node with all its direct or indirect successors are the marked definitions is reached. This point is marked as the adujsted virtual entry point. In Figure 5.4, we outline the algorithm Backward Adjustment for finding the adjusted entry and exit points upon the modification of a use variable $x$ at location $l$.

```
When a use variable $x$ is modified at location $l$,
   Set the node at $l$ as an adjusted virtual exit point
   Perform Back Walk  /* to loacte affected  def-uses */
    Mark the definitions
   **If** a single affected def-use pair is returned **Then**
      Set the marked definition as a virtual entry point
   **Else**
      Perform reverse-depth search from the marked  definitions until a node
         *EN* with direct or indirect successors are the marked definitions is
         reached
     Set *EN* as the adjusted virtual entry point
Return(adjusted virtual entry point, adjusted virtual exit point)
```

*Figure 5.4. Algorithm Backward Adjustment for finding adjusted virtual entry and exit points*

### 5.4.4.3 STM lower bound computation

By setting the adjusted virtual entry/exit point, we obtain a sub-graph $G_1$ of G. Each node in $G_1$ can be reached by the adjusted virtual entry node and each node can reach the adjusted virtual exit node. We refer to the paths of $G_1$ as the basis subpaths in $G_1$ relative to G obtained upon modifing a variable $x$ at location $l$.

To compute the STM lower bound upon the modification of variable $x$ at location $l$, denoted by $SL_G(x/l)$, we compute $SL_G(x/l)=V(G_1)$. $V(G_1)$, the cyclomatic complexity of $G_1$, gives the number of all possible subpaths that must be tested to ensure the coverage of those affected def-use pairs from the virtual adjusted entry to the virtual adjusted exit point, and by implication, the upper and lower numbers of tests to rerun with respect to $G_1$ and G respectively. Note, that $SL_G(x/l)$ approaches the $SU_G(x/l)$ incases where the adjusted virtual entry/exit nodes are those of the program or module under test. Otherwise, $SU_G(x/l) < SL_G(x/l)$ for $V(G_1) < V(G)$; hence, a lower bound is obtained.

### 5.4.4.4 STM lower bound example

To illustrate the computation of our proposed lower bounds, Figure 5.5 shows that upon modifying node 3, forward walk on variable $x$, identifies two uses of $x$ in nodes 4 and 5. The Forward Adjustment algorithm set nodes 3 and 6 as virtual entry and exit points respectively to obtain a reduced graph $G_2$ of $G_1$. The testing effort is reduced to $SL_{G1}(x/3)=V(G_2)=2$. This is the minimal number of tests needed to rerun to test the affected sub-paths relative to $G_1$.

Upon modifying $x$ at location 6 of graph G in Figure 5.6, backward walk on variable $x$, identifies two definitions of $x$ in nodes 4 and 5. Backward Adjustment set nodes 3 and 6 as virtual entry and exit points respectively to obtain a reduced graph $G_2$. The testing effort is reduced to $SL_G(x/6)=V(G_2)=2$. This is the minimal number of tests needed to rerun to test the affected sub-paths. Editing $x$ at location 4 of the same graph, we obtain a reduced graph $G_3$ using Forward Adjustment with $SL_G\ (x/4)$=V $(G_3)$=1.

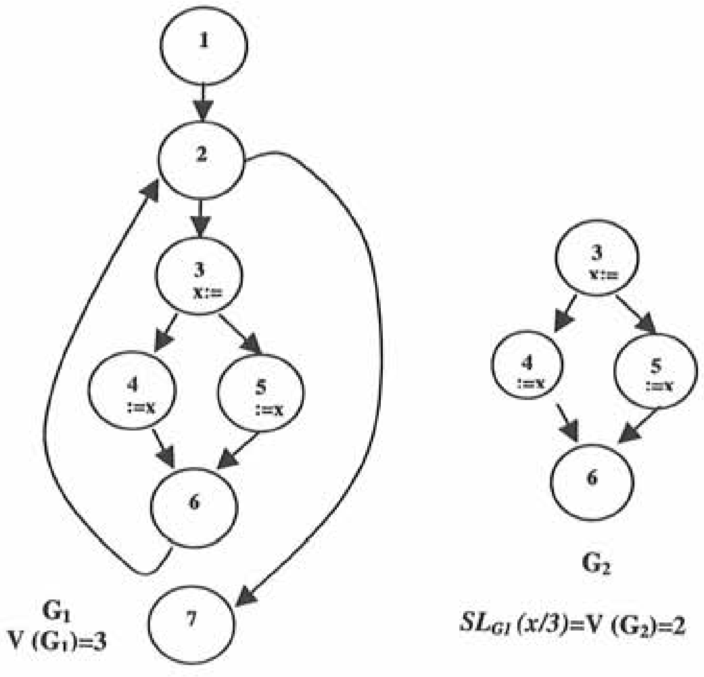

*Figure 5.5. $G_2$ a reduced control flow graph of $G_1$ upon applying Forward Adjustment on variable x at location 3*

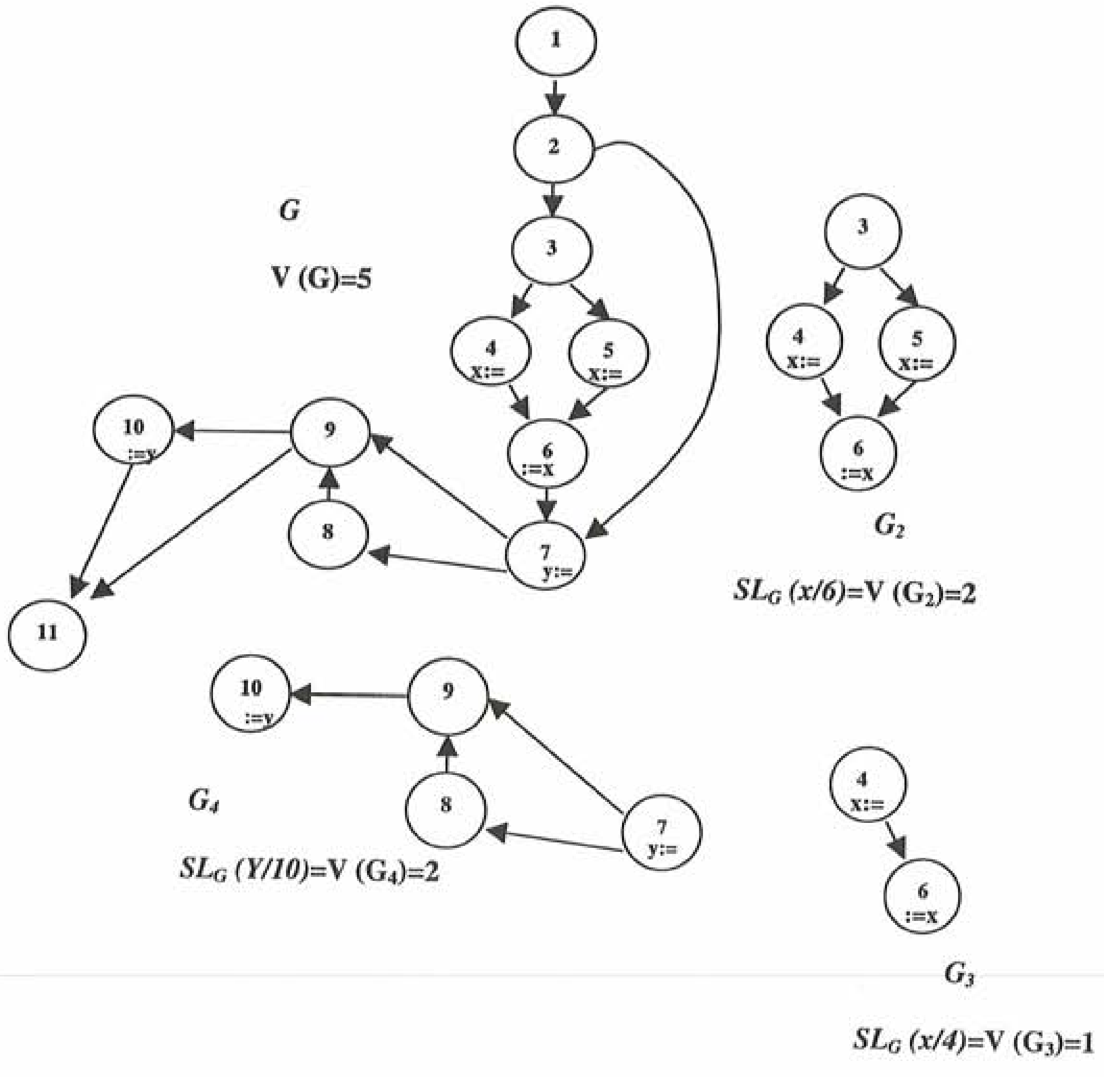

*Figure 5.6. $G_2$,, $G_3$, and $G_4$ reduced control flow graphs of G upon applying Forward/Backward Adjustment*

#### 5.4.4.5 STM Lower bound interpretation

The value of $SL_G(x/l)$ suggests the minimum number of tests/paths to look for. To choose these tests/paths relative to G, the graph representing the program/module under regression test, the tester may take any test exercising a path from the unique entry of G, passing through the adjusted virtual entry and exit of $G_2$ along any of the $G_2$ basic subpaths, to the unique exit of G. The lower bound metric of STM recommends choosing $SL_G(x/l)$ of these tests that are sufficient to test for the basic sub-paths of $G_2$ relative to G. This is the minimum number of tests/paths to look for relative to G.

