A Software Tool For Regression Testing

By

Nidal H. Araby

February 1997
A Software Tool for Regression Testing

by

Nidal H. Araby

B.S., Beirut University College, 1994

Project

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

February 1997

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

Dr. Issam Moghrabi
Assistant Professor of Computer Science
Lebanese American University
Abstract

We have developed a software tool that draws flow graphs for C programs and displays testing coverage information in a graphical and user friendly way. The tool helps the software engineer/maintainer to get a feel for the testing process and tables generated by any other testing tool. It also provides a framework for experimenting with regression testing algorithms. Within this framework, we have implemented a reduction based regression testing algorithm. This algorithm assumes that the testing criteria are given by a set of requirements with associated subsets of test cases and then tries to satisfy these requirements with a minimum of test cases. We have used the tool to compare a number of regression testing algorithms. The comparative study uses a variety of small-size and medium-size modules and is based on quantitative and qualitative criteria. The comparison results show that the six algorithms are suitable for different requirements of regression testing. For medium-size modules, the adapted firewall algorithm is the slowest. The genetic and simulated annealing algorithms produce the least number of retests, followed by reduction, then incremental, slicing, and adapted firewall algorithms.
ACKNOWLEDGEMENTS

I would like to express gratitude to Professor Nashat Mansour, my advisor. I am indebted to him not only for his valuable and inspiring comments, tireless effort, support and encouragement, but also for providing such an intellectually rich research environment. I am also grateful to Dr. Issam Moghrabi, my reader, for his help, interest and comments.

I wish to thank Dr. A. Kabbani, chairperson of the Natural Science Division, for his understanding and sustenance.

Also, I would like to express my thanks to Ghinwa Baradhi for supplying the slicing, incremental and firewall code and to Khaled Fakih for supplying the genetic code.

Most of all, I am very grateful to my family for their devotion to education as a priority in life. To my father, Hasan Araby, for his love and financial support. To my mother, Nahla Araby, for guiding me. To my two sisters, Rabia and Asma Araby for their unlimited love and support. To my fiance Nivine Jundi for her understanding and dedication.
# Contents

List of Figures

List of Tables

Acknowledgments

Chapter

1. **Introduction**  

2. **Background**  
2.1. Slicing algorithm  
   2.1.1. Program slice  
   2.1.2. Definition-use pairs  
   2.1.3. Slicing algorithms  
      2.1.3.1. Backward walk algorithm  
      2.1.3.2. Forward walk algorithm  
2.2. Incremental algorithm  
   2.2.1. Incremental regression techniques  
      2.2.1.1. The execution slice technique  
      2.2.1.2. The dynamic slice technique  
      2.2.1.3. The relevant slice technique  
2.3. Firewall Concept  
   2.3.1. Adapted firewall algorithm  
2.4. Genetic algorithm  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Background</td>
<td>8</td>
</tr>
<tr>
<td>Slicing algorithm</td>
<td>8</td>
</tr>
<tr>
<td>Program slice</td>
<td>8</td>
</tr>
<tr>
<td>Definition-use pairs</td>
<td>9</td>
</tr>
<tr>
<td>Slicing algorithms</td>
<td>10</td>
</tr>
<tr>
<td>Backward walk algorithm</td>
<td>10</td>
</tr>
<tr>
<td>Forward walk algorithm</td>
<td>11</td>
</tr>
<tr>
<td>Incremental algorithm</td>
<td>12</td>
</tr>
<tr>
<td>Incremental regression techniques</td>
<td>13</td>
</tr>
<tr>
<td>The execution slice technique</td>
<td>13</td>
</tr>
<tr>
<td>The dynamic slice technique</td>
<td>14</td>
</tr>
<tr>
<td>The relevant slice technique</td>
<td>15</td>
</tr>
<tr>
<td>Firewall Concept</td>
<td>16</td>
</tr>
<tr>
<td>Adapted firewall algorithm</td>
<td>16</td>
</tr>
<tr>
<td>Genetic algorithm</td>
<td>18</td>
</tr>
</tbody>
</table>
2.5. Simulated annealing algorithm

2.6. Comparison criteria

2.6.1. Quantitative criteria

2.6.1.1. Efficiency

2.6.1.2. Number of test cases for regression testing

2.6.1.3. Precision

2.6.1.4. Inclusiveness

2.6.2. Qualitative criteria

2.6.2.1. User’s parameter setting

2.6.2.2. Type of maintenance

2.6.2.3. Type of testing

2.6.2.4. Level of testing

2.6.2.5. Type of approach

2.6.2.6. Global Variables

3. Implemented reduction algorithm

3.1. Algorithm description

3.2. Examples

3.3. Experimental Examples

3.4. High level design of the reduction algorithm’s code

4. Tool for flow graph and test coverage display

4.1. Tool design

4.1.1. Design of input and control

4.1.2. Design of online dialogue

4.1.3. Design of output
4.1.4. Structure charts 44
4.2. Implementation techniques 45
4.3. Examples 48

5. **Experimental results and comparison** 56

5.1. Experimental design 56
5.2. Quantitative criteria results 58
  5.2.1. Efficiency and number of test cases to rerun 64
  5.2.2. Precision and inclusiveness 66
5.3. Qualitative criteria results 70
  5.3.1. User's parameter setting 70
  5.3.2. Type of maintenance 71
  5.3.3. Type of testing 72
  5.3.4. Level of testing 72
  5.3.5. Type of approach 73
  5.3.6. Global Variables 74
5.4. Summary of results 75

6. **Conclusions and further work** 77

**Appendix A:** Low level design of the reduction algorithm A-1

**Appendix B:** Further examples on the application of the reduction algorithm B-1

**Appendix C:** Low level design of the flow graph and test coverage display tool C-1

**Appendix D:** Further examples on the application of the flow graph and test coverage display tool D-1

**References** R-1
List of Figures

Figure 2.1. Outline of the backward algorithm 9
Figure 2.2. Outline of the forward algorithm 12
Figure 2.3. Dynamic slice algorithm 14
Figure 2.4. An algorithm to compute the potential dependencies of a variable, var, at a location, loc, in a given execution history 15
Figure 2.5. "Firewall" concept algorithm at the unit level 17
Figure 2.6. Hybrid genetic algorithm for optimal retesting 18
Figure 2.7. Simulated annealing algorithm for optimal retesting 20
Figure 2.8. AVG program example 22

Figure 3.1. Heuristic algorithm reduce testsuite 29
Figure 3.2. Structure chart for the implemented reduction algorithm 39
Figure 3.3. The sequence of calling menus in the reduction algorithm 39

Figure 4.1. The menu tree of FGTCD (Flow graph and test coverage display) 43
Figure 4.2. The structure chart of FGTCD 44
Figure 4.3. The structure chart for the module buildlist 45
Figure 4.4. The pseudo code of the algorithm to draw the connectivity 47
Figure 4.5. Example of a connectivity graph generated by the tool 48

Figure 5.1. The # of selected test cases to rerun for the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m1_m5) 60
Figure 5.2. The # of selected test cases to rerun for the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m6_m10) 61

Figure 5.3. The # of selected test cases to rerun for the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m11_m15) 61

Figure 5.4. The # of selected test cases to rerun for the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m16_m20) 62

Figure 5.5. The running times in seconds of the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m1_m5) 62

Figure 5.6. The running times in seconds of the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m6_m10) 63

Figure 5.7. The running time in seconds of the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m11_m15) 63

Figure 5.8. The running time in seconds of the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m16_m20) 64

Figure 5.9. The precision of reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for modules presented in tables 5.6 to 5.11. 68

Figure 5.10. The inclusiveness of reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for modules presented in tables 5.6 to 5.11. 69
List of Tables

Table 2.1. Test history information for the AVG program 23

Table 3.1. Test suite TS consists of test cases $t_i$, testing requirements $REQ_i$, and the associated testing sets are $T_i$ 33

Table 3.2. Test suite TS consists of test cases $t_i$, testing requirements $REQ_i$, and the associated testing sets are $T_i$ for the worst case example 34

Table 3.3. Counts associated with each of the test cases in the first selection attempt 35

Table 3.4. Associated counts for test cases included in the tie for cardinality sizes (3, 4, 5, 6, 7, 8, 9) 35

Table 3.5. Associated counts for test cases included in the tie for cardinality size 10 36

Table 3.6. Map legend for table 3.7. 37

Table 3.7. Experimental examples of reduce test suite algorithm 38

Table 4.1. The connectivity formatted file (CFF) format 41

Table 4.2. The variable information file (VIF) format 42

Table 4.3. The test traversal file (TTF) format 42

Table 4.4. The textual information of the variables as shown on the screen 44

Table 4.5. The example connectivity formatted input file 49

Table 4.6. Table of coordinates generated for example 1 using the algorithm 51

Table 4.7. The second example for the connectivity formatted file 52

Table 4.8. Table for the X and Y coordinates for the example 2 file generated by the algorithm 52
Table 4.9. Example 3 for the connectivity formatted file

Table 4.10. Table for the X and Y coordinates for the example 3 file generated by the algorithm

Table 5.1. The example test traversal file

Table 5.2. The generated requirements file for example given in table 5.1.

Table 5.3. Parameters used in tables 5.4 and 5.5

Table 5.4. The running times in seconds and the number of test cases to rerun of reduction, slicing and incremental algorithms for modules m1 to m20

Table 5.5. The running times in seconds and the number of test cases to rerun of adapted firewall, genetic and simulated annealing algorithms for modules m1 to m20

Table 5.5. Parameters used in Tables 5.6, 5.7, 5.8, 5.9, 5.10, and 5.11

Table 5.6. Precision and inclusiveness results for the slicing approach

Table 5.7. Precision and inclusiveness results for the incremental algorithm

Table 5.8. Precision and inclusiveness results for the adapted firewall algorithm

Table 5.9. Precision and inclusiveness results for the genetic algorithm

Table 5.10. Precision and inclusiveness results for simulated annealing algorithm

Table 5.11. Precision and inclusiveness results for reduction algorithm
Chapter 1

Introduction

Testing is done to provide confidence about the correctness of a software system. However, there are changes that happen to the software after its deployment in the market. This action is described as software maintenance, which refers to changing programs as a result of errors or due to alterations in the user requirements [Hartman and Robson 89]. During such modifications new errors may be introduced, causing unintended adverse side effects in the software. Here, another phase of testing is needed. Estimates have shown that software maintenance costs accounts for as much as two thirds of the overall cost of the software production. One costly but so important activity in this phase is regression testing, the act of ensuring that modifications are correct and that no adverse effect has been introduced by these modifications [Harrold and Rothermel 93].

Modifications to a software system can be classified as one of four types:

- Corrective: It is the modifications done to a system to correct an error in response to an error fault report.
- Adaptive: It is the modifications to a system because of a change to the external environment e.g. migration to another platform (e.g. different operating system)
• Perfective: It is the modifications to a system to enhance its functionality (Speed, accuracy, etc.)

• Preventive: It is the modifications to a system to ease future maintenance e.g. a routine that has become excessively big and complicated may be redesigned to make it easier to understand and maintain. [Hartman and Robson 90]

Note that perfective maintenance call for new functional tests. Corrective, adaptive and preventive maintenance implies no changes to the requirements so no new functional test cases are needed. However, this does not mean that we are not in need for test cases that cover the new paths added to the software. Also, note that the causes for corrective and adaptive maintenance are merely external in response to error report or a change in platform as we have described earlier whereas perfective and preventive maintenance are much a management decision. Whatever is the need for the change and whatever is the category of the change, regression testing is a necessity to retest and revalidate the modified software.

Many approaches can be used to provide confidence about the new software system. One approach is to leave the user experiment with the system and report back to the developer any error that occurred and the developer correct these errors and returns the new version to the end user and so on. This approach is not logical in the sense that the user has always to deal with a software that is full of bugs which deprives him from benefiting from the ultimate objectives for buying a software system: Speed in processing, cost reduction, etc. A second approach is to retest the whole program from scratch. This approach provide the highest level of confidence in the software system
but what about the cost and time spent in executing all test cases found in the test suite. In addition, usually during regression testing we have a time constraint that we must meet. Therefore, this approach is not feasible. A third approach is to execute random test cases that can ensure confidence in the overall functionality of the software system satisfying ourselves that the system will be used under normal conditions. This approach will not work well for a critical software system that is controlling a nuclear plant where a tiny fault may cost the lives of humans. Therefore, we can not afford such a method in some systems. There is a solution for this problem? Research today is oriented towards what is called selective retest[Harrold and Rothermel 93, Harrold et al. 93, Hartman and Robson 90].

A selective approach to regression testing attempts to reuse tests from an existing test suite to test a modified software. These selected tests are those promising that will exhibit a different behavior on the modified version of the software and may potentially reveal errors in modifications done [Harrold and Rothermel 93]. Therefore, we need algorithms that will do selective retest at the minimum cost, with the maximum efficiency and precision and if possible automate the whole work that must be done to do it fast.

Therefore, the objectives for this work are:

- Reduction of time and cost associated with regression testing.
- Automation as much as possible of the maintenance phase.
- Reducing errors associated with modifications if possible.
Regression testing differs from the ordinary testing phase in the following points:

- During the testing phase we create the test suite while in regression testing we have a developed test suite that we must choose from for retesting the modified software.
- Testing is done to the whole system while regression testing is done for a section of code usually.
- Testing is done before the software is released in the market after the implementation while regression testing is done after the release in the market and after a modification to the software. [Hartman and Robson 90]

Two types of regression testing have been identified: Progressive and corrective regression testing [Leung and White 89]. The former involves testing major changes to the specifications. The latter is performed on a specifications that essentially remains unchanged, so that only minor alterations that do not affect the overall program structure, require revalidation [Hartman and Robson 89].

A number of approaches for selective regression testing have been proposed. Such approaches are classified as data flow approaches, slicing approaches, integration approaches, or minimization approaches. Data flow approaches use the data dependencies in a program to guide the selection of the test cases that must be rerun after determining the relationships between definitions of variables and uses of the same
variable [Harrold et al. 93]. Incremental approaches use information saved from previous testing sessions to determine the effect of a program modifications. They update the test case requirements to reflect the changed program and identify a subset of the test suite for the regression testing [Agrawal et al. 93]. Slicing approaches decompose programs by analyzing data flow and control flow graphs. The decomposition yields a method for maintainers to use so that changes can be assured to be completely contained in the modules under consideration, and that there is no undetected “linkages” between the modified code and unmodified code [Gallager and Lyle 88]. Minimization approaches assume the goal of regression testing is to re-establish satisfaction of some structural coverage criterion and aim to identify a minimal set of test cases that must be rerun to meet that criterion [Fischer and Chruscicki 81, Harrold and Rothermel 94, Mansour and Goel 94, Fakih and Mansour 96]. Another approach which is described in this thesis is the reduction approach suggested by Harrold, Gupta, and Soffa [Harrold et al. 93].

Now it is the time to give selective regression testing a formal definition. One of the definitions given by [Harrold and Rothermel 93] is:

**Problem**: Given program $P$, its modified version $P'$, and test set $T$ used previously to test $P$. Find a way, making use of $T$, to gain sufficient confidence in the correctness of $P'$.

Typical solutions to the problem follow the general outline proposed below:
- Identify the modifications made to P, and obtain a mapping between code segments in P and P'.
- Using the results of step 1, select \( T' \subseteq T \), a set of tests that may reveal modification related errors in P'.
- Run \( T' \) on P', establishing P's correctness with respect to T'.
- If necessary, create new tests for P'. These may include new functional tests required by change in specifications, and/or new structural tests required by applicable coverage criteria.
- Create \( T'' \), a new test set/history for P'. [Harrold and Rothermel 93].

Other definitions are provided in other papers. However, all definitions agree that we must select \( T' \) in such a way that it is most likely to reveal modifications related errors and to avoid as much as possible inclusion of obsolete and redundant test cases.

In this work, we have developed a software tool that provides a framework for regression testing and for comparing regression testing methods. Within this framework, we have implemented a reduction regression testing algorithm and we present a comparative study done to assess the applicability of regression testing methods in different situations. The assessment criteria are: (i) efficiency, (ii) number of test cases for regression testing selected, (iii) user's parameters settings, (iv) the extent to which global variables complicate the regression testing, (v) precision, (vi) inclusiveness, (vii) level of testing, (viii) type of testing, (ix) type of maintenance that is supports, (x) and type of the approach. It is worth noting that the tool is a part of a larger project undertaken at LAU, leading for the development of the ELISSAR tool.
This thesis is organized as follows. Chapter 2 contains background information about the previously implemented approaches and their comparison criteria. Chapter 3 describes the implemented reduction based regression testing algorithm and provides some examples. Chapter 4 presents the tool for flow graph and test coverage display. Chapter 5 gives the experimental results. Chapter 6 contains the conclusion.
Chapter 2

Background

This chapter describes some popular implemented algorithms presented in a study. The algorithms are described here because the work we have done is a continuation of these methods and the experiments done on them. Also, in this chapter the criteria on which the comparison of methods were done are explained.

2.1. Slicing Algorithm

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

2.1.1. Program slice

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

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

Figure 2.1. Outline of Backward algorithm.

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

2.1.3. Slicing algorithms

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

2.1.3.1. Backward walk algorithm

This algorithm is a backward walk through the program from the point of the edit that searches for definitions related to the changed statement. It identifies the definitions of a set of variables $U$ that reach a program point $s$. An outline of the algorithm is given in Figure 2.1.

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

2.1.3.2. Forward walk algorithm

This algorithm is a forward walk through the program from the point of the edit that detects the uses and the subsequent definitions and uses that are affected by a definition that is changed as a result of the program edit. Any definition-use pairs that depend on a changed predicate are identified. Also, the algorithm identifies definition-use pairs that are control dependent on an affected predicate, even though the value computed by the definition is unaffected. The algorithm is outlined in Figure 2.2. It identifies the uses of values, that are directly or indirectly affected by a change in either a value of $v$ at point $s$ or a change in a predicate. ValueUseTriples have the form of triples $(s, u, v)$ indicating that the value of variable $v$ at statement $s$, affected by the change, is used by statement $u$. The returned ValueUseTriples are used to compute the affected definition-use pairs. The algorithm inputs a set of pairs representing the definitions whose uses are to be found, along with a Boolean that indicates whether the change is only a predicate change. For each statement node $n$, two sets are maintained containing the definitions whose uses are to be found, since their values are affected by the edit. One set contains the values just before $n$ whose uses are to be found. The other set contains the values just after $n$ whose uses are to be found. A queue is maintained based on a depth first ordering of nodes in the control flow graph. The queue consists of those nodes that must be visited and examined for c-uses and p-uses of the definitions of variables in the set consisting of the values just before $n$, along with definitions in statements that are control dependent on a changed or affected predicate.
When the queue is empty, all ValueUseTriples that are affected by a change are identified and the algorithm terminates.

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

Figure 2.2. Outline of Forward algorithm.

### 2.2. Incremental algorithm

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

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

2.2.1. Incremental regression testing techniques

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

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

2.2.1.1. The execution slice technique

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

13
is used to solve this problem. For this algorithm, it has been assumed that the execution slices for the test cases are given.

2.2.1.2. The dynamic slice technique

<table>
<thead>
<tr>
<th>For each test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursively DO</td>
</tr>
<tr>
<td>For each statement $S$ in the test case</td>
</tr>
<tr>
<td>if a p-use is found</td>
</tr>
<tr>
<td>search for its successor(s) in the dynamic slice</td>
</tr>
<tr>
<td>if found</td>
</tr>
<tr>
<td>add $S$ to the dynamic slice</td>
</tr>
<tr>
<td>else if a definition is found</td>
</tr>
<tr>
<td>if the dynamic slice statements does not contain any of its successor</td>
</tr>
<tr>
<td>include $S$ in the dynamic slice</td>
</tr>
<tr>
<td>endfor;</td>
</tr>
<tr>
<td>endfor;</td>
</tr>
</tbody>
</table>

Figure 2.3. Dynamic Slice Algorithm.

This technique is based on observation 3. It suggests that not every statement that is executed under a test case has an effect on the program output for that test case. Dynamic program slices are computed to determine for each test case the statements that have an effect on the program output. A dynamic program slice is obtained by recursively traversing the data and control dependence edges in the dynamic dependence graph of the program for the given test case. All slices are computed during off-line processing. Then, after the program is modified, the new program is rerun on only those test cases whose dynamic program slices contain a modified statement. Figure 2.3 outlines the algorithm for computing the dynamic slices. However, this technique fails to identify test cases that if executed will affect the output if they are evaluated differently. The relevant slice technique will solve the problem.
2.2.1.3. The relevant slice technique

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

\[
\text{static-defs} = \text{static reaching definitions of var at loc;}
\]
\[
\text{dynamic-defs} = \text{the dynamic reaching definition of var at loc;}
\]
\[
\text{control-nodes} = \text{the closure of the static control dependencies}
\]
\[
\text{of the statements in static-defs;}
\]
\[
\text{initialize potential-deps to null;}
\]
\[
\text{mark all nodes in the dynamic dependence graph between loc and}
\]
\[
\text{dynamic-def as not visited;}
\]
\[
\text{for each node, n, in the dynamic dependence graph starting at}
\]
\[
\text{loc and going back up to dynamic-def do}
\]
\[
\text{if n is marked as visited then}
\]
\[
\text{continue; /*skip this node*/}
\]
\[
\text{mark n as visited;}
\]
\[
\text{if n belongs to control-nodes then}
\]
\[
\text{add n to potential-deps;}
\]
\[
\text{mark all the dynamic control dependencies of n}
\]
\[
\text{between n and dynamic-defs as visited;}
\]
\[
\text{endif;}
\]
\[
\text{endfor;}
\]
\[
\text{return potential-deps;}
\]

Figure 2.4. An algorithm to compute the potential dependencies of a variable, var, at a location, loc, in a given execution history

Figure 2.4 depicts the algorithm for computing the potential dependencies of a variable, \textit{var}, at a location, \textit{loc}, in a given execution history. The relevant slices are composed of
the union set of the potential dependencies and the dynamic slice for each test case in
the test suite.

2.3. Firewall Concept

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

2.3.1. Adapted firewall algorithm

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

The adapted firewall is based on the control-flow graph where each node is a segment
and the arcs represent connectivity between segments. Associated with the control-flow
are a connectivity matrix and a reachability matrix that depict the connectivity of
segments to each other and the reachability of segments from each other respectively.
Additionally, a test case cross reference matrix is associated with the control-flow
graph that identifies segments executed by test cases during the testing phase of
software development.
The adapted firewall algorithm proposed encloses the set of segments that must be unit-tested after a modification is made to the program. A "firewall" is built around the modified segments and some related segments. Only those segments within the "firewall" need to be re-unit and regression tested.

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

- the direct ascendants of definitions that are affected by a program edit,
- and the direct descendants where no uses are affected by the program edit.

The algorithm to construct the "firewall" at the unit level is described in Figure 2.5.

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

Figure 2.5. "Firewall" Concept Algorithm at the Unit Level.
2.4. Genetic Algorithm

Genetic algorithms are based on the mechanics of natural evolution. They mimic natural populations reproduction and selection operations to achieve efficient and robust optimization. The optimal retesting problem consists of finding values for \((X_1, X_2, \ldots, X_N)\) that minimize the cost function

\[
Z = c_1X_1 + c_2X_2 + \ldots + c_NX_N
\] (1)

Subject to the constraints

\[
\sum_{j=1}^{N} a_{ij}X_j \geq b_i ; \ i = 1, \ldots, M
\]

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

Random generation of initial population, Size POP;
Evaluates fitness of individuals;

repeat

Rank individuals and allocate reproduction trials;
for \((i=1\) to \(POP, \) step 2) do

Randomly select 2 parents from list of reproduction trials;

Apply crossover and mutation;

endfor

Evaluate fitness of offsprings;
Check feasibility of individuals;
Do feasibilization, penalization and hill-climbing;

until (termination criterion is satisfied)

Solution = Fittest.

Figure 2.6. Hybrid Genetic Algorithm for Optimal Retesting

An outline of a hybrid genetic algorithm for minimizing that cost function \(Z\), is described in Figure 2.6.
HGAs algorithm is an array of POP individuals. An individual in the population is encoded as an n-element vector \([X_1, X_2, X_3, X_n]\) that corresponds to a candidate solution for the optimal retesting problem. An element (gene) \(X_j = 1\) (or 0) indicates the inclusion (or exclusion) of test case \(j\) from the selected subset of retests. The initial population of individuals is randomly generated. The fitness of an individual \(Z\) is evaluated by adding the genes of an individual. Henceforth, the optimal subset of retests corresponds to the minimum fitness \(Z\) of all feasible individuals.