## 5.5 REGRESSION TESTING WITH McCABE-BASED COMPLEXITY BOUNDS

A selective regression testing algorithm based on the suggested bounds would provide: (i) a deterministic measure of the modification complexity introduced; (ii) bounds on the number of retests for revalidation; (iii) testing coverage adequacy with respect to the affected portion of the program; (iv) and degree of testing reliability and safety. These are necessary aspects that most regression test selection techniques fail to address. Tables 5.1 and 5.2 summarize how the RTM and STM bounds could be used to monitor the test selection process and identify where additional tests may be required.

In this section, we also outline an example of using STM bounds to address the test selection problem in data flow testing. Our example could be extended to address other coverage selective regression testing techniques.

*Table 5.1 Number of retests(#R) relative to RTM bounds*

| *Metric* | *Interpretation #R relative to RU(l)* | | *Action* | *Reliability/Safety of Retesting* |
|---|---|---|---|---|
| | $\#R < RU_G(l)$ | Inadequacy of testing with respect to all potentially affected requirements<br><br>Coverage of the affected requirement is not certainly attained | Update test suite to attain coverage adequacy with respect to the potentially affected portion and retest | Unreliable/Unsafe<br><br>— |
| *RTM* | $\#R = RU_G(l)$ | Coverage of all potentially affected requirements is attained through all possible independent potentially affected paths | | More Reliable |
| | $\#R > RU_G(l)$ | Coverage of all potentially affected requirements is attained at least once<br><br>Redundancy of retest | Optional: manage the test suite to eliminate redundancy<br><br>Test for $RU_G(l)$ | + |

*Table 5.2 Number of retests(#R) relative to STM bounds*

| *Metric* | *Interpretation #R and STM* | | *Action* | *Reliability/Safety* |
|---|---|---|---|---|
| *STM* | $\#R < SL_G(x/l)$ | Inadequacy of testing<br><br>Coverage of affected slices is not attained | Update test suite to cover affected slices and retest | Unreliable<br>– |
| | $\#R = SL_G(x/l)$ | Coverage of the affected portion is attained with minimal retests | | Less Reliable |
| | $\#R = SU_G(x/l)$ | Coverage is attained through all possible independent affected paths | | More reliable |
| | $\#R > SU_G(x/l)$ | Coverage attained at least once<br><br>Redundancy of retest | manage the test suite to eliminate redundancy<br><br>Test for $\#R = SU_G(x/l)$ | + |

### 5.5.1 Dataflow selective regression testing using STM bounds

Data flow testing based on backward/forward algorithms finds all the affected def-use pairs due to a modification in the program. The technique then requires selecting all test cases in the regression testing suite that satisfy these affected pairs. Such test selection tends to include all redundant tests in the regression suite satisfying the affected def-use pairs and leads high number of selected retests in some cases. This fact is experimentally observed. To select a minimal regression suite with redundant tests eliminated, we recommend using STM bounds as indictors in the test selection process. This is an outline of the recommended strategy:

1) Determine affected def-use pairs due to the modification (e.g., using Backward/Forward algorithms).

2) Compute STM lower and upper bounds. STM lower bound provides a proper indication of the number of the affected sub-paths. STM upper bound indicates the maximum of tests to rerun to revalidate the affected independent paths.

3) Using STM bounds as indicators of test coverage adequacy, select test case from the test suite that will force the execution of each of the sub-paths.

## 5.6 SUMMARY

In this section, we summarize the major properties and characteristics of the McCabe-based metrics suggested in previous sections. Tables 5.3 and 5.4 summarize the objectives, characteristics, requirements, and suitability of applications of the suggested RTM and STM metrics respectively.

*Table 5.3. RTM metric table*

| *Metric* | | *Description and objectives* | *Characteristics and applications* |
|---|---|---|---|
| *RTM* | *Upper Bound* | Denoted by $RU_G(l)$, RTM upper bound upon modifying location $l$ in control graph $G$.<br><br>Indicates an upper bound of retests with respect to all potentially affected requirements<br><br>Monitors testing coverage adequacy with respect to all potentially affected independent paths<br><br>Quantifies the complexity of the modification and the maximum testing effort for revalidating potentially affected requirements<br><br>Helps in revealing redundancy of tests (if any) in the initial test suite | Requires control graph modeling<br><br>Requires reachability information<br><br>McCabe based computation<br><br>Location dependent<br><br>Varies with the modification and degree of reachability the modification exhibits with other requirements<br><br>Could be integrated with selective regression testing techniques to guide the test selection process towards safety |

*Table 5.4. STM summary table*

| *Metric* | | *Description and objectives* | *Characteristics and applications* |
|---|---|---|---|
| *STM* | *Lower Bound* | Denoted by $SL_G(x/l)$, STM lower bound upon editing $x$ at location $l$ in control graph $G$. | Requires control flow modeling and data flow information |
| | | Provides a lower bound on numbers of tests to rerun for revalidating affected definition-use pairs | McCabe based computation |
| | | | Variable and location dependent |
| | | Monitors minimal testing coverage adequacy with respect to all potentially affected independent paths | Suitable for selective regression techniques thatuses/incorporates data flow slicing-based analysis |
| | | Quantifies minimal testing effort | For structured programs, upper and lower bounds varies with the modification |
| | | Helps in revealing shortage in tests (if any) in the initial test suite | For unstructured programs, upper bound approaches cyclomatic complexity |
| | *Upper Bound* | Denoted by $SU_G(x/l)$, STM upper bound upon editing $x$ at location $l$ in control graph $G$. | Could be integrated with Dataflow-based selective regression testing techniques to guide the test selection process. |
| | | Indicates an upper bound of retests with respect to all affected definition-use slices | |
| | | Monitors testing coverage adequacy with respect to all potentially affected independent paths | |
| | | Quantifies the complexity of the modification and the maximum testing effort for revalidation | |
| | | Helps in revealing redundancy of tests (if any) in the initial test suite | |

# CHAPTER 6

# EMPIRICAL RESULTS AND DISCUSSION

In this chapter, we use 60 test problems to empirically compare the proposed reduction-based selective regression testing methods described in chapters 3 and 4 to other three minimization-oriented selective techniques. These are SA (Mansour and El-Fakih, 1999), RED (Harrold et al., 1993), and SLI (Agrawal et al., 1993). We base our comparison on four quantitative criteria. These criteria are the test cases selection percentage, algorithm's execution time, precision, and inclusiveness.

Furthermore, we use the test problems to try our suggested McCabe-based metric on two coverage-oriented selective regression testing methods. These are Gupta, Harrold, and Soffa's dataflow (DF) algorithms (Gupta et al., 1996), and Leung and White's segment-Firewall (SFW) algorithm (Leung and White, 1992).

## 6.1 TEST ENVIRONMENT

The experiments have been done on a PC running a Pentium II 233 Mhz CPU. Seventeen program modules (M1-M17) are used, for which the flowgraphs and the initial suite of test cases, *TS*, have been manually generated. These modules are described in Table 6.1, where the size (in Lines Of Code), number of segments (M), and McCabe's cyclomatic number (*v*) are also given for each module. The cyclomatic number is included to give an idea about the complexity of the modules. These modules are of small size and simple complexity (M1-M8) and of medium size and complexity (M9-M17). For each module, we have also derived the table of the segment/requirement coverage information of the *n* test cases used. We use different combinations of values for *n* and the location/number of the modified segment, $S_{mod}$, in order to generate 60 test problems from the 17 modules. These problems are identified in the table of results (Table 6.2, Table 6.4) by *MiTj*, where *i* refers to the module (1-17) and *j* is a sequential number for a test problem for module *i*. For all these problems, we have designed *TS* to at least include tests that cover all the independent paths suggested by *v*. This approach ensures that all requirements are covered at least once.

*Table 6.1 Test modules*

| *Module* | ***Function*** | *M* | *LOC* | *v* |
|---|---|---|---|---|
| M1 | Determines a triangle's type | 11 | 24 | 3 |
| M2 | Calculates the sum and average of 100 or fewer bounded numbers | 13 | 21 | 6 |
| M3 | Determines the root of an algebraic equation iteratively | 13 | 38 | 5 |
| M4 | Reads a sequence of three binary bits and determines the implied status | 14 | 32 | 7 |
| M5 | Searches for a palindrome in a string | 15 | 31 | 8 |
| M6 | Raises a number to the appropriate power and displays the results | 16 | 46 | 5 |
| M7 | Reads a string and replaces it with an encoded character | 16 | 22 | 8 |
| M8 | Code containing nested loops and case statements | 16 | 21 | 7 |
| M9 | Code from phase II of an exam scheduling code | 54 | 162 | 27 |
| M10 | Code from phase II of an exam scheduling code | 60 | 170 | 31 |
| M11 | Phase I of an exam scheduling code | 63 | 229 | 33 |
| M12 | Code from phase II of an exam scheduling code | 69 | 203 | 35 |
| M13 | Code from phase II of an exam scheduling code | 75 | 223 | 39 |
| M14 | Code from phase III of an exam scheduling code | 90 | 350 | 44 |
| M15 | Code from phase II of an exam scheduling code | 91 | 364 | 42 |
| M16 | Phase III of an exam scheduling code | 104 | 381 | 49 |
| M17 | Phase II of an exam scheduling code | 109 | 313 | 58 |

## 6.2 REDUCTION-BASED METHODS-EXPERIMENTAL COMPARISONS AND RESULTS

In this section, we briefly explain the four quantitative criteria used for evaluating and comparing the proposed PR, MBR1, and MBR2 to the other three minimization-based regression testing algorithms. These are SA, RED, and SLI. We present the experimental results, comparisons, and discussions based on these criteria.

### 6.2.1 Quantitative criteria

The quantitative criteria used in the comparison are the test cases selection percentage, algorithm's execution time, precision, and inclusiveness:

(i) Percentage of test selection refers to the percentage of test cases selected by an algorithm from TS to rerun for revalidation, denoted by *% selection*;

(ii) Execution time of a regression-testing algorithm, denoted as $t_{exec}$;

(iii) Precision: Precision is defined in terms of modification-revealing and modification-traversing tests. Modification-revealing tests are those that produce different outputs for the modified program from those for the original version. Modification-traversing tests are those that execute modified code. Obviously, not all modification-traversing tests are modification-revealing. Precision measures the ability of an algorithm to omit non-modification-revealing tests that will not cause the modified program to produce different output (Rothermel and Harrold, 1996). If the initial test suite *TS* contains *nmr* non-modification-revealing tests, and a regression testing algorithm omits *onmr* of these tests. That is, it selects *snmr (= nmr-onmr)* of them. Then, the precision of the algorithm is given by the ratio *(onmr/nmr)*, where $nmr \neq 0$;

(iv) Inclusiveness: Inclusiveness measures the extent to which a regression testing algorithm selects modification-revealing tests that will cause the modified program to produce different output (Rothermel and Harrold, 1996). If the test suite T contains *mr* modification-revealing tests, and a regression testing algorithm selects smr of these tests, then the inclusiveness of the algorithm is given by the ratio *(smr/mr)*, where $mr \neq 0$.

### 6.2.2 Selection percentages and execution time results

Table 6.2 gives the percentage of selection, % selection, and the execution times, $t_{exec}$, in seconds, of the six algorithms for the 60 test problems. Table 6.3 gives aggregate results based on those in Table 6.2. We note that the time 0.00 in Table 6.2 means very small time, namely less than 0.01 seconds. We also note that the small magnitude of the execution times reported in Table 6.2 might look insignificant to justify a comparison of the speed of the five algorithms. But, since typical maintenance practices imply that changes to a program are done in a large number of small modification steps, the comparison of $t_{exec}$ of the algorithms becomes useful.

*Table 6.2. Selection (%) and execution times (sec) for the six algorithms*

| Ref | Test Problem | M | N | U | $S_{mod}$ | % Selection | | | | | | $t_{exec}$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | SA | RED | SLI | PR | MBR 1 | MBR 2 | SA | RED | SLI | PR | MBR 1 | MBR 2 |
| 1 | M1T1 | 11 | 20 | 5 | 5 | 10.0 | 15.0 | 30.0 | 5.0 | 10.0 | 5.0 | 0.38 | 0.00 | 2.25 | 0.00 | 0.00 | 0.00 |
| 2 | M2T1 | 13 | 12 | 6 | 6 | 8.3 | 16.7 | 16.7 | 8.3 | 16.7 | 8.3 | 0.22 | 0.00 | 2.25 | 0.00 | 0.00 | 0.00 |
| 3 | M3T1 | 13 | 8 | 5 | 8 | 25.0 | 25.0 | 25.0 | 12.5 | 25.0 | 12.5 | 1.64 | 0.00 | 1.86 | 0.00 | 0.00 | 0.00 |
| 4 | M3T2 | 13 | 16 | 5 | 8 | 12.5 | 12.5 | 25.0 | 6.3 | 12.5 | 6.3 | 0.33 | 0.00 | 10.05 | 0.00 | 0.00 | 0.00 |
| 5 | M3T3 | 13 | 32 | 5 | 7 | 6.3 | 6.3 | 25.0 | 3.1 | 6.3 | 6.3 | 0.66 | 0.00 | 5.65 | 0.00 | 0.00 | 0.00 |
| 6 | M3T4 | 13 | 64 | 5 | 7 | 3.1 | 3.1 | 32.8 | 1.6 | 3.1 | 1.6 | 1.54 | 0.00 | 15.54 | 0.00 | 0.00 | 0.00 |
| 7 | M3T5 | 13 | 64 | 5 | 5 | 3.1 | 3.1 | 34.4 | 1.6 | 3.1 | 3.1 | 1.54 | 0.00 | 1.92 | 0.00 | 0.00 | 0.00 |
| 8 | M4T1 | 14 | 8 | 7 | 12 | 50.0 | 50.0 | 12.5 | 12.5 | 50.0 | 50.0 | 0.11 | 0.00 | 27.14 | 0.00 | 0.00 | 0.00 |
| 9 | M4T2 | 14 | 8 | 7 | 7 | 50.0 | 50.0 | 25.0 | 25.0 | 50.0 | 25.0 | 0.11 | 0.00 | 1.86 | 0.00 | 0.00 | 0.00 |
| 10 | M4T3 | 14 | 32 | 7 | 7 | 12.5 | 12.5 | 28.1 | 6.3 | 12.5 | 6.3 | 0.66 | 0.00 | 43.90 | 0.00 | 0.00 | 0.00 |
| 11 | M5T1 | 15 | 48 | 8 | 13 | 4.2 | 2.1 | 12.5 | 2.1 | 2.1 | 2.1 | 0.99 | 0.00 | 34.56 | 0.00 | 0.00 | 0.00 |
| 12 | M5T2 | 15 | 96 | 8 | 7 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.16 | 0.00 | 39.23 | 0.00 | 0.00 | 0.00 |
| 13 | M6T1 | 16 | 6 | 6 | 13 | 33.3 | 66.7 | 16.7 | 16.7 | 33.3 | 33.3 | 0.11 | 0.00 | 2.25 | 0.00 | 0.00 | 0.00 |
| 14 | M6T2 | 16 | 48 | 6 | 13 | 4.2 | 8.3 | 16.7 | 4.2 | 4.2 | 4.2 | 0.99 | 0.00 | 3.90 | 0.00 | 0.00 | 0.00 |
| 15 | M6T3 | 16 | 48 | 6 | 6 | 4.2 | 4.2 | 35.4 | 4.2 | 4.2 | 4.2 | 0.99 | 0.00 | 4.83 | 0.05 | 0.00 | 0.00 |
| 16 | M7T1 | 16 | 65 | 8 | 7 | 1.5 | 1.5 | 1.5 | 1.5 | 1.5 | 1.5 | 1.59 | 0.00 | 2.25 | 0.00 | 0.00 | 0.00 |
| 17 | M8T1 | 16 | 8 | 7 | 6 | 50.0 | 50.0 | 12.5 | 12.5 | 50.0 | 12.5 | 1.09 | 0.00 | 5.99 | 0.00 | 0.00 | 0.00 |
| 18 | M9T1 | 54 | 120 | 27 | 13 | 1.7 | 4.2 | 1.7 | 1.7 | 4.2 | 2.5 | 3.02 | 0.05 | 2.25 | 0.00 | 0.00 | 0.05 |
| 19 | M9T2 | 54 | 120 | 27 | 0 | 1.7 | 4.2 | 2.5 | 1.7 | 4.2 | 2.5 | 3.02 | 0.05 | 2.04 | 0.00 | 0.00 | 0.05 |
| 20 | M9T3 | 54 | 90 | 27 | 13 | 3.3 | 5.6 | 7.8 | 4.4 | 5.6 | 4.4 | 2.36 | 0.05 | 8.82 | 0.00 | 0.00 | 0.05 |
| 21 | M9T4 | 54 | 90 | 27 | 16 | 3.3 | 5.6 | 6.7 | 4.4 | 5.6 | 4.4 | 2.36 | 0.05 | 2.52 | 0.00 | 0.00 | 0.05 |
| 22 | M10T1 | 61 | 80 | 31 | 15 | 2.5 | 6.3 | 10.0 | 3.8 | 5.0 | 5.0 | 2.74 | 0.05 | 4.25 | 0.00 | 0.00 | 0.00 |
| 23 | M10T2 | 61 | 80 | 31 | 55 | 2.5 | 6.3 | 10.0 | 3.8 | 5.0 | 3.8 | 2.74 | 0.05 | 5.87 | 0.00 | 0.00 | 0.00 |
| 24 | M10T3 | 61 | 60 | 31 | 53 | 5.0 | 8.3 | 11.7 | 5.0 | 5.0 | 5.0 | 2.74 | 0.05 | 6.31 | 0.00 | 0.00 | 0.00 |
| 25 | M10T4 | 61 | 60 | 31 | 17 | 5.0 | 8.3 | 5.0 | 5.0 | 8.3 | 8.3 | 2.74 | 0.05 | 4.60 | 0.00 | 0.00 | 0.00 |
| 26 | M11T1 | 63 | 101 | 33 | 16 | 3.0 | 5.0 | 5.0 | 2.0 | 5.0 | 5.0 | 2.75 | 1.97 | 7.47 | 0.00 | 1.97 | 0.00 |
| 27 | M11T2 | 63 | 101 | 33 | 61 | 3.0 | 5.0 | 6.9 | 3.0 | 5.0 | 5.0 | 2.75 | 1.97 | 10.77 | 0.00 | 1.97 | 0.00 |
| 28 | M11T3 | 63 | 101 | 33 | 12 | 3.0 | 5.0 | 3.0 | 2.0 | 5.0 | 5.0 | 2.75 | 1.97 | 2.69 | 0.00 | 1.97 | 0.00 |
| 29 | M11T4 | 63 | 101 | 33 | 17 | 3.0 | 5.0 | 5.0 | 2.0 | 5.0 | 5.0 | 2.75 | 1.97 | 7.41 | 0.00 | 1.97 | 0.00 |
| 30 | M11T5 | 63 | 101 | 33 | 19 | 3.0 | 5.0 | 5.0 | 2.0 | 5.0 | 5.0 | 2.75 | 1.97 | 2.69 | 0.00 | 1.97 | 0.00 |
| 31 | M12T1 | 69 | 94 | 35 | 67 | 1.1 | 6.4 | 2.1 | 1.1 | 6.4 | 6.4 | 2.53 | 0.00 | 6.86 | 0.00 | 0.00 | 0.05 |
| 32 | M12T2 | 69 | 94 | 35 | 8 | 1.1 | 6.4 | 2.1 | 2.1 | 6.4 | 4.3 | 2.53 | 0.00 | 4.01 | 0.00 | 0.00 | 0.05 |
| 33 | M12T3 | 69 | 94 | 35 | 13 | 1.1 | 6.4 | 3.2 | 2.1 | 6.4 | 6.4 | 2.53 | 0.00 | 5.16 | 0.00 | 0.00 | 0.05 |
| 34 | M12T4 | 69 | 130 | 35 | 36 | 2.3 | 3.1 | 58.5 | 3.1 | 3.1 | 2.3 | 3.57 | 0.05 | 9.45 | 0.05 | 0.05 | 0.00 |
| 35 | M12T5 | 69 | 130 | 35 | 2 | 2.3 | 3.1 | 63.8 | 3.1 | 3.1 | 2.3 | 3.57 | 0.05 | 3.68 | 0.00 | 0.05 | 0.05 |
| 36 | M13T1 | 75 | 180 | 39 | 3 | 2.8 | 1.7 | 97.2 | 2.2 | 1.7 | 1.7 | 4.84 | 0.11 | 2.53 | 0.05 | 0.11 | 0.05 |
| 37 | M13T2 | 75 | 165 | 39 | 68 | 2.4 | 3.0 | 93.3 | 3.0 | 3.0 | 2.4 | 4.56 | 0.00 | 3.24 | 0.05 | 0.00 | 0.11 |
| 38 | M13T3 | 75 | 165 | 39 | 16 | 2.4 | 3.0 | 97.0 | 2.4 | 3.0 | 2.4 | 4.56 | 0.00 | 12.86 | 0.05 | 0.00 | 0.11 |
| 39 | M13T4 | 75 | 93 | 39 | 72 | 4.3 | 6.5 | 2.2 | 1.1 | 6.5 | 5.4 | 2.58 | 0.00 | 4.56 | 0.00 | 0.00 | 0.00 |
| 40 | M13T5 | 75 | 93 | 39 | 71 | 4.3 | 6.5 | 1.1 | 1.1 | 6.5 | 2.2 | 2.58 | 0.00 | 3.84 | 0.00 | 0.00 | 0.00 |
| 41 | M13T6 | 75 | 93 | 39 | 2 | 4.3 | 6.5 | 1.1 | 1.1 | 6.5 | 5.4 | 2.58 | 0.00 | 10.27 | 0.00 | 0.00 | 0.00 |
| 42 | M13T7 | 75 | 93 | 39 | 10 | 4.3 | 6.5 | 1.1 | 1.1 | 6.5 | 6.5 | 2.58 | 0.00 | 3.84 | 0.00 | 0.00 | 0.05 |
| 43 | M13T8 | 75 | 93 | 39 | 13 | 4.3 | 6.5 | 1.1 | 1.1 | 6.5 | 3.2 | 2.58 | 0.00 | 3.15 | 0.00 | 0.00 | 0.00 |
| 44 | M14T1 | 90 | 65 | 44 | 5 | 10.8 | 10.8 | 3.1 | 3.1 | 10.8 | 10.8 | 1.76 | 0.00 | 20.98 | 0.00 | 0.00 | 0.00 |
| 45 | M14T2 | 90 | 65 | 44 | 1 | 10.8 | 10.8 | 1.5 | 1.5 | 10.8 | 10.8 | 1.76 | 0.00 | 53.35 | 0.00 | 0.00 | 0.00 |
| 46 | M14T3 | 90 | 95 | 44 | 76 | 4.2 | 7.4 | 23.2 | 3.2 | 7.4 | 2.1 | 2.69 | 0.00 | 11.87 | 0.00 | 0.00 | 0.00 |
| 47 | M15T1 | 91 | 60 | 42 | 62 | 5.0 | 8.3 | 8.3 | 5.0 | 6.7 | 6.7 | 2.74 | 0.00 | 2.47 | 0.00 | 0.00 | 0.00 |
| 48 | M15T2 | 91 | 60 | 42 | 17 | 5.0 | 8.3 | 6.7 | 6.7 | 8.3 | 8.3 | 2.74 | 0.00 | 4.00 | 0.00 | 0.00 | 0.00 |
| 49 | M15T3 | 91 | 80 | 42 | 75 | 2.5 | 6.3 | 1.3 | 1.3 | 5.0 | 5.0 | 2.74 | 0.00 | 2.80 | 0.00 | 0.00 | 0.00 |
| 50 | M15T4 | 91 | 80 | 42 | 17 | 2.5 | 6.3 | 3.8 | 3.8 | 6.3 | 6.3 | 2.74 | 0.00 | 11.00 | 0.00 | 0.00 | 0.00 |
| 51 | M15T5 | 91 | 80 | 42 | 77 | 2.5 | 6.3 | 2.5 | 2.5 | 5.0 | 5.0 | 2.74 | 0.00 | 2.85 | 0.00 | 0.00 | 0.00 |
| 52 | M15T6 | 91 | 80 | 42 | 66 | 2.5 | 6.3 | 1.3 | 1.3 | 5.0 | 5.0 | 2.74 | 0.00 | 3.86 | 0.00 | 0.00 | 0.00 |
| 53 | M16T1 | 104 | 65 | 49 | 6 | 6.2 | 10.8 | 3.1 | 3.1 | 10.8 | 9.2 | 1.87 | 0.00 | 2.03 | 0.00 | 0.00 | 0.00 |
| 54 | M16T2 | 104 | 65 | 49 | 49 | 6.2 | 10.8 | 3.1 | 3.1 | 10.8 | 9.2 | 1.87 | 0.00 | 4.95 | 0.00 | 0.00 | 0.00 |
| 55 | M16T3 | 104 | 65 | 49 | 65 | 6.2 | 10.8 | 1.5 | 1.5 | 10.8 | 4.6 | 1.87 | 0.00 | 20.93 | 0.00 | 0.00 | 0.00 |
| 56 | M16T4 | 104 | 65 | 49 | 101 | 6.2 | 10.8 | 4.6 | 4.6 | 10.8 | 4.6 | 1.87 | 0.00 | 4.01 | 0.00 | 0.00 | 0.00 |
| 57 | M17T1 | 109 | 96 | 58 | 72 | 5.2 | 7.3 | 94.8 | 7.3 | 7.3 | 7.3 | 2.91 | 0.00 | 5.88 | 0.00 | 0.00 | 0.00 |
| 58 | M17T2 | 109 | 96 | 58 | 2 | 5.2 | 7.3 | 94.8 | 7.3 | 7.3 | 6.3 | 2.91 | 0.00 | 2.87 | 0.00 | 0.00 | 0.05 |
| 59 | M17T3 | 109 | 120 | 58 | 0 | 3.3 | 4.2 | 95.8 | 5.8 | 4.2 | 4.2 | 3.57 | 0.05 | 5.49 | 0.05 | 0.05 | 0.05 |
| 60 | M17T4 | 109 | 80 | 58 | 2 | 7.5 | 5.0 | 48.8 | 5.0 | 5.0 | 5.0 | 2.09 | 0.05 | 2.11 | 0.00 | 0.05 | 0.05 |