2.5. Simulated annealing algorithm

The simulated annealing algorithm simulates the natural annealing phenomenon by a search (perturbations) process in the solution space (energy landscape) optimizing some cost function (energy) [Mansour and Goel 1994]. It starts with some initial solution at a high (artificial) temperature and reduces the temperature gradually to a freezing point. At each temperature, regions in the solution space are searched by the Metropolis algorithm. The algorithm is given in Figure 2.7, where a candidate solution is represented by the configuration \(X_{soln} = (X_1, X_2, \ldots, X_n)\). The energy is given by the cost function \(Z\) in equation 1 described above.

2.6. Comparison Criteria

In this section we present the criteria used for evaluating and comparing regression testing approaches. The criteria are divided into two categories: quantitative and qualitative. The quantitative criteria are efficiency, quality, precision, and inclusiveness. The qualitative criteria are user’s parameter setting, global variables, type of maintenance, type of testing, level of testing, and type of approach.
Initial configuration = random $X_{soln}$;
Determine initial temperature $T(0)$;
Determine freezing temperature $T_f$;
while ($T(i) > T_f \text{ and not converged}$) do
    repeat
        /*Metropolis Algorithm*/
        perturb ($X_{soln}$);
        if (perturbed $X_{soln}$ is feasible) then
            accept_or_not();
        else
            reject_perturbation();
        until equilibrium
    save_bestsofar();
    check_convergence();
    $T(i+1) = \alpha T(i);$ /*cooling schedule*/
endwhile;
procedure accept_or_not()
    if ($\Delta Z \leq 0$) then
        update ($X_{soln}$); /*accept perturbation*/
    else
        if ($\text{random}(0, 1) < e^{\Delta Z/T(i)}$) then
            update ($X_{soln}$)
        else
            reject_perturbation();
        endif;
endif;

Figure 2.7. Simulated Annealing Algorithm for Optimal Retesting.

2.6.1. Quantitative criteria

2.6.1.1. Efficiency

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

2.6.1.2. Number of test cases for regression testing

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

2.6.1.3. Precision

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

(i) modification-revealing tests

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

(ii) modification-traversing tests

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

However, although modification-revealing tests are necessarily modification-traversing, not all modification-traversing tests are modification-revealing. For example, consider the program example AVG in Figure 2.8 with its history information in Table 2.1. Suppose statement $S1$ count = 0, is modified. In this case, test $T2$ is modification-traversing because it executes the modified version of $S1$ in the new version of the procedure. However, test $T2$ is not modification-revealing: it traverses no statement
that uses the value computed in S1 and thus cannot produce different output in the new version of AVG [Harrold and Soffa 1994].

```
S1:    count = 0
S2:    fread(fileptr, n)
S3:    while (not EOF) do
S4:        if (n < 0)
S5:            return(error)
S6:        else
S7:            numarray[count] = 0
S8:            count++
S9:        endif
S10:    endwhile
S11:    avg = calcavg(numarray, count)
S12:    return(avg)
```

Figure 2.8. AVG Program Example.

Precision measures the extent to which a regression testing method omits tests that are non-modification-revealing [Harrold and Rothermel 1995]. It is the ability of a method to avoid choosing these tests that will not cause the modified program to produce different output. A test would produce different output for the modified program, if it executes modified code. We can compare the precision of methods by showing how much they can select the modification-revealing tests only. Suppose the test suite $T$ contains $n$ non-modification-revealing tests, and a regression testing approach, $S$, selects $m$ of these tests, then the precision of $S$ relative to the original program, the modified program, and the test suite $T$, is the percentage calculated by the expression $((m/n) \times 100)$. For example, if $T$ contains 50 tests, 44 of which are non-modification-revealing with respect to the modified program, and $S$ omits 33 of these 44 tests, then $S$ is 75% precise relative to the original program, the modified program, and the test suite $T$. 
<table>
<thead>
<tr>
<th>test number</th>
<th>input</th>
<th>output</th>
<th>execution history</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>empty file</td>
<td>0</td>
<td>S1,S2,S3,S9,S10</td>
</tr>
<tr>
<td>T2</td>
<td>-1</td>
<td>error</td>
<td>S1,S2,S3,S4,S5</td>
</tr>
<tr>
<td>T3</td>
<td>1 2 3</td>
<td>2</td>
<td>S1,S2,S3,S4,S6,S7,S8,S3,....,S9,S10</td>
</tr>
</tbody>
</table>

Table 2.1. Test history information for AVG program

2.6.1.4. Inclusiveness

Inclusiveness measures the extent to which a regression testing method chooses tests that will cause the modified program to produce different output. It is the ability of a method to specifically select all tests that cover affected statements. We can compare the inclusiveness of a method by showing how much it can account for the effects of new and/or deleted statements. Also, conclusions about inclusiveness can be drawn by analyzing the test suite available and checking for a modification-revealing test that is missed out by the regression testing method. Suppose the test suite $T$ contains $n$ modification-revealing tests, and a regression testing approach, $S$, selects $m$ of these tests, then the inclusiveness of $S$ relative to the original program, the modified program, and the test suite $T$, is the percentage calculated by the expression $\left(\frac{m}{n}\right) \times 100$. For example, if $T$ contains 50 tests, 8 of which are modification-revealing with respect to the modified program, and $S$ selects only 2 of the 8 modification-revealing tests, then $S$ is 25% inclusive relative to the original program, the modified program, and the test suite $T$.

2.6.2. Qualitative criteria
2.6.2.1. User’s parameter setting

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

2.6.2.2. Type of maintenance

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

2.6.2.3. Type of testing

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

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

2.6.2.5. Type of approach

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

2.6.2.6. Global variables

The ability to regression test global variables should be considered. It is important not to overlook testing these variables. Using global variables affects not only the understandability, readability and maintainability of the software, but also its testability. Global variables create data-flow dependencies between modules which are not directly callable and may be well separated in the call graph [Leung and White 1990b].
Chapter 3

Implemented Reduction Algorithm

In this chapter, we describe the reduction based algorithm proposed to solve the test selection and test update problem simultaneously. [Harrold et al. 93]. Also, this chapter explains the operation of the algorithm through examples and presents its high level design. We would add here an introductory remark.

A reduction in the size of the test suite decreases the overhead of maintaining the test suite as well as the number of test cases that must be rerun. However, the management of test suite includes two major divisions: Adding new test cases, and eliminating unnecessary test cases. Eliminating unnecessary test cases includes two kinds: obsolete and redundant test cases. A change in a program causes a test suite to be obsolete by removing the reason for its inclusion in the test suite. Also, a test case is redundant if other test cases in the test suite provides the same coverage. So it is desirable to determine a minimum set of test cases that provides the same coverage of the changed or affected parts of the program. However, finding all the subsets of a set is an NP-Complete and there is no polynomial deterministic algorithm that can solve this problem in a reasonable time [Harrold et al. 93].

3.1. Algorithm Description

Harrold, Gupta, and Soffa's heuristic algorithm [Harrold et al. 93] can be used to construct a representative set of test cases from the available test suite which can still
provides the same coverage. The heuristic presented requires an association between the test cases and the testing requirements of the program, but is independent of the testing selection criterion and can applied if this association can be made.

For a particular program, a test selection criterion translates into a set of test case requirements whose satisfaction provides the desired measure of completeness with respect to that criterion. For each test requirement, an associated testing set consists of a subset of the test suite that satisfies the requirement. The test suite reduction technique uses both the test case requirements and their associated testing subsets. A program modification may cause a change in program's test case requirements: new requirements may be added or existing requirements may be deleted or modified. Although some test cases in the existing test suite may retest the modified software, the change in test case requirements may require new test cases and may also results in unnecessary test cases to be eliminated. Both regression testing approaches, Incremental and Retest all, benefit from the reduction technique presented here. An incremental approach to testing after program changes can use the reduction technique in two ways. First, it can use the technique to find a reduced set of test cases for retesting. With incremental testing, an analysis uses information saved from previous testing sessions to determine the effect of program modifications. The analysis updates the test case requirements to reflect the changed program and identifies a subset of the test suite for retesting. There may be redundancy in this subset since several test cases may test the same changed parts of the program. So it is desirable to eliminates these redundant test cases. The second way that an incremental approach can use the reduction technique is to reduce the size of the stored test suite. In the retest all
approach, the associated testing sets are not updated to reflect the removal of test case requirements and thus, obsolete test cases remain in the test suite. Additionally, there is no attempt to remove redundant test cases. The reduction technique can be used to eliminate both obsolete and redundant test cases from the test suite. A statement of the problem of selecting a representative set of test cases that provides the desired testing coverage of a program or part of the program is as follows:

Given: A test suite TS, a set of test case requirements \( r_1, r_2, r_3, \ldots, r_n \) that must be satisfied to provide the testing coverage of the program, and subsets of TS, \( T_1, T_2, T_3, \ldots, T_m \), 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 \( r_i \).

Problem: Find a representative set of test cases from TS that satisfies all of the \( r_i \)'s i.e. satisfy all the requirements.

The \( r_i \)'s can represent all of the program test case requirements or those requirements related to program modifications. A representative set of test cases that satisfies the \( r_i \)'s must contain at least one test case from each \( T_i \). Such a set is called the hitting set of the group of sets \( T_1, T_2, T_3, \ldots, T_n \). A maximum reduction is achieved by selecting the smallest representative set of test cases, an NP-complete problem. Therefore, since we are unaware of any approximate solution to the problem, the authors developed a heuristic to find a representative set that approximates the minimum cardinality hitting set [Harrold et al. 93].
Input: T_1, T_2, ..., T_n, associated testing subsets for each T_i, respectively, containing test cases from t_1, t_2, ..., t_n.
Output: RS, a representative set of T_1, T_2, ..., T_n.

Declare:
- MAX_CARD: integer
- CURR_CARD: integer
- LIST: list of T_i's
- NEXT_TEST: one of T_i's
- MARKED: array[1..n] of Boolean, initially false
- MAY_REDUCE: Boolean

MAX: return the maximum of a set of numbers
CARD(y) returns the cardinality of a set

/* step 1: initialization */
MAX_CARD := MAX(CARD(T_i)); /* get the maximum cardinality of the T_i's */
RS := \bigcup T_i; CURR_CARD := 1; /* take union of all the singleton elements of the T_i's */
For each T_i such that T_i \cap RS = \emptyset do marked[i] := true; /* mark all T_i's containing elements in the representative set */
CURR_CARD := 1; /* consider single elements sets first */

/* step 2: compute RS according to the heuristic rules for sets of higher cardinality */
Loop:
CURR_CARD := CURR_CARD + 1;
while there is a T_i such that (CARD(T_i) = CURR_CARD and not marked[i]) do
/* process all unmarked sets of current cardinality */
LIST := all T_i, where CARD(T_i) = CURR_CARD and not marked[i]
    in T_i of size CURR_CARD */
NEXT_TEST := select(T_i, LIST); /* get another T_i to include in RS */
RS := RS \cup NEXT_TEST
MAY_REDUCE := false;
for each T_i, where NEXT_TEST \cap T_i do
    MARKED[i] := true; /* mark T_i containing NEXT_TEST */
    IF CARD(T_i) = MAXCARD then MAY_REDUCE := true;
end for
IF MAY_REDUCE then /* try to reduce MAXCARD */
    MAXCARD := MAXCARD(T_i), for all i, where MARKED[i] = false
end while;
Until CURR_CARD = MAX_CARD;
END REDUCE TEST SUITE

Function: Select(T_i, LIST)
/* This function is used to select the next T_i to be included in RS */
declare COUNT: array[1..n];
begins
for each T_i in LIST do compute COUNT[i], the number of unmarked T_i's of cardinality SIZE containing T_i;
Construct TEST_LIST consisting of T_i's from list for which COUNT[i] is the maximum.
IF CARD(TEST_LIST) > 1 then return the test case in this TEST_LIST
else if SIZE = MAXCARD then return any test case in the TEST_LIST
else return select(SIZE, TEST_LIST);
end select.