The least percentages of selection for 73% of the time, as shown in Table 6.3, are recorded by our proposed PR algorithm. Such small selection percentages are expected since PR selects only modification-revealing tests and omits all non modification-revealing and redundant ones. A redundant test case is one that covers the same structural or functional requirement as another test case in *TS*. The selected tests are sufficient to provide testing coverage of the relevant slices containing a modified requirement(s) and might affect the output of the maintained program. Thus, the minimal number of the selected tests is not obtained on the expense of the desired coverage. As far as the execution time is concerned, Tables 6.2 and 6.3 show that PR is the fastest of all the algorithms.

*Table 6.3. Aggregate results for the 60 test problems*

| *% Times* | *SA* | *RED* | *SLI* | *PR* | *MBR1* | *MBR2* |
|---|---|---|---|---|---|---|
| Least #R | 43% | 10% | 37% | 72% | 13% | 37% |
| Worst #R | 10% | 55% | 52% | 0% | 47% | 22% |
| Fastest texec. | 0% | 73% | 0% | 95% | 87% | 75% |
| Slowest texec. | 15% | 0% | 83% | 0% | 0% | 0% |

The test selection percentages of the MBR1 algorithm record the least for only 13% of the time ranking it the fourth as far as the least test selection percentage is concerned. The MBR1 algorithm shows only 17% improvement in selecting fewer tests when compared to RED. Such little improvement is related to the fact that most of the modules used exhibit strong dependency; hence, most of the segments are said to be reachable from/reach the modified one. Under such conditions, MBR1 tends to test for almost all segments and such results do not reflect an exact behavior of the algorithm. However, the test selection percentage varies with the location of the modified segment(s) and the degree of reachability it exhibits with other segments in a particular program. This makes MBR1 more suitable for problems with components exhibiting low dependency. Like RED and SA, MBR1 is capable to omit all redundant test cases, a feature inherited from RED. MBR1 is the second fastest among the five algorithms according to Tables 6.2 and 6.3.

The test selection percentages of the MBR2 algorithm record least for 37% of the time ranking it the third as far as the least test selection percentage is concerned. The MBR2 algorithm shows 47 % improvement in selecting fewer tests when compared to MBR1. Such improvement is related to the fact that MBR2 omits tests that do not reach the modification. Like RED, SA, and MBR1, MBR2 is capable to omit all redundant test cases, a feature inherited from RED. MBR2 is the third fastest among the six algorithms according to Tables 6.2 and 6.3.

The SA test selection percentages ranks the second for 43% of the time. Such low percentages are expected since SA selects only the test cases that are necessary to execute (once) the modified segment and all the segments reachable from it. Thus, SA omits many redundant test cases. A low test selection percentage would still be obtained by SA even if all segments are reachable from all other segments (due to an outer loop). However, Tables 6.2 and 6.3 show that SA is the second slowest among the five algorithms.

The RED algorithm also selects a small number of retests. Its test selection percentages record the lowest for 10% of the test problems. RED is also capable to omit redundant test cases. This capability places it close to MBR1 as far as the test selection percentage is concerned. Its execution time is also quite low and makes RED among the fastest (and close to) MBR2 and MBR1 algorithms. The RED algorithm has a unique weakness in comparison with the other four algorithms: the location of the modified segment has no effect on the selected tests when the technique addresses regression testing.

For the SLI algorithm, the test selection percentage is highly dependent on the location of the modified segment with respect to the dynamic slices of the test cases. That is why, Table 6.2 shows sharp changes in percentages for different values of $S_{mod}$ in the same module. For some test problems in Table 6.2 (e.g.; see problems 36-38), where $S_{mod}$ is in many dynamic slices, the selection percentage even approaches 97%. For these problems, SLI does not exclude redundant test cases. For some other problems, where the modified segment is included in one dynamic slice, SLI selects only one retest. So, although Table 6.3 reveals that SLI yields the lowest selection percentages for 37% of the time, this does not indicate a trend for SLI's behavior due to its strong dependency on $S_{mod}$. Further, Tables 6.2 and 6.3 clearly show that SLI is the slowest among the algorithms.

### 6.2.3 Precision and inclusiveness

Table 6.4 gives the precision and inclusiveness results for the 60 test problems. Table 6.5 gives the aggregate results. Table 6.6 illustrates in detail how precision and inclusiveness values are computed for two test problems.

*Table 6.4. Precision and inclusiveness percentage results for the six algorithms*

| *Ref.* | *Test Problem* | *Precision %* | | | | | | *Inclusiveness %* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *SA* | *RED* | *SLI* | *PR* | *MBR 1* | *MBR 2* | *SA* | *RED* | *SLI* | *PR* | *MBR 1* | *MBR 2* |
| *1* | *M1T1* | 80 | 60 | 100 | 100 | 100 | 100 | 7 | 7 | 40 | 7 | 13 | 7 |
| *2* | *M2T1* | 75 | 100 | 100 | 100 | 100 | 100 | 0 | 25 | 25 | 13 | 25 | 13 |
| *3* | *M3T1* | 50 | 50 | 100 | 100 | 50 | 100 | 17 | 17 | 33 | 17 | 17 | 17 |
| *4* | *M3T2* | 75 | 75 | 100 | 100 | 75 | 100 | 8 | 8 | 33 | 8 | 8 | 8 |
| *5* | *M3T3* | 88 | 88 | 100 | 100 | 88 | 88 | 4 | 4 | 33 | 4 | 4 | 4 |
| *6* | *M3T4* | 94 | 94 | 100 | 100 | 94 | 100 | 2 | 2 | 44 | 2 | 2 | 2 |
| *7* | *M3T5* | 94 | 94 | 100 | 100 | 94 | 100 | 2 | 2 | 46 | 2 | 2 | 4 |
| *8* | *M4T1* | 50 | 50 | 100 | 100 | 50 | 50 | 50 | 50 | 17 | 17 | 50 | 50 |
| *9* | *M4T2* | 50 | 50 | 100 | 100 | 50 | 100 | 50 | 50 | 33 | 33 | 50 | 33 |
| *10* | *M4T3* | 75 | 88 | 100 | 100 | 88 | 100 | 8 | 13 | 38 | 8 | 13 | 8 |
| *11* | *M5T1* | 90 | 100 | 100 | 100 | 100 | 100 | 3 | 3 | 16 | 3 | 3 | 3 |
| *12* | *M5T2* | 100 | 100 | 100 | 100 | 100 | 100 | 1 | 1 | 1 | 1 | 1 | 1 |
| *13* | *M6T1* | 50 | 50 | 100 | 100 | 100 | 100 | 25 | 75 | 25 | 25 | 50 | 50 |
| *14* | *M6T2* | 78 | 89 | 100 | 100 | 100 | 100 | 0 | 8 | 21 | 5 | 5 | 5 |
| *15* | *M6T3* | 100 | 100 | 100 | 100 | 100 | 100 | 5 | 5 | 44 | 5 | 5 | 5 |
| *16* | *M7T1* | 88 | 100 | 100 | 100 | 100 | 100 | 0 | 2 | 2 | 2 | 2 | 2 |
| *17* | *M8T1* | 50 | 50 | 100 | 100 | 50 | 100 | 50 | 50 | 17 | 17 | 50 | 17 |
| *18* | *M9T1* | 98 | 97 | 100 | 100 | 97 | 99 | 0 | 25 | 50 | 50 | 25 | 50 |
| *19* | *M9T2* | 98 | 97 | 100 | 100 | 97 | 99 | 0 | 25 | 75 | 50 | 25 | 50 |
| *20* | *M9T3* | 97 | 99 | 100 | 100 | 99 | 99 | 8 | 31 | 54 | 31 | 31 | 23 |
| *21* | *M9T4* | 97 | 99 | 100 | 100 | 99 | 99 | 8 | 31 | 46 | 31 | 31 | 23 |
| *22* | *M10T1* | 97 | 91 | 100 | 100 | 94 | 97 | 2 | 4 | 17 | 6 | 4 | 6 |
| *23* | *M10T2* | 97 | 91 | 100 | 100 | 94 | 97 | 2 | 4 | 18 | 7 | 4 | 4 |
| *24* | *M10T3* | 95 | 85 | 100 | 100 | 95 | 100 | 5 | 5 | 18 | 8 | 5 | 8 |
| *25* | *M10T4* | 97 | 83 | 100 | 100 | 90 | 97 | 7 | 0 | 10 | 10 | 7 | 13 |
| *26* | *M11T1* | 97 | 96 | 100 | 100 | 96 | 98 | 0 | 14 | 71 | 29 | 14 | 43 |
| *27* | *M11T2* | 97 | 96 | 100 | 100 | 96 | 97 | 0 | 14 | 100 | 43 | 14 | 29 |
| *28* | *M11T3* | 97 | 96 | 100 | 100 | 96 | 98 | 0 | 14 | 43 | 29 | 14 | 43 |
| *29* | *M11T4* | 97 | 96 | 100 | 100 | 96 | 96 | 0 | 14 | 71 | 29 | 14 | 14 |
| *30* | *M11T5* | 97 | 96 | 100 | 100 | 96 | 97 | 0 | 14 | 71 | 29 | 14 | 29 |
| *31* | *M12T1* | 99 | 93 | 100 | 100 | 93 | 96 | 0 | 0 | 40 | 20 | 0 | 40 |
| *32* | *M12T2* | 99 | 93 | 100 | 100 | 93 | 98 | 0 | 0 | 40 | 40 | 0 | 40 |
| *33* | *M12T3* | 99 | 93 | 100 | 100 | 93 | 97 | 0 | 0 | 60 | 40 | 0 | 60 |
| *34* | *M12T4* | 98 | 94 | 100 | 100 | 94 | 100 | 2 | 1 | 92 | 5 | 1 | 4 |
| *35* | *M12T5* | 98 | 94 | 100 | 100 | 94 | 98 | 2 | 1 | 100 | 5 | 1 | 2 |
| *36* | *M13T1* | 80 | 60 | 100 | 100 | 60 | 100 | 2 | 1 | 100 | 2 | 1 | 2 |
| *37* | *M13T2* | 100 | 100 | 100 | 100 | 100 | 100 | 3 | 3 | 96 | 3 | 3 | 3 |
| *38* | *M13T3* | 100 | 100 | 100 | 100 | 100 | 100 | 3 | 3 | 100 | 3 | 3 | 3 |
| *39* | *M13T4* | 96 | 93 | 100 | 100 | 93 | 97 | 0 | 0 | 67 | 33 | 0 | 67 |
| *40* | *M13T5* | 96 | 93 | 100 | 100 | 93 | 100 | 0 | 0 | 33 | 33 | 0 | 67 |
| *41* | *M13T6* | 96 | 93 | 100 | 100 | 93 | 98 | 0 | 0 | 33 | 33 | 0 | 100 |
| *42* | *M13T7* | 96 | 93 | 100 | 100 | 93 | 98 | 0 | 0 | 33 | 33 | 0 | 100 |
| *43* | *M13T8* | 96 | 93 | 100 | 100 | 93 | 100 | 0 | 0 | 33 | 33 | 0 | 100 |
| *44* | *M14T1* | 91 | 91 | 100 | 100 | 91 | 93 | 29 | 29 | 29 | 29 | 29 | 43 |
| *45* | *M14T2* | 91 | 91 | 100 | 100 | 91 | 91 | 29 | 29 | 14 | 14 | 29 | 29 |
| *46* | *M14T3* | 96 | 91 | 100 | 100 | 91 | 100 | 4 | 4 | 88 | 12 | 4 | 8 |
| *47* | *M15T1* | 95 | 91 | 100 | 100 | 95 | 95 | 5 | 8 | 13 | 8 | 8 | 8 |
| *48* | *M15T2* | 97 | 91 | 100 | 100 | 94 | 97 | 8 | 8 | 16 | 16 | 12 | 16 |
| *49* | *M15T3* | 97 | 88 | 100 | 100 | 94 | 94 | 2 | 2 | 2 | 2 | 4 | 4 |
| *50* | *M15T4* | 98 | 93 | 100 | 100 | 93 | 95 | 3 | 5 | 8 | 8 | 5 | 8 |
| *51* | *M15T5* | 97 | 91 | 100 | 100 | 97 | 97 | 2 | 4 | 4 | 4 | 6 | 6 |
| *52* | *M15T6* | 97 | 91 | 100 | 100 | 97 | 97 | 2 | 4 | 2 | 2 | 6 | 6 |
| *53* | *M16T1* | 97 | 92 | 100 | 100 | 92 | 95 | 50 | 50 | 100 | 50 | 50 | 75 |
| *54* | *M16T2* | 97 | 92 | 100 | 100 | 92 | 95 | 50 | 50 | 50 | 50 | 50 | 50 |
| *55* | *M16T3* | 97 | 92 | 100 | 100 | 92 | 98 | 50 | 50 | 25 | 25 | 50 | 50 |
| *56* | *M16T4* | 97 | 92 | 100 | 100 | 92 | 100 | 50 | 50 | 75 | 75 | 50 | 75 |
| *57* | *M17T1* | 100 | 0 | 100 | 100 | 0 | 40 | 5 | 2 | 100 | 8 | 2 | 4 |
| *58* | *M17T2* | 100 | 0 | 100 | 100 | 0 | 40 | 5 | 2 | 100 | 8 | 2 | 3 |
| *59* | *M17T3* | 100 | 100 | 100 | 100 | 100 | 100 | 3 | 4 | 100 | 6 | 4 | 4 |
| *60* | *M17T4* | 90 | 93 | 100 | 100 | 93 | 98 | 5 | 3 | 100 | 10 | 3 | 8 |