Figure 3.1. Heuristic Algorithm REDUCETESTSUITE
The heuristic first includes all test cases that occur in a single element $T_i$'s in the representative set and marks all the $T_i$'s and mark all the $T_i$'s containing these test cases as being satisfied. Then all unmarked $T_i$ of cardinality two are considered. Repeatedly, the test case that occurs in the maximum number of $T_i$'s of cardinality two is chosen and added to the representative set. Again, all unmarked $T_i$'s that contains 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 begins with subsets of $T_i$'s that contains as few elements as possible to reduce the time of the process. At any given time, only those $T_i$'s from which no element has yet been chosen are considered. When examining the $T_i$'s of size $n$, there may be a tie because several test cases occur in the maximum number of $T_i$'s of that size. In this case, the heuristic examines all 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 can be made.

The reduction algorithm that simulates this idea is given in Figure 3.1. Its pseudo code implementation is given in appendix A of the paper.

In step 1, algorithm REDUCETESTSUITE initializes and preprocesses all of the $T_i$'s. First, the maximum cardinality, MAX_CARD, of the input sets is determined and the cardinality being processed, CUR_CARD, is initialized to 1. Next, REDUCETESTSUITE initially forms the representative set, RS, by taking the union of all the $T_i$'s that are single element set. All $T_i$'s containing these elements in the initial
RS are marked; marked sets are not reprocessed. Then in step 2,
REDUCETESTSUITE determines the remainder of RS. Unmarked T_i's with increasing
cardinality are examined. A list, LIST, is constructed that consists of those test cases in
the T_i's with cardinality equal to the current cardinality, CURR_CARD. A new test
case, NEXT_TEST, is chosen from the list and added to the RS. The function
selecttest, is used to determine NEXT_TEST, the next test case to be added to RS.
Selecttest recursively examines the T_i's of cardinality SIZE for test cases contained in
the greatest number of T_i's. The array count is used to store the number of T_i's of
cardinality SIZE containing t_j in the LIST. If several t_j's are found, the T_i's of the next
higher cardinality are examined to select a single test case from these test cases.
REDUCETESTSUITE repeats this process until either a single test case is found or
one is chosen at random. A random choice is made if no higher cardinality set can be
examined to break the tie. The Boolean variable MAY_REDUCE is set whenever one
of the sets with cardinality MAX_CARD is marked. Whenever MAY_REDUCE is set,
MAX_CARD is reinitialized to the highest remaining cardinality of the remaining sets.
Resetting the MAX_CARD is an optimization that may reduce the time to select the
next test case.

Harrold et al. have analyzed the worst case running of time of the algorithm
REDUCETESTSUITE to demonstrate its efficiency and therefore its suitability to
incorporate into a test suite management methodology. Let n denote the number of
associated testing sets T_i, nt denote the number of test cases t_i, and MAX_CARD, the
maximum cardinality of the groups of sets. REDUCETESTSUITE 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 representative 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 \( O(n \times \text{MAX\_CARD}) \) time since there are \( n \) sets and all elements are examined once. Selecting the next test case to be included in the representative set requires examining the counts associated with each test case once. This step takes at most \( O(nt \times \text{MAX\_CARD}) \) time. Selecting a test case and recomputing the counts is repeated at most \( n \) times since after selecting a test case at least one additional test case is covered by the representative set. Therefore, the overall run time of the algorithm REDUCETESTSUITE is \( O(n(n+t)\text{MAX\_CARD}) \). The experiments has showed that it executes much faster. Note that the algorithm would perform poorly if minimum overlap between the subsets will occur, which will force the algorithm to always go for higher cardinality to break several ties that will occurs. The algorithm presented in this section can be either integrated into a testing tool or can be a stand alone tool.

### 3.2. Examples

In this section, we present two examples. The first example would be a simple normal case example while the second example would be a worst case scenario. For further examples you can review appendix B. As for the first example, Table 3.1 specifies the input to the algorithm.

In Table 3.1, the test suite TS consists of test cases, \( t_i \), the test case requirements, REQ\(_i\), and the associated testing sets, \( T_i \). The heuristic first adds test case \( t_5 \) to the
representative set since $T_2$ is the only singleton in the subsets $T_i$. $REQ_1$ and $REQ_2$ are marked as being satisfied since $t_5$ is associated with each of them. Then, we consider unmarked $T_i$'s of cardinality two (i.e. $T_4$, $T_5$, and $T_6$). Each of these test cases $t_3$ and $t_4$ appears in one of the $T_i$'s. Since there is a tie between test cases $t_1$ and $t_6$ for the maximum, we continue processing with unmarked $T_i$'s of the next higher cardinality. Thus, $T_3$ and $T_7$ are considered next. We only use the test cases involved in the tie to compute the maximum for cardinality 3. Test case $t_1$ appears in $T_3$ while test case $t_6$ appears in neither of the $T_i$'s. Thus, test case $t_1$ is chosen and added to the representative set. $T_3$, $T_5$, and $T_6$ are marked since they contain $t_1$. Now the representative set contains $RS = \{t_5, t_1\}$ and the marked $T_i$'s are $T_1, T_2, T_3, T_5$ and $T_6$. And the process continues until all requirements are satisfied. For the test selection problem, only test cases $t_1, t_3$, and $t_5$ must be rerun to achieve the desired coverage of the program itself or its changes.

<table>
<thead>
<tr>
<th>Number</th>
<th>Requirements</th>
<th>Test Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>1</td>
<td>$REQ_1$</td>
<td>${t_1, t_3}$</td>
</tr>
<tr>
<td>2</td>
<td>$REQ_2$</td>
<td>${t_3}$</td>
</tr>
<tr>
<td>3</td>
<td>$REQ_3$</td>
<td>${t_1, t_3, t_5}$</td>
</tr>
<tr>
<td>4</td>
<td>$REQ_4$</td>
<td>${t_3, t_5}$</td>
</tr>
<tr>
<td>5</td>
<td>$REQ_5$</td>
<td>${t_1, t_4}$</td>
</tr>
<tr>
<td>6</td>
<td>$REQ_6$</td>
<td>${t_1, t_5}$</td>
</tr>
<tr>
<td>7</td>
<td>$REQ_7$</td>
<td>${t_3, t_4, t_7}$</td>
</tr>
<tr>
<td>8</td>
<td>$REQ_8$</td>
<td>${t_3, t_4, t_7}$</td>
</tr>
</tbody>
</table>

Table 3.1. Test suite TS consists of test cases $t_i$, testing requirements $REQ_i$, and the associated testing sets are $T_i$

In this paragraph, we demonstrate a worst case scenario that will illustrate the case of multiple ties. Table 3.2 will present the needed input for the next example. This example present a worst case scenario in the sense that we must always refer to subsets
of higher cardinality to solve the tie that will always occur when dealing with subsets of maximum cardinality 2. This worst case scenario can be described by a general formula given below.

Each subset \(i < n\), where \(n\) is the number of requirements, take the test cases in the subset to be two test cases \(\{t_i, t_{i+1}\}\). As for the for the final requirement which is requirement number \(n\) take the subset of test cases to be all test cases from 1 till \(n\) i.e. \(T_n=\{t_1, t_2, ..., t_n\}\). For our purpose of explanation, we will take the value of \(n\) to be 10.

Table 3.2 shows the required format.

<table>
<thead>
<tr>
<th>Number</th>
<th>Requirement</th>
<th>Test Subsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>REQ_1</td>
<td>({t_1, t_2})</td>
</tr>
<tr>
<td>2</td>
<td>REQ_2</td>
<td>({t_2, t_3})</td>
</tr>
<tr>
<td>3</td>
<td>REQ_3</td>
<td>({t_3, t_4})</td>
</tr>
<tr>
<td>4</td>
<td>REQ_4</td>
<td>({t_4, t_5})</td>
</tr>
<tr>
<td>5</td>
<td>REQ_5</td>
<td>({t_5, t_6})</td>
</tr>
<tr>
<td>6</td>
<td>REQ_6</td>
<td>({t_6, t_7})</td>
</tr>
<tr>
<td>7</td>
<td>REQ_7</td>
<td>({t_7, t_8})</td>
</tr>
<tr>
<td>8</td>
<td>REQ_8</td>
<td>({t_8, t_9})</td>
</tr>
<tr>
<td>9</td>
<td>REQ_9</td>
<td>({t_9, t_{10}})</td>
</tr>
<tr>
<td>10</td>
<td>REQ_{10}</td>
<td>({t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}})</td>
</tr>
</tbody>
</table>

Table 3.2. Test suite TS consists of test cases \(t_i\) testing requirements \(REQ_i\), and the associated testing sets are \(T_i\) for the worst case example.

Let us take our example forward and apply the algorithm on it. As a first step, we search for the maximum cardinality and we obtain it to be 10 in \(T_{10}\). Now the algorithm should tries to initialize the representative set with all the singleton sets found in the test suite. However, RS would result to be empty since there is no one singleton set in our example. Therefore, we continue to examine sets of cardinality 2. Subsets to consider all are subsets inclusive in the range \(T_1\) to \(T_9\) since all of them are subsets with cardinality 2. However, in this case there is a very little overlap between these subsets resulting in a count for test cases given in Table 3.3.
<table>
<thead>
<tr>
<th>Number</th>
<th>Test Case</th>
<th>Test Case Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t₁</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>t₂</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>t₃</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>t₄</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>t₅</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>t₆</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>t₇</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>t₈</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>t₉</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>t₁₀</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3. Counts associated with each of the test cases in the first selection attempt.

As it can be seen from the results given in the Table 3.3, there is a tie among the test cases ranging from t₂ to t₉, all having a count of 2. So the algorithm would go for cardinality 3 counting the number of occurrences of all these test cases in subsets of Tᵢ with cardinality 3. Since there are no Tᵢ with cardinality 3 the associated counts will be zeros as given in the Table 3.4.

<table>
<thead>
<tr>
<th>Number</th>
<th>Test Case</th>
<th>Test Case Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t₂</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>t₃</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>t₄</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>t₅</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>t₆</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>t₇</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>t₈</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>t₉</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4. Associated Counts for test cases included in the tie for cardinality sizes 3, 4, 5, 6, 7, 8, 9

So the algorithm would go for higher cardinality repeatedly until it reaches a cardinality of 10 computing each time the counts to be 0. When it reaches the cardinality 10, the counts associated with test cases is given in Table 3.5.
<table>
<thead>
<tr>
<th>Number</th>
<th>Test Case</th>
<th>Test Case Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t_2$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$t_3$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$t_4$</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>$t_5$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>$t_6$</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>$t_7$</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>$t_8$</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>$t_9$</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.5. Associated Counts for test cases included in tie for cardinality size 10

Since we have reached the final cardinality which is equal to the MAX_CARD and the associated counts still give a result of a tie, a test case is selected randomly. In our case, let us take the first test case which is $t_2$. After we have chosen $t_2$, the test subsets that are satisfied in the Table 3.5 are $T_1$, $T_2$ and $T_{10}$. Therefore RS contains only $t_2$. For the next time, we are still dealing with cardinality 2; however, with less number of test case requirements, ranging between $T_3$ to $T_9$. Here comes the usefulness of the Boolean variable MAY REDUCE to reassign the highest cardinality to 2. Therefore next time when we consider left $T_i$, we won’t go for a cardinality that is higher than 2. Therefore the algorithm will return to its optimal state selecting in turn $t_4$, $t_6$, $t_8$, and $t_{10}$ and we will have a representative test suite containing the following test cases $RS = \{t_2, t_4, t_6, t_8, t_{10}\}$.

More examples on the algorithm are provided in appendix D at the end of the paper.

Finally as we can see the algorithm has some optimization steps that helps it to adapt to new situations and produce results in a reasonable amount of time that does not surpasses seconds. Also the algorithm has a wide range of application, that is it can applied approximately along with any other testing criterion whether it is structural based or functional based as long as it can be translated into a set of requirements and associated testing subsets. Also, the algorithm provides a high percentage of reduction in the number of test cases in the test suite if it contains unnecessary test cases.
3.3. Experimental Examples

In this section we show some of the results of the algorithm that was tested on and its performance. First we have to note that the reduction algorithm was developed using Turbo C IDE (Integrated Development Environment) and the main data structures used were arrays, used for their known advantage of fast processing, and linked lists, used for their known storage capacity. Also the computer used for the testing purposes was 486 DX-33 MHz with 4 MB of RAM. The needed input files for the experiments were provided manually. The needed information for the experiments were (for each module/test):

1. The modified segments if there is any specified as a requirement.
2. A Table of the test cases and their segment coverage information.
3. Set of requirements to be satisfied.