*Table 6.5. Inclusiveness and precision Aggregate results for the 60 test problems*

| *% Times* | *SA* | *RED* | *SLI* | *PR* | *MBR1* | *MBR2* |
|---|---|---|---|---|---|---|
| Highest Precision | 13% | 13% | 100% | 100% | 18% | 40% |
| Lowest Precision | 30% | 78% | 0% | 0% | 60% | 5% |
| Highest Inclusiveness | 12% | 15% | 83% | 12% | 12% | 21% |
| Lowest Inclusiveness | 68% | 54% | 18% | 37% | 46% | 21% |

*Table 6.6. Comparison of detailed precision and inclusiveness results for two test problems*

| *Algorithm* | *N* | *mr* | *nmr* | *#R* | *smr* | *snmr* | *Onmr* | *Precision %* | *Inclusiveness %* |
|---|---|---|---|---|---|---|---|---|---|
| SA | 65 | 4 | 61 | 4 | 2 | 2 | 59 | 97 | 50 |
| RED | 65 | 4 | 61 | 7 | 2 | 5 | 56 | 92 | 50 |
| SLI | 65 | 4 | 61 | 2 | 2 | 0 | 61 | 100 | 50 |
| PR | 65 | 4 | 61 | 2 | 2 | 0 | 61 | 100 | 50 |
| MBR1 | 65 | 4 | 61 | 7 | 2 | 5 | 56 | 92 | 50 |
| MBR2 | 65 | 4 | 61 | 6 | 3 | 3 | 3 | 95 | 75 |

*(a) Test problem M16T1*

| *Algorithm* | *N* | *mr* | *nmr* | *#R* | *smr* | *snmr* | *Onmr* | *Precision %* | *Inclusiveness %* |
|---|---|---|---|---|---|---|---|---|---|
| SA | 96 | 91 | 5 | 5 | 5 | 0 | 5 | 100 | 5 |
| RED | 96 | 91 | 5 | 7 | 2 | 5 | 0 | 0 | 2 |
| SLI | 96 | 91 | 5 | 91 | 91 | 0 | 5 | 100 | 100 |
| PR | 96 | 91 | 5 | 7 | 7 | 0 | 5 | 100 | 8 |
| MBR1 | 96 | 91 | 5 | 7 | 2 | 5 | 0 | 0 | 2 |
| MBR2 | 96 | 91 | 5 | 7 | 4 | 3 | 2 | 40 | 4 |

*(b) Test problem M17T1*

Table 6.4 shows that the PR and SLI algorithms have the highest precision and inclusiveness among the five algorithms. This is due to their ability to omit all non-modification revealing test cases and select only ones that are modification revealing. That is why, SLI and PR are fully precise. SLI's inclusiveness depends on the location of the modified segment and, generally, not all modification-revealing test cases are selected. But, the inclusiveness of

PR depends on the selected non-redundant test cases whose relevant slice contain a modified segment(s) and is generally low: if several redundant test cases with relevant slices traversing a modified segments, PR attempts to select only one of them.

The SA, RED, MBR1, and MBR2 algorithms show similar precision and inclusiveness properties. Although they are not fully precise, they exhibit nearly-full precision for most of the 60 problems. The precision of MBR2 is improved over that of MBR1 and RED, for MBR2 omits tests that fail to execute the modification; hence, improving the chance of selecting modification-revealing tests. When testing for programs with segments exhibiting low reachability to the modified one, MBR1 tends to give higher precision values than that reported by RED (e.g., see problems 13 and 14 of Table 3). Interestingly, RED and MBR1 have zero precision for test problem M17T1. Table 6.6(b) shows that it happened that the 7 selected retests included all the 5 non-modification-revealing test cases in addition to only 2 modification revealing ones. However, the inclusiveness of SA, RED, MBR1, and MBR2 is low, since: (i) RED does not explicitly target modification revealing test cases due to its $S_{mod}$-independence, and (ii) if several modification revealing test cases traverse a particular segment, RED, MBR1, MBR2 and SA attempt to select only one of them.

For the quantitative criteria, we note that: (a) we are doing an assessment using only a limited number of test problems, (b) the test problems used are not random, (c) the test case design is essentially based on all-statement coverage and where the value of *n* is increased, arbitrary test cases were added, and (d) changing the location of $S_{mod}$ was done in an arbitrary, but not in a random way. Due to these considerations, we do not claim that our results reflect an exact behavior of the algorithms. But, the results indicate a trend of their behavior that is consistent with the theoretical interpretation.

## 6.3 McCABE-BASED METRICS-EXPERIMENTATIONS

In this section, we use the proposed STM and RTM metrics to evaluate the test selection adequacy for two coverage-based regression test selection algorithms. These are DF and SFW.

DF and SFW define all the definition-use pairs that may be affected by the modification and select test cases accordingly. The values of the #R are influenced by $S_{mod}$ and are variable and location dependent. These techniques do not exclude redundant test cases and yield to high test selection percentages that require a large regression testing effort, which is not always affordable. Therefore, it is suitable and would be favorable to use the STM bounds to monitor the testing coverage adequacy of these techniques.

### 6.3.1 STM results

For each of the 60 test problems, we have computed both the STM upper and lower bounds using the approaches discussed in chapter 5. Table 6.7 gives the computed bounds normalized to a percentage figure to allow the comparison with *selection%*. Figure 6.1 sketches samples of the results for some test problems. The computed lower bound quantifies the minimal testing effort required to test for an edit of a variable *var* at location $S_{mod}$ for test problem *MiTj*. It also indicates the minimal tests needed to rerun for revalidating the affected definition-use pairs. We note that any drop below the computed lower bound indicates shortage in retests and inadequacy of testing coverage. The STM upper bound gives an indication of the sufficient tests required for revalidating the change. Selected retests exceeding the upper bound are considered as redundant tests. We have used Table 5.2 to interpret the results.

Comparing the % selection with the computed STM lower bound, Table 6.6 shows that none of the selected retests of the 60 test problems has dropped below the STM lower bound for either DF or SFW. This could be referred to the fact that *TS* is developed using *v* to include tests that exercise at least the independent basis paths in the program. Such approach has provided at least a minimal adequate coverage of all the affected definition-use pairs. Note that % lying on the lower bound indicate that a minimal testing coverage is attained.

Comparing the *% selection* with the computed STM upper bound, the same table indicates that *% selection* has exceeded the bound for some test problems (e.g., see problems sketched in Figure 6.1). Selection % exceeding the upper bound reveals redundancy in the selected tests, which is favorable to avoid. Under such conditions, regression testing for the STM upper bound would be favorable. An optional action in this case would require managing and controlling the test suite to eliminate redundant tests that exercise the same coverage.