Note that all experiments were done at the module level. The result of the algorithm is presented for various module sizes and with various number of segments in Table 3.7. Table 3.6 is a legend Table that explains the symbols used in Table 3.7.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE NAME</td>
<td>Specify the file name that contains the test requirements.</td>
</tr>
<tr>
<td>N</td>
<td>Specify the number of test cases</td>
</tr>
<tr>
<td>M</td>
<td>Specify the number of requirements processed</td>
</tr>
<tr>
<td>Reduction Percentage</td>
<td>Specify the percentage of obsolete test cases</td>
</tr>
<tr>
<td>Worst case</td>
<td>Specify whether a file is a worst case or not</td>
</tr>
<tr>
<td>Time</td>
<td>Specify the time taken in seconds for processing the file</td>
</tr>
</tbody>
</table>

Table 3.6. Map Legend for Table 3.7.
<table>
<thead>
<tr>
<th>File Name</th>
<th>N</th>
<th>M</th>
<th>Worst Case</th>
<th>Percentage Reduction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5W0.DAT</td>
<td>4</td>
<td>5</td>
<td>YES</td>
<td>50</td>
<td>0.0000001</td>
</tr>
<tr>
<td>F6N0.DAT</td>
<td>6</td>
<td>6</td>
<td>NO</td>
<td>20</td>
<td>0.0000001</td>
</tr>
<tr>
<td>F10N0.DAT</td>
<td>7</td>
<td>10</td>
<td>NO</td>
<td>50</td>
<td>0.0000001</td>
</tr>
<tr>
<td>F10W1.DAT</td>
<td>9</td>
<td>10</td>
<td>YES</td>
<td>50</td>
<td>0.0000001</td>
</tr>
<tr>
<td>F15W0.DAT</td>
<td>14</td>
<td>15</td>
<td>YES</td>
<td>50</td>
<td>0.0000001</td>
</tr>
<tr>
<td>F25W0.DAT</td>
<td>24</td>
<td>25</td>
<td>YES</td>
<td>50</td>
<td>0.0000001</td>
</tr>
<tr>
<td>F50W0.DAT</td>
<td>49</td>
<td>50</td>
<td>YES</td>
<td>50</td>
<td>0.2747250</td>
</tr>
<tr>
<td>F75W0.DAT</td>
<td>74</td>
<td>75</td>
<td>YES</td>
<td>50</td>
<td>0.8241760</td>
</tr>
<tr>
<td>F100W0.DAT</td>
<td>99</td>
<td>100</td>
<td>YES</td>
<td>50</td>
<td>1.8681320</td>
</tr>
<tr>
<td>F126W0.DAT</td>
<td>125</td>
<td>126</td>
<td>YES</td>
<td>50</td>
<td>3.2967300</td>
</tr>
<tr>
<td>F150W0.DAT</td>
<td>149</td>
<td>150</td>
<td>YES</td>
<td>50</td>
<td>5.1098900</td>
</tr>
</tbody>
</table>

Table 3.7. Experimental examples of Reduce Test Suite Algorithm.

It is worth noting that most of the results were given as a worse case scenario of the algorithm and performance should rise as the scenario gets better. As we can see from the experimental results that the less are the test cases and the more are the requirements, the least can be the percentage of reduction in the test suite. However, if the number of test cases is not so far greater than the number of requirements, we can achieve a better reduction percentage. Also, the time of the algorithm is considered to be good according to the percentage of the reduction it gives. The algorithm has not surpassed in its worst case 5 seconds execution on a 486 machine. However, the needed preparation for the algorithm should be carried out before hand. The more difficult it is to determine the requirements, the more difficult will be to associate the test subsets to it. More results of the performance of the algorithm will be presented in chapter 5 of the paper.
3.4. High Level Design of Reduction Algorithm’s Code

Figure 3.2 contains the first step of the high level design of the implemented reduction algorithm. The structure chart includes all the important functions that are related directly to the reduction algorithm.

Figure 3.2. Structure Chart for the implemented reduction algorithm.

Figure 3.3 shows the sequence of the calling menus in the reduction algorithm.

Figure 3.3. The sequence of calling menus in the reduction algorithm.
Chapter 4

Tool For Flow Graph and Test Coverage Display

This connectivity matrix and reachability of a program are hard to visualize using a matrix data structure. The chapter describes a tool that allows us to visualize the connectivity and display the testing coverage of each independent test case. The tool is called Flow Graph and Test Coverage Display (FGTCD) tool.

4.1. Tool Design

The tool takes as input a connectivity file and generates the control flow graph automatically without the help or intervention of the user. The tool can be used in conjunction with a control flow generation tool, a data flow generation tool, or as a stand alone tool that can be supplied with input manually. Let us give a formal definition of the problem of drawing the connectivity.

Given : Number of segments, A segment connection file ( a connectivity matrix ) that must be drawn, and subset of segment connections \( S_0, S_1, \ldots, S_n \) one associated with each segment such that each subset consists of two numbers ( the segments to connect current segment to ).
Problem: Find a possible set of coordinates for each of the $S_i$ in order to draw the connectivity matrix in a two dimensional space.

4.1.1. Design of Input and Control

The FGTCD tool reads three kinds of input files:

- The Connectivity File (CFF).
- The Variable Information File (VIF).
- The Test Traversal File (TTF).

The connectivity file comes in a format given by the Table 4.1.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Segment</th>
<th>Connected Segment 1</th>
<th>Connected Segment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Function Name</td>
<td># of Segments</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>Segment 0</td>
<td>Connection 1</td>
<td>Connection 2</td>
</tr>
<tr>
<td>L3-1</td>
<td>Segment N-2</td>
<td>Connection 1</td>
<td>Connection 2</td>
</tr>
<tr>
<td>Lm-1</td>
<td>Segment N-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1. The connectivity formatted file (CFF) format.

L1 would contain the function name for which we draw the connectivity as well as the number of segments. L2 contains the first segment to connect, segment number 0, and the two connections to which the segment is connected. This situation remains until we reach segment N where the two connections must be zeros to indicate no connections for the last segment.

The input validation done at this level requires that:
• No segment is connected to a segment that is below it i.e. segment 5 cannot be connected to segment 4 (no looping).

• Every segment should have at least one connection to go to except for the last segment case.

The variable information file comes in a format given by Table 4.2.

<table>
<thead>
<tr>
<th>Kinds</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind 1</td>
<td>Segment Number</td>
<td>Variable Name</td>
<td>Use Type (d/c)</td>
<td></td>
</tr>
<tr>
<td>Kind 2</td>
<td>Segment I</td>
<td>Segment J</td>
<td>Variable Name</td>
<td>Predicate Use</td>
</tr>
</tbody>
</table>

Table 4.2. The Variable Information File (VIF) format.

There are two kinds of lines in the variable information file. The first kind is used to identify all variables that occurs within a single segment as either computation use or definition use. As for the second kind, the line specifies the predicate use of a variable to branch from segment I to segment J in the control flow graph. There is no constraint on this kind of input files imposed by FGTCD.

The test traversal file has the format described in Table 4.3.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Segment Traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Segment Traversed</td>
</tr>
<tr>
<td>L2</td>
<td>Segment Traversed</td>
</tr>
<tr>
<td>L3</td>
<td>Last Segment Traversed</td>
</tr>
</tbody>
</table>

Table 4.3. The Test Traversal File (TTF) format.

Each line in this file contains the number of the traversed segment. Note that the last traversed segment in the file should be the last segment of the connectivity usually; however, there are no constraints of this kind on the input file. In addition, the test traversal file may contains loops i.e. a sequence of segments that starts and ends with the same segment number.
drawing is affected by the menu choices taken such as the test traversal which would color all the traversed nodes. Figure 4.5 gives a real drawing example generated for a read file. The other kind of output is textual and concerns the uses of the variables in the connectivity. The format of the output on the screen is given by Table 4.4.

<table>
<thead>
<tr>
<th>Screen Line</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind 1</td>
<td>Segment Number</td>
<td>Variable Name</td>
<td>Use Type (d/u)</td>
<td></td>
</tr>
<tr>
<td>Kind 2</td>
<td>Segment I</td>
<td>Segment J</td>
<td>Variable Name</td>
<td>Predicate Use</td>
</tr>
</tbody>
</table>

Table 4.4. The textual information of the variables as shown on the screen.

Figure 4.2. Structure Chart of FGTCD.

4.1.4. Structure Charts

Figure 4.2 contains the structure chart for the drawing tools. The structure chart display all the important modules of the drawing tool. The main module buildlist which builds the drawing list is shown in the general structure chart in reverse video because
of its importance. The module buildlist has its attached modules which are illustrated in a separate structure chart given in Figure 4.3.

The whole tool is based on one menu interface that contains all the options of the program. This fact can be seen from the structure that contains one module which produce the main menu that contains all the options. However, there is from time to time some dialog boxes which asks for information such as the file name or to choose between two options.

![Structure Chart for the module buildlist](image)

**Figure 4.3.** The structure chart for the module buildlist.

### 4.2. Implementation Techniques

In this section, the algorithm used for implementation of the tool will be described after presenting its pseudo code. The algorithm pseudo code is given in Figure 4.4.
Function getxx
1- \texttt{maxx} \leftarrow 0
2- for every node that is a predecessor of the segment
   2.1- find position in the drawing list
   2.2- adjust the minimum and maximum
   2.3- add the position pointer
3- divide the position pointer by the number of predecessors
4- return the position as position pointer to the calling function

Function getdown
1- find the node that is the only predecessor of the segment:
2- adjust the starting position and adjust the y-axis
3- return to the calling function

Function getdownleft
1- find the node that is the only predecessor of the segment:
2- adjust the x-position to the left and adjust the y-axis
3- compute the new starting x-position and ending x-position
4- return to the calling function

Function getdownright
1- find the node that is the only predecessor of the segment:
2- adjust the x-position to the right and adjust the y-axis
3- compute the new starting x-position and ending x-position
4- return to the calling function

Function buildlist --- main function
1- draw node 1 (adjust its starting x-position and y-position)
2- curr_node \leftarrow 1
3- while the last segment is not drawn
   3.1- compute the successors for the curr_node
   3.2- if successors = 1
      3.2.1- compute all predecessors
   3.2.2- if all predecessors are drawn
      3.2.2.1- call getxx to compute new position
      3.2.2.2- draw the node
      3.2.2.3- increment the curr_node
   3.2.3- else if one predecessor
      3.2.3.1- call getdown to compute new position

46
3.2.3.2- draw the node
3.2.3.3- increment the curr node
3.3- else /* no of successors is 2 */
   3.3.1- call getdownleft for the left node
   3.3.2- draw left node
   3.3.3- increment curr node
   3.3.4- if node on the right = curr node + 1
       3.3.4.1- call getdownright to position the node
       3.3.4.2- draw node on the right
       3.3.4.3- increment curr node
   3.3.5- else add the right node to predecessor list
end while
4- return to the calling function

Figure 4.4. The pseudo code of the algorithm to draw the connectivity.

The algorithm described here uses properties of the connectivity. Such properties are:

- A node can not have more than two children.
- A node must have at least one child unless it is the last node of the connectivity.
- A node may have an unlimited number of predecessors.
- A node can not be connected to a node that is lower than its number i.e. node number 5 can not be connected to number 3.

Also, the algorithm uses four main procedures to compute the x-coordinate of the new node. The first procedure, getxx, is used to compute the x-coordinate of a node with more than 1 predecessor by taking the average of their x-coordinates. The second procedure, getdown, will take the x-coordinate value of the parent and gives it to its child if the node has one parent. The third procedure, getdownleft, is used to position the new node to the left of the parent node by a computed value equal half the margin.
available to the parent node. The last procedure, getdownright, is used to position the new node to the right of the parent node by a computed value equal to the half of the right margin of the parent node.