*Table 6.7.Selection (%), RTM(%), and STM(%)*

| Ref | Test Problem | $M$ | $N$ | $v$ | $S_{mod}$ | %RTM | % STM | | % Selection | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | RU | SU | SL | SFW | DF |
| 1 | M1T1 | 11 | 20 | 5 | 5 | 20 | 15 | 15 | 60 | 25 |
| 2 | M2T1 | 13 | 12 | 6 | 6 | 50 | 8 | 8 | 17 | 17 |
| 3 | M3T1 | 13 | 8 | 5 | 8 | 75 | 50 | 13 | 50 | 38 |
| 4 | M3T2 | 13 | 16 | 5 | 8 | 38 | 25 | 6 | 31 | 38 |
| 5 | M3T3 | 13 | 32 | 5 | 7 | 19 | 13 | 6 | 16 | 13 |
| 6 | M3T4 | 13 | 64 | 5 | 7 | 9 | 6 | 3 | 22 | 13 |
| 7 | M3T5 | 13 | 64 | 5 | 5 | 9 | 8 | 3 | 66 | 75 |
| 8 | M4T1 | 14 | 8 | 7 | 12 | 88 | 75 | 50 | 50 | 50 |
| 9 | M4T2 | 14 | 8 | 7 | 7 | 88 | 75 | 50 | 38 | 100 |
| 10 | M4T3 | 14 | 32 | 7 | 7 | 22 | 19 | 13 | 72 | 22 |
| 11 | M5T1 | 15 | 48 | 8 | 13 | 15 | 10 | 8 | 33 | 50 |
| 12 | M5T2 | 15 | 96 | 8 | 7 | 7 | 5 | 2 | 76 | 67 |
| 13 | M6T1 | 16 | 6 | 6 | 13 | 67 | 50 | 17 | 83 | 17 |
| 14 | M6T2 | 16 | 48 | 6 | 13 | 8 | 6 | 2 | 69 | 58 |
| 15 | M6T3 | 16 | 48 | 6 | 6 | 6 | 4 | 4 | 92 | 13 |
| 16 | M7T1 | 16 | 65 | 8 | 7 | 12 | 9 | 5 | 38 | 57 |
| 17 | M8T1 | 16 | 8 | 7 | 6 | 88 | 50 | 13 | 25 | 13 |
| 18 | M9T1 | 54 | 120 | 27 | 13 | 23 | 15 | 2 | 43 | 63 |
| 19 | M9T2 | 54 | 120 | 27 | 0 | 23 | 7 | 2 | 78 | 45 |
| 20 | M9T3 | 54 | 90 | 27 | 13 | 30 | 20 | 2 | 38 | 50 |
| 21 | M9T4 | 54 | 90 | 27 | 16 | 30 | 13 | 3 | 10 | 47 |
| 22 | M10T1 | 61 | 80 | 31 | 15 | 35 | 33 | 3 | 36 | 45 |
| 23 | M10T2 | 61 | 80 | 31 | 55 | 36 | 36 | 1 | 5 | 13 |
| 24 | M10T3 | 61 | 60 | 31 | 53 | 48 | 48 | 2 | 5 | 10 |
| 25 | M10T4 | 61 | 60 | 31 | 17 | 48 | 48 | 7 | 28 | 40 |
| 26 | M11T1 | 63 | 101 | 33 | 16 | 33 | 33 | 1 | 11 | 28 |
| 27 | M11T2 | 63 | 101 | 33 | 61 | 33 | 33 | 3 | 70 | 77 |
| 28 | M11T3 | 63 | 101 | 33 | 12 | 33 | 33 | 3 | 35 | 25 |
| 29 | M11T4 | 63 | 101 | 33 | 17 | 33 | 33 | 5 | 52 | 76 |
| 30 | M11T5 | 63 | 101 | 33 | 19 | 33 | 33 | 3 | 71 | 76 |
| 31 | M12T1 | 69 | 94 | 35 | 67 | 37 | 37 | 5 | 30 | 55 |
| 32 | M12T2 | 69 | 94 | 35 | 8 | 37 | 37 | 2 | 50 | 31 |
| 33 | M12T3 | 69 | 94 | 35 | 13 | 37 | 37 | 2 | 46 | 66 |
| 34 | M12T4 | 69 | 130 | 35 | 36 | 27 | 27 | 8 | 73 | 37 |
| 35 | M12T5 | 69 | 130 | 35 | 2 | 27 | 27 | 2 | 68 | 41 |
| 36 | M13T1 | 75 | 180 | 39 | 3 | 22 | 22 | 2 | 98 | 57 |
| 37 | M13T2 | 75 | 165 | 39 | 68 | 24 | 24 | 2 | 73 | 30 |
| 38 | M13T3 | 75 | 165 | 39 | 16 | 24 | 24 | 1 | 61 | 64 |
| 39 | M13T4 | 75 | 93 | 39 | 72 | 42 | 42 | 3 | 38 | 51 |
| 40 | M13T5 | 75 | 93 | 39 | 71 | 42 | 42 | 2 | 49 | 45 |
| 41 | M13T6 | 75 | 93 | 39 | 2 | 42 | 42 | 2 | 67 | 40 |
| 42 | M13T7 | 75 | 93 | 39 | 10 | 42 | 42 | 2 | 55 | 58 |
| 43 | M13T8 | 75 | 93 | 39 | 13 | 42 | 42 | 2 | 48 | 35 |
| 44 | M14T1 | 90 | 65 | 44 | 5 | 68 | 60 | 3 | 15 | 38 |
| 45 | M14T2 | 90 | 65 | 44 | 1 | 68 | 60 | 3 | 15 | 45 |
| 46 | M14T3 | 90 | 95 | 44 | 76 | 46 | 41 | 3 | 4 | 42 |
| 47 | M15T1 | 91 | 60 | 42 | 62 | 63 | 63 | 3 | 63 | 62 |
| 48 | M15T2 | 91 | 60 | 42 | 17 | 67 | 67 | 7 | 28 | 40 |
| 49 | M15T3 | 91 | 80 | 42 | 75 | 48 | 48 | 1 | 39 | 13 |
| 50 | M15T4 | 91 | 80 | 42 | 17 | 50 | 50 | 5 | 36 | 45 |
| 51 | M15T5 | 91 | 80 | 42 | 77 | 48 | 46 | 1 | 38 | 14 |
| 52 | M15T6 | 91 | 80 | 42 | 66 | 48 | 48 | 3 | 40 | 58 |
| 53 | M16T1 | 104 | 65 | 49 | 6 | 75 | 75 | 6 | 94 | 48 |
| 54 | M16T2 | 104 | 65 | 49 | 49 | 75 | 75 | 5 | 29 | 37 |
| 55 | M16T3 | 104 | 65 | 49 | 65 | 75 | 75 | 3 | 100 | 52 |
| 56 | M16T4 | 104 | 65 | 49 | 101 | 75 | 75 | 5 | 100 | 51 |
| 57 | M17T1 | 109 | 96 | 58 | 72 | 60 | 60 | 3 | 66 | 59 |
| 58 | M17T2 | 109 | 96 | 58 | 2 | 60 | 60 | 2 | 78 | 46 |
| 59 | M17T3 | 109 | 120 | 58 | 0 | 48 | 48 | 2 | 97 | 49 |
| 60 | M17T4 | 109 | 80 | 58 | 2 | 73 | 73 | 3 | 76 | 51 |

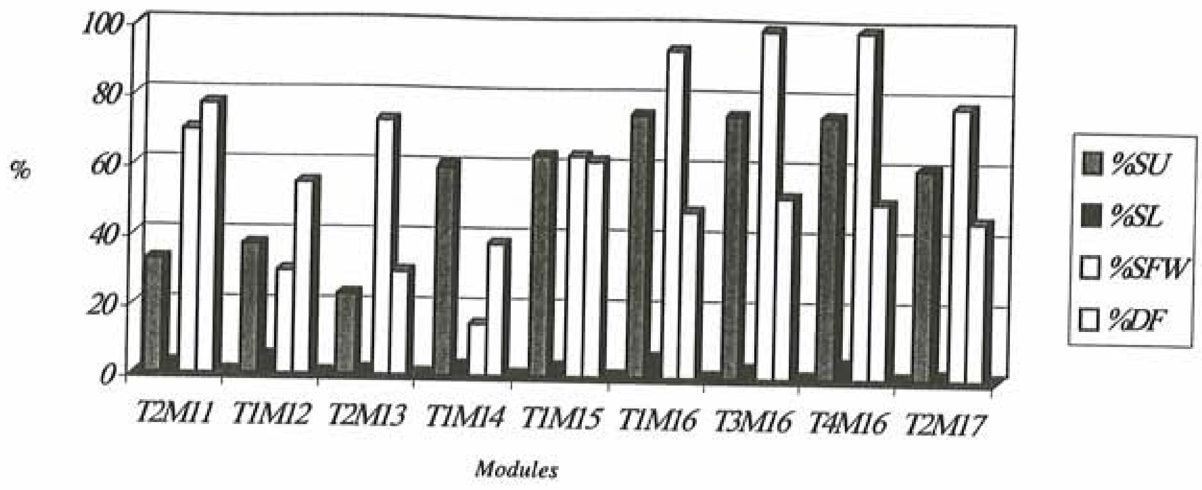

*Figure 6.1 STM samples from m11-m17*

For tests problems that report test selection ling between the two bounds, we note that as the number of the reported selection percentage approaches the upper bound, the chance of selecting modification revealing test is improved and the selection would be then characterized as more safe and reliable.

### 6.3.2 RTM results

For each of the 60 test problems, we have also computed the RTM upper bounds using the approach discussed in chapter 5. Table 6.6 gives the RTM bounds normalized to a percentage figure to allow the comparison with *selection %* of SFW and DF. Figure 6.2 sketches samples of the RTM results. The computed RTM bound quantifies the maximum testing effort required to test for all potentially affected by the modification. The computed bounds vary with the test problem and the reachability that $S_{mod}$ exhibits with other segments within the same module (e.g., see Figure 6.2).