4.3. Examples

![Diagram](image)

Figure 4.5. Example of a connectivity Graph Generated By the Tool

We will begin this section by providing a simple example to follow and providing its graph. First, we will give the general structure of a formatted connectivity input file that conforms with the tool. The control flow graph is given in Figure 4.5 and the connectivity formatted file is given in Table 4.5. We will be explaining how the algorithm would work on the input file given in Table 4.5. In line 1 of the file, we have the function name as well as the last segment number of the function (it is the number of segments connected and the number of segment connection lines in the file since the function name line and segment number 0 connection are ignored when drawing). The
second line L2 contains the connection for segment 0 which is always connected to 1 and 0, meaning it is connected to segment number 1 only; Therefore it is ignored in drawing. Starting from L3 and so on, the line contains the segment number itself first, then the two other numbers represent the segments that are successors of the current segment. Here, we can have either of three cases:

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Segment</th>
<th>Connected Segment 1</th>
<th>Connected Segment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>TEST1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>L3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>L4</td>
<td>2</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>L5</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>L6</td>
<td>4</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>L7</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>L9</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>L10</td>
<td>8</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>L11</td>
<td>9</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>L12</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.5- The example connectivity formatted input file

- The segment is connected to two other segments, denoted by numbers that must be bigger than the current segment. For example L3.
- The segment is connected to one segment, denoted by a positive number bigger than the current segment number, and the other connection is given the value 0 meaning no connection. For example L4.
- The segment has no connection, denoted by two consecutive zero. This the case for the last segment only and no other line should contains such format. For example L12.
The algorithm would go as follow assuming that it is going to draw the picture in 1 page width of the x-axis: Read the function name and the last segment number and use it as a counter for the number of lines then Read the second line and ignore it. Initialize the page width to 550 on the x-axis and draw node 1 in the middle of the page (x-coordinate = 275). In line 3 of the file, since we have two successors of the node, the algorithm would draw node 2 on the left of node 1 given it x-axis coordinate to be (0+275)/2=137. Node 3 then must be drawn since node 2 has been drawn. The algorithm would place the node 3 on the right side by computing its coordinate (275+550)/2=412. Reading line 4, we see that segment 2 is connected to segment 10 only. Since node 9 is not yet drawn, node 10 may have some predecessors that are not yet drawn; in this case the algorithm would postpone drawing of node 10 and add 2 to the list of predecessors related to node 10. Next, we read line 5. In line 5 we have a similar case to line 3. Node 3 have two immediate successors. Therefore we compute the value of node 4 x-coordinate to be between node 1 and node 3: (275+412)/2=343 and node 5 between node 3 and end of the page on the right side:(412+550)/2=481. Next, we read line 6. In line 6 a similar case for line 4. The predecessors of node 10 have may not yet been drawn all; Thus we add node 4 to the predecessors list for node number 10. The algorithm would go on and read line 7, a similar case for line 3. Therefore node 6 x-axis would be computed as follow: (412+481)/2=446 and node 7 x-axis would be:(481+550)/2=515. Line 8 is similar to line 4 and node 6 is added to predecessors list of node 10. Line 9 is a case of line 3 with two successors node. Therefore, the computation would lead a value for x-axis for node 8 equal to 498 since it is going to lie between node 5 and node 7 while for node 9 it would give a value of 532 because of its position between node 7 and the end of the page. In line 10 and line
11, node 8 and 9 respectively would be added to the predecessors list of node 10. Reading the last line of the file, we retrieve all the predecessors of node 10 from the list of predecessors. The result would be node 2, 4, 6, 8 and 9. The x-coordinate for node 10 would be given as the average value of the values of the x-axis of all its predecessors i.e. Thus the computation would be based on the following formula:

\[
\text{average}(x\text{-axis segment 2, x-axis segment 4, x-axis segment 6, x-axis segment 8, x-axis segment 9}) = \text{average}(137, 343, 446, 498, 532) = \frac{137+343+446+498+532}{5} = 391.
\]

Table 4.6 gives the end coordinates given by the algorithm for segment node placing assuming that the x-axis start from the left and the maximum is 550 while the y-axis start at the top from value 0 and continue down incrementing the value.

Note that during the whole discussion we didn’t include any reference how do we compute the y-axis coordinate. Well, the y-coordinate is very simple to compute in the sense it is done in an incremental fashion. Every time we need to get down further, we only add to the old y-coordinate an increment of 60.

<table>
<thead>
<tr>
<th>Segment Number</th>
<th>X-coordinate</th>
<th>Y-coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>275</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>137</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>412</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>343</td>
<td>135</td>
</tr>
<tr>
<td>5</td>
<td>481</td>
<td>135</td>
</tr>
<tr>
<td>6</td>
<td>446</td>
<td>195</td>
</tr>
<tr>
<td>7</td>
<td>515</td>
<td>195</td>
</tr>
<tr>
<td>8</td>
<td>498</td>
<td>255</td>
</tr>
<tr>
<td>9</td>
<td>532</td>
<td>255</td>
</tr>
<tr>
<td>10</td>
<td>391</td>
<td>315</td>
</tr>
</tbody>
</table>

Table 4.6. Table of coordinates generated for example 1 using the algorithm.

Another example would be an example that illustrate the use of two pages width while drawing the connectivity. Therefore, the maximum limit on the x-axis would be 1100
pixel. The example would follow the same logic of the old one. The example test file with its coordinate table is given in Table 4.7 and Table 4.8.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Segment</th>
<th>Connected Segment 1</th>
<th>Connected Segment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>TEST2</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>L3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>L4</td>
<td>2</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>L5</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>L6</td>
<td>4</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>L7</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>L9</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>L10</td>
<td>8</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>L11</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>L12</td>
<td>10</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>L13</td>
<td>11</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>L14</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.7. The second example for connectivity formatted file

<table>
<thead>
<tr>
<th>Segment Number</th>
<th>X-coordinate</th>
<th>Y-coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>550</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>275</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>825</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>687</td>
<td>135</td>
</tr>
<tr>
<td>5</td>
<td>962</td>
<td>135</td>
</tr>
<tr>
<td>6</td>
<td>893</td>
<td>195</td>
</tr>
<tr>
<td>7</td>
<td>1031</td>
<td>195</td>
</tr>
<tr>
<td>8</td>
<td>996</td>
<td>255</td>
</tr>
<tr>
<td>9</td>
<td>1065</td>
<td>255</td>
</tr>
<tr>
<td>10</td>
<td>1048</td>
<td>315</td>
</tr>
<tr>
<td>11</td>
<td>1082</td>
<td>315</td>
</tr>
<tr>
<td>12</td>
<td>830</td>
<td>375</td>
</tr>
</tbody>
</table>

Table 4.8. Table for the x and y coordinates for the example 2 file generated by the algorithm
The last example provided would contain more special cases for the drawing algorithm to reveal the general handling in the drawing of the connectivity. The formatted connectivity file is given in Table 4.9:

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Segment</th>
<th>Connected Segment 1</th>
<th>Connected Segment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>MAIN</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1.3</td>
<td>1</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1.5</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>1.6</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1.7</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1.8</td>
<td>6</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>1.9</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>1.10</td>
<td>8</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>1.11</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>1.12</td>
<td>10</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>1.13</td>
<td>11</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>1.15</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.9. Example 3 for the connectivity formatted file

For line L1 and line L2, the algorithm would neglect them after reading and begin processing from line L3. Before reading line L3, the algorithm would draw node 1 in the middle of the page. Assuming one page width drawing, the largest x-axis coordinate would be 550. Therefore, x-coordinate would be 275 which is equal to (0+550)/2=275. After reading Line L3, the algorithm would draw node 2 to the left of node 1 i.e. in the range 0 to 275 with x-coordinate equal to (0+275)/2=137. As for node 9, it may happen that the node have predecessors other than node 1(node 8 is a predecessor of node 9); thus node 1 is added to the predecessor list of 9. Reading line L4, there is one connection to node 3. The algorithm would search the predecessor list for all the predecessor of node 3. Since there is only one predecessor for node 3 (node 2), the algorithm would not change the x-axis value of the node 3 and would draw it just
below node 2 i.e. increment the y-axis value only and take the x-axis value to be as for node 2 which is equal to 137. Processing line L5 of the file, we have node 3 connected to node 4 and node 8. As for node 4 the algorithm would draw node 4 to the left of node 3, postpone the drawing for node 8 for the completion of predecessors purpose, and add node 3 to the predecessor list of node 8. Therefore, only node 4 would have an x-coordinate which is equal to \((0+137)/2=68\). The algorithm continues processing with line L6. Line 6 is just a repeated case of line L4. Node 5 will get an x-axis value as node 4 i.e. 68 and increment the y-value. Line L7 contains two connections with two immediate nodes; thus node 6 is drawn to the left of node 5 while node 7 is drawn to the right of node 5. Therefore the x-coordinates for the node 6 and node 7 would be \((0+68)/2=34\) and \((68+137)/2=102\) respectively. Lines L8 and L9 would only add nodes 6 and node 7 to the predecessors list of Node 8 because of the completion of predecessors problem. Line L10 introduces a change since node 8 must be drawn now. We find that the predecessors for node 8 by now are nodes 3, 6, 7. Therefore the x-value would be an average of the 3 values of x-values i.e. \((137+34+102)/3=91\). Also, in this line we add node 8 to the predecessors list of node 9. Line L11 introduces another predecessors problem as well as two immediate successors at the same time. Node 9 position on the x-axis is given by the sum of the x-values on the predecessors list (node 2 and node 8), \((137+91)/2=114\). As for two immediate successors, the values would be \((0+114)/2=57\) and \((114+137)/2=125\) for nodes 10 and 11 respectively. As for lines L12 and L13, nodes 10 and 11 are added to the predecessors list only. As for the last line all predecessors are gathered and the average of the x-values is also taken and the result would be \((57+125)/2=91\). Table 4.10 below would show all the values computed by the algorithm.
<table>
<thead>
<tr>
<th>Segment Number</th>
<th>X-coordinate</th>
<th>Y-coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>550</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>137</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>137</td>
<td>135</td>
</tr>
<tr>
<td>4</td>
<td>68</td>
<td>195</td>
</tr>
<tr>
<td>5</td>
<td>68</td>
<td>255</td>
</tr>
<tr>
<td>6</td>
<td>34</td>
<td>315</td>
</tr>
<tr>
<td>7</td>
<td>102</td>
<td>315</td>
</tr>
<tr>
<td>8</td>
<td>91</td>
<td>375</td>
</tr>
<tr>
<td>9</td>
<td>114</td>
<td>435</td>
</tr>
<tr>
<td>10</td>
<td>57</td>
<td>495</td>
</tr>
<tr>
<td>11</td>
<td>125</td>
<td>495</td>
</tr>
<tr>
<td>12</td>
<td>91</td>
<td>555</td>
</tr>
</tbody>
</table>

Table 4.10. Table for the x and y coordinates for the example 3 file generated by the algorithm

For more examples on the running of the algorithm, the reader may consult appendix D for such examples with their associated printed connectivity files.
Chapter 5

Experimental Results and Comparison

In this chapter, we compare the quantitative and qualitative properties of reduction, slicing, incremental, adapted firewall, genetic, and simulated annealing algorithms described in Chapters 2 and 3. We have used the code and the results of the work reported in [Fakih and Mansour 1996], [Mansour and Goel 1994], and [Baradhi 1996].

5.1. Experimental Design

In this section we present a transformation done to the input files that are used for comparison in order to be able to use them with the reduction algorithm. We have said earlier that the reduction algorithm takes input as a set of requirements while the input files are presented as test traversal files. Therefore a transformation of input files into a readable form for the reduction algorithm is needed. We will illustrate the kind of transformation using a running example. The input test traversal file is given in Table 5.1. Each column in this table represents a segment. The input test traversal file is a 9 test cases passing by 6 segments.

A 0 in the segment column means that this segment is not traversed by the test case specified while a 1 means that this segment is traversed by this test case. Note that the first and the last segment should be traversed by every test case since they are the entry and exit nodes. The transformation into a requirements file is done under one condition:
The requirements in the new input file would be passing through every segment i.e. the subset of the test cases associated with the requirements would be all test cases that traverses that segment. Therefore, there will be test cases as much as the number of segments in the file and the maximum cardinality in the subset would be the maximum number of test cases. With this condition in mind we achieve the requirements input file given in Table 5.2.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Seg 1</th>
<th>Seg 2</th>
<th>Seg 3</th>
<th>Seg 4</th>
<th>Seg 5</th>
<th>Seg 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1. The example test traversal file.

<table>
<thead>
<tr>
<th>Number</th>
<th>Requirements</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>REQ1</td>
<td>{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9}</td>
</tr>
<tr>
<td>2</td>
<td>REQ2</td>
<td>{t_1,t_2,t_4,t_5,t_6}</td>
</tr>
<tr>
<td>3</td>
<td>REQ3</td>
<td>{t_3}</td>
</tr>
<tr>
<td>4</td>
<td>REQ4</td>
<td>{t_1,t_4,t_5,t_6,t_9}</td>
</tr>
<tr>
<td>5</td>
<td>REQ5</td>
<td>{t_2,t_3,t_6,t_9}</td>
</tr>
<tr>
<td>6</td>
<td>REQ6</td>
<td>{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9}</td>
</tr>
</tbody>
</table>

Table 5.2. The generated requirements file for example given in Table 5.1.

Let us take as an example how the requirement for segment number 2 was generated. If we look at Table 6.1 for the column of segment number 2, we can see that we have a 1 in the 2, 3, 5, 6, and 7th row corresponding for test cases 1, 2, 4, 5, 6. Therefore in our generated requirements file we include a subset for segment number 2 constituting of test cases \{t_1,t_2,t_4,t_5,t_6\}.  

57
The whole transformation was done in order to compare all the algorithms in a consistent way.

5.2. Quantitative criteria results

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

Table 5.4 gives the running times, texec, in seconds, of slicing, incremental, adapted firewall, genetic, and simulated annealing algorithms for small-size modules (m1\_ m14) and medium-size modules (m15\_ m20). Also, the table gives the number of test cases that should be rerun after modifications. The results are also depicted in Figures 5.1 through 5.8 for small-size and medium-size modules. Table 5.3 explains the symbols used in Table 5.4.
<table>
<thead>
<tr>
<th>M</th>
<th>Number of segments</th>
<th>N</th>
<th>Number of test cases</th>
<th>S_{mod} Number of the modified segment</th>
<th>#T Number of test cases to rerun</th>
<th>t_{exec} Running time of the algorithm in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>0.01</td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>m2</td>
<td>8</td>
<td>32</td>
<td>8</td>
<td>0.01</td>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>m3</td>
<td>10</td>
<td>15</td>
<td>8</td>
<td>0.01</td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>m4</td>
<td>10</td>
<td>27</td>
<td>10</td>
<td>0.01</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>0.05</td>
<td>1</td>
<td>1.4</td>
</tr>
<tr>
<td>m6</td>
<td>14</td>
<td>32</td>
<td>14</td>
<td>0.01</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>m7</td>
<td>21</td>
<td>9</td>
<td>12</td>
<td>0.01</td>
<td>3</td>
<td>1.3</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>0.01</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>0.01</td>
<td>5</td>
<td>0.9</td>
</tr>
<tr>
<td>m10</td>
<td>24</td>
<td>23</td>
<td>12</td>
<td>0.01</td>
<td>4</td>
<td>1.3</td>
</tr>
<tr>
<td>m11</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>0.01</td>
<td>2</td>
<td>5.6</td>
</tr>
<tr>
<td>m12</td>
<td>34</td>
<td>126</td>
<td>32</td>
<td>0.05</td>
<td>1</td>
<td>7.0</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>0.05</td>
<td>6</td>
<td>2.3</td>
</tr>
<tr>
<td>m14</td>
<td>40</td>
<td>28</td>
<td>39</td>
<td>0.05</td>
<td>5</td>
<td>4.6</td>
</tr>
<tr>
<td>m15</td>
<td>45</td>
<td>32</td>
<td>27</td>
<td>0.10</td>
<td>8</td>
<td>2.9</td>
</tr>
<tr>
<td>m16</td>
<td>45</td>
<td>96</td>
<td>43</td>
<td>0.01</td>
<td>1</td>
<td>7.0</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>0.01</td>
<td>6</td>
<td>5.5</td>
</tr>
<tr>
<td>m18</td>
<td>56</td>
<td>108</td>
<td>50</td>
<td>0.05</td>
<td>4</td>
<td>9.4</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>0.10</td>
<td>9</td>
<td>4.9</td>
</tr>
<tr>
<td>m20</td>
<td>60</td>
<td>145</td>
<td>46</td>
<td>0.05</td>
<td>5</td>
<td>9.2</td>
</tr>
</tbody>
</table>

Table 5.3. Parameters used in Tables 5.4 and 5.5.

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>S_{mod}</th>
<th>t_{exec}</th>
<th>#T</th>
<th>t_{exec}</th>
<th>#T</th>
<th>t_{exec}</th>
<th>#T</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>0.01</td>
<td>1</td>
<td>0.3</td>
<td>5</td>
<td>0.3</td>
<td>4</td>
</tr>
<tr>
<td>m2</td>
<td>8</td>
<td>32</td>
<td>0.01</td>
<td>1</td>
<td>0.8</td>
<td>26</td>
<td>0.8</td>
<td>8</td>
</tr>
<tr>
<td>m3</td>
<td>10</td>
<td>15</td>
<td>0.01</td>
<td>1</td>
<td>0.3</td>
<td>5</td>
<td>0.6</td>
<td>5</td>
</tr>
<tr>
<td>m4</td>
<td>10</td>
<td>27</td>
<td>0.01</td>
<td>1</td>
<td>0.5</td>
<td>8</td>
<td>1.0</td>
<td>8</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>0.05</td>
<td>1</td>
<td>1.4</td>
<td>6</td>
<td>1.4</td>
<td>6</td>
</tr>
<tr>
<td>m6</td>
<td>14</td>
<td>32</td>
<td>0.01</td>
<td>1</td>
<td>1.5</td>
<td>32</td>
<td>0.7</td>
<td>8</td>
</tr>
<tr>
<td>m7</td>
<td>21</td>
<td>9</td>
<td>0.01</td>
<td>3</td>
<td>1.3</td>
<td>5</td>
<td>1.3</td>
<td>5</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>0.01</td>
<td>2</td>
<td>2.0</td>
<td>11</td>
<td>2.0</td>
<td>7</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>0.01</td>
<td>5</td>
<td>0.9</td>
<td>5</td>
<td>0.9</td>
<td>3</td>
</tr>
<tr>
<td>m10</td>
<td>24</td>
<td>23</td>
<td>0.01</td>
<td>4</td>
<td>1.3</td>
<td>5</td>
<td>1.3</td>
<td>3</td>
</tr>
<tr>
<td>m11</td>
<td>34</td>
<td>58</td>
<td>0.01</td>
<td>2</td>
<td>5.6</td>
<td>12</td>
<td>3.1</td>
<td>8</td>
</tr>
<tr>
<td>m12</td>
<td>34</td>
<td>126</td>
<td>0.05</td>
<td>1</td>
<td>7.0</td>
<td>29</td>
<td>5.2</td>
<td>8</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>0.05</td>
<td>6</td>
<td>2.3</td>
<td>7</td>
<td>2.3</td>
<td>3</td>
</tr>
<tr>
<td>m14</td>
<td>40</td>
<td>28</td>
<td>0.05</td>
<td>5</td>
<td>4.6</td>
<td>9</td>
<td>4.2</td>
<td>3</td>
</tr>
<tr>
<td>m15</td>
<td>45</td>
<td>32</td>
<td>0.10</td>
<td>8</td>
<td>2.9</td>
<td>4</td>
<td>2.9</td>
<td>4</td>
</tr>
<tr>
<td>m16</td>
<td>45</td>
<td>96</td>
<td>0.01</td>
<td>1</td>
<td>7.0</td>
<td>16</td>
<td>7.0</td>
<td>8</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>0.01</td>
<td>6</td>
<td>5.5</td>
<td>8</td>
<td>6.7</td>
<td>9</td>
</tr>
<tr>
<td>m18</td>
<td>56</td>
<td>108</td>
<td>0.05</td>
<td>4</td>
<td>9.4</td>
<td>22</td>
<td>8.2</td>
<td>5</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>0.10</td>
<td>9</td>
<td>4.9</td>
<td>15</td>
<td>4.9</td>
<td>3</td>
</tr>
<tr>
<td>m20</td>
<td>60</td>
<td>145</td>
<td>0.05</td>
<td>5</td>
<td>9.2</td>
<td>22</td>
<td>6.5</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.4. The running times in seconds, and the number of tests to rerun of Reduction, Slicing, Incremental algorithms for modules m1 to m20.
<table>
<thead>
<tr>
<th>Module</th>
<th>M</th>
<th>N</th>
<th>$S_{med}$</th>
<th>$t_{sec}$</th>
<th>#T</th>
<th>$t_{sec}$</th>
<th>#T</th>
<th>$t_{sec}$</th>
<th>#T</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>0.3</td>
<td>5</td>
<td>8.0</td>
<td>2</td>
<td>0.3</td>
<td>2</td>
</tr>
<tr>
<td>m2</td>
<td>8</td>
<td>32</td>
<td>8</td>
<td>0.9</td>
<td>23</td>
<td>6.7</td>
<td>1</td>
<td>2.1</td>
<td>1</td>
</tr>
<tr>
<td>m3</td>
<td>10</td>
<td>15</td>
<td>8</td>
<td>1.3</td>
<td>5</td>
<td>4.9</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>m4</td>
<td>10</td>
<td>27</td>
<td>10</td>
<td>1.2</td>
<td>8</td>
<td>1.4</td>
<td>1</td>
<td>1.8</td>
<td>1</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>1.8</td>
<td>8</td>
<td>4.2</td>
<td>2</td>
<td>1.7</td>
<td>1</td>
</tr>
<tr>
<td>m6</td>
<td>14</td>
<td>32</td>
<td>14</td>
<td>1.6</td>
<td>32</td>
<td>4.0</td>
<td>1</td>
<td>2.3</td>
<td>1</td>
</tr>
<tr>
<td>m7</td>
<td>21</td>
<td>9</td>
<td>12</td>
<td>1.7</td>
<td>9</td>
<td>3.5</td>
<td>2</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>2.4</td>
<td>10</td>
<td>7.4</td>
<td>1</td>
<td>1.3</td>
<td>1</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>1.3</td>
<td>8</td>
<td>9.2</td>
<td>3</td>
<td>1.1</td>
<td>3</td>
</tr>
<tr>
<td>m10</td>
<td>24</td>
<td>23</td>
<td>12</td>
<td>1.6</td>
<td>10</td>
<td>9.8</td>
<td>2</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>m11</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>6.0</td>
<td>15</td>
<td>9.9</td>
<td>2</td>
<td>4.3</td>
<td>2</td>
</tr>
<tr>
<td>m12</td>
<td>34</td>
<td>126</td>
<td>32</td>
<td>8.0</td>
<td>31</td>
<td>9.1</td>
<td>1</td>
<td>10.6</td>
<td>1</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>4.1</td>
<td>9</td>
<td>4.6</td>
<td>3</td>
<td>1.4</td>
<td>3</td>
</tr>
<tr>
<td>m14</td>
<td>40</td>
<td>28</td>
<td>39</td>
<td>6.9</td>
<td>9</td>
<td>5.3</td>
<td>1</td>
<td>2.0</td>
<td>1</td>
</tr>
<tr>
<td>m15</td>
<td>45</td>
<td>32</td>
<td>27</td>
<td>4.6</td>
<td>8</td>
<td>7.3</td>
<td>4</td>
<td>2.5</td>
<td>3</td>
</tr>
<tr>
<td>m16</td>
<td>45</td>
<td>96</td>
<td>43</td>
<td>11.8</td>
<td>20</td>
<td>6.3</td>
<td>4</td>
<td>8.2</td>
<td>3</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>8.2</td>
<td>8</td>
<td>5.7</td>
<td>3</td>
<td>2.8</td>
<td>3</td>
</tr>
<tr>
<td>m18</td>
<td>56</td>
<td>108</td>
<td>50</td>
<td>15.0</td>
<td>23</td>
<td>5.9</td>
<td>2</td>
<td>9.6</td>
<td>2</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>7.8</td>
<td>17</td>
<td>7.9</td>
<td>4</td>
<td>7.6</td>
<td>3</td>
</tr>
<tr>
<td>m20</td>
<td>60</td>
<td>145</td>
<td>46</td>
<td>13.4</td>
<td>25</td>
<td>6.4</td>
<td>3</td>
<td>12.9</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.5. The running times in seconds, and the number of tests to rerun of Adapted Firewall, Genetic, and Simulated Annealing algorithms for m1 to m20.