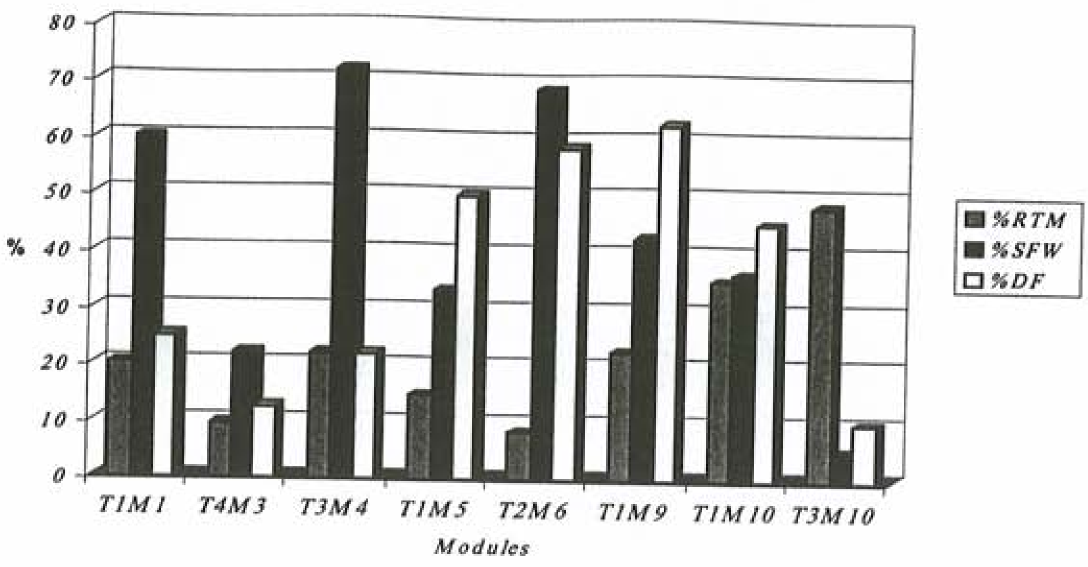

*Figure 6.2. RTM samples from m1-m10*

# CHAPTER 7

# CONCLUSIONS AND FURTHER WORK

In this thesis, we have addressed two major selective regression testing problems: (1) the test selection problem, the problem of selecting tests from an existing test suite and (2) the coverage identification problem, the problem of determining where additional tests may be required. We have addressed the former by proposing three reduction-based selective regression-testing methods. We have suggested an approach to the former and the latter by suggesting bounds on the selected retests based on McCabe cyclomatic complexity.

The proposed three reduction-based selective regression testing methods are referred to as MBR1, MBR2, and PR. MBR1 improves the RED algorithm by using reachability information to only account for the modification and its impacts in the reduction test selection process. MBR2 improves MBR1 to only select test cases that execute the modification. PR uses incremental slicing-based analysis to only select modification-revealing tests. These algorthims reduce the number of selected test cases and eliminate redundant tests.

We have suggested two McCabe-based regression test selection metrics to address the coverage identification problem. These are Reachability regression Test selection McCabe-based metric, and dataflow Slices regression Test

McCabe-based metric. The former, RTM, is an upper bound metric that derives its measures from reachability information and provides an upper indication of paths that must be tested for being potentially affected by the modification, and by implication, an upper bound of tests to rerun for exercising these paths at least once. The latter, STM, a data flow McCabe-based variable dependent metric, derives its measures from information related to slices affected by the modification and computes two bounds: an upper and a lower bound of the number of retests to rerun to test the affected definition-use pairs.

We have empirically compared our reduction-based methods to other three-minimization approaches. These are SA, RED, and SLI. Experimentation has shown that significant reduction in the test suite may be realized. The results show that PR has selected the least number of tests for almost all the test problems with full precision. MBR2 has improved MBR1 by further reduction of 47% in the number of selected tests for the 60 test problems. MBR1 has slightly improved RED for programs with segments exhibiting low reachability from the modified one.

We have applied the suggested McCabe-based metrics to DF and SFW. We have used the calculated RTM and STM bounds to monitor the test selection adequacy of these techniques.

We indicate that our proposed reduction-based methods could be applied once the impact of change is determined and a test-requirement association is established. This makes our methods flexible and easier to adapt and extend to address the revalidation of various programming paradigms. For instance, the two versions of Modification-Based Reduction could be easily extended to address the revalidation of object-oriented software on the class integration level using a similar approach to that described in (Hsia et al., 1997).

Also, we note that most of the selective regression testing techniques are frequently unaware of how the original test suite was designed (Rosenblum and Weyuker, 1997). Most of the regression testing strategies that have been described in the literature are independent of any coverage criterion that may have been used to create the original test suite. Selective regression testing based on the criterion used to generate the original test suite might be more favorable and help in monitoring the quality and effectiveness of regression testing through test-coverage adequacy analysis and metric collection. For instance, test suite development using McCabe cyclomatic complexity and regression testing based on the proposed McCabe-based regression test selection metrics illustrates a typical example of a recommended strategy that extend the approach used in test development to address the retesting problem in a more deterministic context through facilitating test-coverage adequacy analysis and metric collection.

We conclude by noting that the costs of testing and maintenance dominate the cost of the evolved software and are motivated to a large degree, by a desire to ensure software quality. The long-term quality of the evolved software is dependent on the effectiveness of maintenance and retesting. To retest for quality, we recommend studying various factors that might affect the effectiveness of these regression-testing techniques such as the fault-detection effectiveness, test-coverage adequacy, and safety measures.

Further work would involve: (1) automating the generation of the preprocessing requirements of the different algorithms; (2) extending McCabe-based regression test selection metrics to address various selective regression testing criteria; (3) designing and implementing realistic scenarios for testing (to construct *TS*), modifying, regression testing, and executing the test codes to quantitatively evaluate and compare the fault-detection capabilities of the proposed reduction-based algorithms; (4) to analytically and empirically investigate factors in software and test design that affect the regression testability of software; and (5) adapting the proposed reduction-based selective regression testing methods for addressing the revalidation of object-oriented software .

## REFERENCES

Agrawal, H., Horgan, J.R., Krauser, E.W., 1993. Incremental regression testing. In: *Proc. Conference on Software Maintenance*, 348-357.

Baradhi, G., Mansour, N., 1997. A comparative study of five regression testing algorithms. In: *Proc. Australian Software Engineering Conference*, 174-182.

Beizer, B., 1989. *Software Testing Techniques*. Van Nostrand Reinhold.

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

Bennet, K.H., 1991. The software maintenance of large software systems : management, methods and tools. *Reliability Engineering and Systems Safety*, 32,135-154.

Binkley, D., 1997. Semantics guided regression test cost reduction. *IEEE Trans. Software Engineering*, 23(8), 498-516.

Chen, Y., Rosenblum, D.S., Vo, K., 1994. TestTube: a system for selective regression testing. In: *Proc. Int. Conference on Software Engineering*, 211-220.

Fischer, K., Raji, F., Chruscicki, A., 1981. A methodology for retesting modified software. In: *Proc. National Telecommunications Conf.*, November, B6.3.1-B6.3.6.

Gallagher, K.B., Lyle, J.R., 1988. Using program decomposition to guide modifications. In: *Proc. Conference on Software Maintenance*, 265-269.

Graves, T.L., Harrold, M.J., Kim, J.M., Porter, A., Rothermel, G., 1998. An empirical study of regression test selection techniques. In: *Proc. Int. Conf. on Software Engineering*, 188-197.

Gupta, R., Harrold, M.J., Soffa M.L., 1996. Program slicing-based regression testing techniques. *Software Testing, Verification and Reliability*, 6(2), 83-111.

Harrold, M.J., Soffa, M.L., 1988. An incremental approach to unit testing during maintenance. In: *Proc. Conference on Software Maintenance*, 362-367.

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

Hartmann, J., Robson, D.J., 1988. Approaches to regression testing. In: *Proc. Conference on Software Maintenance*, 368-372.

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

Hsia, P., Li, X., Kung, D.C., Hsu, C-T, Li, L., Toyoshima, Y., Chen, C., 1997. A technique for the selective revalidation of OO software. *Journal of Software Maintenance*, 9, 217-233.

Leung, H.K.N., White, L., 1989. Insights into regression testing. In: *Proc. Conference on Software Maintenance*, 60-69.

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

Leung, H.K.N., White, L., 1990b. Insights into testing and regression testing global variables. *Journal of Software Maintenance*, Vol. 2, 209-221.

Leung, H.K.N., White, L., 1991. A cost model to compare regression test strategies. In: *Proc. Conference on Software Maintenance,* 201-208.

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

McCabe, T.,1976. A complexity measure. *IEEE Trans. on Software Engineering*, Vol. SE2 (3), 308-319.

Mansour, N., El-Fakih, K., 1999. Simulated annealing and genetic algorithms for optimal regression testing. *Journal of Software Maintenance*, Vol. 11, 19-34.

von Mayrhauser, A., Zhang, N., 1999. Automated regression testing using DBT and Sleuth . *Journal of Software Maintenance*, Vol. 11, 93-116.

Rosenblum, D.S., Weyuker, E., 1997. Using coverage information to predict the cost-effectiveness of regression testing Strategies. *IEEE Trans. Software Engineering*, 23(3), 146-156.

Rothermel, G., Harrold, M.J., 1996. Analyzing regression test selection techniques. *IEEE Trans. Software Engineering,* 22(8), August, 529-551.

Rothermel, G., Harrold, M.J., 1997. A safe, efficient regression test selection technique. *ACM Trans. Software Engineering and Methodology*, 6(2), 173-210.

Rothermel, G., Harrold, M.J., 1998. Empirical studies of a safe regression test selection technique. *IEEE Trans. Software Engineering*, 24(6), 401-419.

Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C., 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: *Proc. Int. Conf. on Software Maintenance*, 34-43.

Vokolos, F.I., Frankl, P.G., 1997. Pythia: A regression test selection tool based on textual differencing. In: *Proc. 3rd Int. Conf. on Reliability, Quality and Safety of Software-Intensive Systems* (ENCRESS'97), 3-21.

Vokolos, F.I., Frankl, P.G., 1998. Empirical evaluation of the textual differencing regression testing technique. In: *Proc. Int. Conf. on Software Maintenance*, 44-53.

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

Wong, W.E., Horgan, J.R., London, S., Mathur, A.P., 1998. Effect of test set minimization on fault detection effectiveness. *Software Practice and Experience*, 28(4), pp. 347-369.

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