Figure 5.1. The # of selected test cases to rerun for reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m1_m5).
Figure 5.2. The # of selected test cases to rerun for reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m6_m10).

Figure 5.3. The # of selected test cases to rerun for reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m11_m15).
Figure 5.4. The # of selected test cases to rerun for reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for \(m_{16}\_m_{20}\).

Figure 5.5. The running times in seconds of the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for \(m_{1}\_m_{5}\).
Figure 5.6. The running times in seconds of the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m6-m10).

Figure 5.7. The running times in seconds of the reduction, slicing, incremental, adapted firewall, genetic and simulated annealing algorithms for (m11-m15).
For medium-size modules (m15 - m20), the adapted firewall algorithm takes the largest time with a number of test cases to rerun close to that of the slicing algorithm. The incremental algorithm gives better results in the number of retests, at an execution time similar to that of the slicing algorithm. The reduction algorithm gives a number of retests close to that of the incremental with less time. The best number of retests, is still offered by the genetic and simulated annealing algorithms, although at a slower execution.

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

The incremental algorithm uses a data-flow graph derived from the control-flow graph. Thus the time it takes to construct these graphs is at most $O(n^2)$ which is acceptable. The hard part is in the computation of static data dependencies. Unfortunately, precise static data dependencies becomes hard to compute when the program makes use of pointers, arrays, and dynamic memory allocations.
The reduction algorithm in turn needs only the test traversal file which is transformed into a file of requirements as described in section 5.1. However, the test traversal file can not be generated without generating the control flow graph. For this reason, the reduction algorithm time of preparation takes $O(n^2)$ which is also an acceptable time.

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

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

### 5.2.2. Precision and inclusiveness

Tables 5.6, 5.7, 5.8, 5.9, 5.10 and 5.11 present the results obtained for precision and inclusiveness of the implemented algorithms. Table 5.5 explains the symbols used in these tables. Results of the tables are also shown in Figures 5.9 and 5.10.

<table>
<thead>
<tr>
<th>M</th>
<th>Number of segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Number of test cases</td>
</tr>
<tr>
<td>$S_{\text{mod}}$</td>
<td>Number of the modified segment</td>
</tr>
<tr>
<td>#T</td>
<td>Number of test cases to rerun</td>
</tr>
<tr>
<td>mr</td>
<td># of Modification-revealing test cases</td>
</tr>
<tr>
<td>nmr</td>
<td># of Non-modification-revealing test cases</td>
</tr>
<tr>
<td>m</td>
<td># of Modification-revealing test cases selected</td>
</tr>
<tr>
<td>n</td>
<td># of Non-modification-revealing test cases omitted</td>
</tr>
<tr>
<td>Preq%</td>
<td>Precision percentage</td>
</tr>
<tr>
<td>Inc%</td>
<td>Inclusiveness percentage</td>
</tr>
</tbody>
</table>

Table 5.5. Parameters used in Tables 5.6, 5.7, 5.8, 5.9, 5.10 and 5.11.
<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Smed</th>
<th>#T</th>
<th>mr</th>
<th>nmr</th>
<th>m</th>
<th>n</th>
<th>Prec%</th>
<th>Inc%</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>4</td>
<td>12</td>
<td>86</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>11</td>
<td>12</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>n9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>m13</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>12</td>
<td>8</td>
<td>50</td>
<td>8</td>
<td>46</td>
<td>92</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>7</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>4</td>
<td>17</td>
<td>81</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>15</td>
<td>35</td>
<td>55</td>
<td>10</td>
<td>50</td>
<td>91</td>
</tr>
</tbody>
</table>

Table 5.6. Precision and inclusiveness Results for the Slicing Approach.

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Smed</th>
<th>#T</th>
<th>mr</th>
<th>nmr</th>
<th>m</th>
<th>n</th>
<th>Prec%</th>
<th>Inc%</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>6</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>7</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>n9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>m13</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>8</td>
<td>8</td>
<td>50</td>
<td>8</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>9</td>
<td>13</td>
<td>21</td>
<td>9</td>
<td>21</td>
<td>100</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>3</td>
<td>35</td>
<td>55</td>
<td>3</td>
<td>55</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.7. Precision and inclusiveness Results for the Incremental Algorithm.

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Smed</th>
<th>#T</th>
<th>mr</th>
<th>nmr</th>
<th>m</th>
<th>n</th>
<th>Prec%</th>
<th>Inc%</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>n5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>8</td>
<td>10</td>
<td>14</td>
<td>8</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>10</td>
<td>12</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>66</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>75</td>
</tr>
<tr>
<td>m13</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>15</td>
<td>8</td>
<td>50</td>
<td>8</td>
<td>43</td>
<td>86</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>7</td>
<td>20</td>
<td>95</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>17</td>
<td>35</td>
<td>55</td>
<td>17</td>
<td>55</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.8. Precision and inclusiveness Results for the Adapted firewall Algorithm.

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Smed</th>
<th>#T</th>
<th>mr</th>
<th>nmr</th>
<th>m</th>
<th>n</th>
<th>Prec%</th>
<th>Inc%</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>2</td>
<td>10</td>
<td>14</td>
<td>2</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>1</td>
<td>12</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>m11</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>2</td>
<td>8</td>
<td>50</td>
<td>2</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>3</td>
<td>13</td>
<td>21</td>
<td>3</td>
<td>21</td>
<td>100</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>4</td>
<td>35</td>
<td>55</td>
<td>4</td>
<td>55</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.9. Precision and inclusiveness Results for the Genetic Algorithm.
Table 5.10. Precision and inclusiveness Results for the Simulated Annealing Algorithm

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>S_wed</th>
<th>#T</th>
<th>mtr</th>
<th>mnr</th>
<th>m</th>
<th>n</th>
<th>Prec%</th>
<th>Inc%</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>1</td>
<td>10</td>
<td>14</td>
<td>1</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>1</td>
<td>12</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>m11</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>2</td>
<td>8</td>
<td>50</td>
<td>2</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>3</td>
<td>13</td>
<td>21</td>
<td>3</td>
<td>21</td>
<td>100</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>3</td>
<td>35</td>
<td>55</td>
<td>3</td>
<td>55</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.11 Precision and inclusiveness Results for the reduction algorithm.

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>S_wed</th>
<th>#T</th>
<th>mtr</th>
<th>mnr</th>
<th>m</th>
<th>n</th>
<th>Prec%</th>
<th>Inc%</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>1</td>
<td>10</td>
<td>14</td>
<td>1</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>2</td>
<td>12</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>m11</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>2</td>
<td>8</td>
<td>50</td>
<td>2</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>6</td>
<td>13</td>
<td>21</td>
<td>1</td>
<td>16</td>
<td>76</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>9</td>
<td>35</td>
<td>55</td>
<td>1</td>
<td>47</td>
<td>85</td>
</tr>
</tbody>
</table>

Figure 5.9. The precision of Slicing, Incremental, Adapted firewall, Genetic, and Simulated Annealing algorithms for modules presented in Tables 5.6 to 5.11. The slicing and the adapted firewall algorithms give a good precision percentage. For some modules we can see that the precision percentage is not very high. This is due to the fact that these algorithms define also all the def-use pairs that are affected by the modification. This would result in selecting other test cases that are modification-
revealing to the affected def-use pairs. The reduction algorithm gives a precision that is acceptable due to way the input files were transformed.

![Inclusiveness of Algorithms](image)

Figure 5.10. The Inclusiveness of Slicing, Incremental, Adapted firewall, Genetic, and Simulated Annealing algorithms for modules presented in Tables 5.6 to 5.11.

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

The slicing and the adapted firewall algorithm select only modification-traversing tests by selecting tests that exercise definition-use pairs. However, the methods may fail to identify tests that execute modified output statements that contain no variable uses, although these statements may cause the program to produce different outputs. Also, the methods cannot detect tests that include deleted statements of a modified program. Thus, depending on the modification the inclusiveness percentage varies for the selected modules.
The precision percentage of the incremental algorithm also varies for the selected modules. The algorithm uses sets of techniques to determine the test cases in the regression test suite on which the new and the old programs may produce different outputs. Only these test cases need to be rerun. However, not all of these test cases are selected. If several tests exercise a particular affected statement, only some of them are selected depending on the control-dependencies of the related statements.

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

5.3. Qualitative criteria results

5.3.1. User’s parameter setting

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

The genetic algorithm requires parameters to be set by the user, such as the ranking range for reproduction, convergence threshold, and crossover and mutation rates. The annealing algorithm requires the parameters: convergence threshold, and initial probability for accepting perturbations.
The reduction algorithm may not require parameter settings, but the reduction algorithm requires generation of requirements and their associated testing subsets. These requirements are set by the tester and according to his preferences.

5.3.2. Type of maintenance

All unit regression testing methods evaluated in this chapter are corrective regression testing where modifications are done only to correct an error report. On the other hand, the firewall concept at the integration level supports the corrective regression testing and the perfective regression testing.

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

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

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

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

At the end, the reduction algorithm works for corrective maintenance. For the perfective maintenance, we need to re-generate the control flow graph and the test traversal file. Therefore, perfective maintenance can be achieved but with some overhead of computational cost.

5.3.3. Type of testing

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

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

5.3.4. Level of testing

The slicing, incremental, reduction and adapted firewall algorithms are applied at the unit level. Integration testing cannot be applied using slicing since it is based on identifying the definition-use pairs at the unit level. In incremental regression testing,
computation of the execution slices can be computed at the function or module level. Then, we need to execute the new program on a test case only if a modified function or module was invoked during the original program's execution on that test case. In reduction, we need to treat modules as segments and merge subsets of test cases in one set to achieve requirements at the module level. The firewall algorithm is an integration level of testing.

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

5.3.5. Type of approach

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

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

Genetic and simulated algorithms belong to the minimization regression test selection method. If several tests exercise a particular modified statement, only one such statement is selected.
The firewall algorithm belongs to the coverage algorithms that rely on coverage criteria. It selects all tests that should be integration-tested.

The reduction algorithm as described in chapter 3 is independent of the criteria that the tester may choose as long as the association between the criteria and test cases can be made. Therefore, it offers a great deal of flexibility.

5.3.6. Global variables

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

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

5.4. Summary of results

Tables 5.12 and 5.13 summarize the comparison results for the quantitative and qualitative criteria. We note that our assessment is based on the following considerations:

- Larger size modules are more important for conclusions, since they are more realistic.
- Since the test cases were manually developed, it was not possible to run experiments that were statistically sound, especially for the execution time.
- Execution time results are based on an overall assessment, although the results are not consistent for all cases. The terms used for algorithms’ speed are based on comparing these algorithms with each other.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Integration</th>
<th>Genetic</th>
<th>Simulated Annealing</th>
<th>Reduction</th>
<th>Slicing</th>
<th>Incremental</th>
<th>Adapted Firewall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time for small-size modules</td>
<td>-----------</td>
<td>acceptable</td>
<td>acceptable</td>
<td>very fast</td>
<td>fast</td>
<td>fast</td>
<td>fast</td>
</tr>
<tr>
<td>Execution time for medium-size modules</td>
<td>-----------</td>
<td>acceptable</td>
<td>slow</td>
<td>very fast</td>
<td>acceptable</td>
<td>fast</td>
<td>slow</td>
</tr>
<tr>
<td>Number of test cases to rerun</td>
<td>-----------</td>
<td>very good</td>
<td>very good</td>
<td>good</td>
<td>acceptable</td>
<td>good</td>
<td>acceptable</td>
</tr>
<tr>
<td>Precision</td>
<td>-----------</td>
<td>very high</td>
<td>very high</td>
<td>high</td>
<td>high</td>
<td>very high</td>
<td>high</td>
</tr>
<tr>
<td>Inclusiveness</td>
<td>-----------</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>User's parameter Setting</td>
<td>not required</td>
<td>required</td>
<td>required</td>
<td>not required</td>
<td>required</td>
<td>not required</td>
<td>not required</td>
</tr>
<tr>
<td>Type of Correction</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Type of Maintenance</td>
<td>yes</td>
<td>can be with additions</td>
<td>can be with additions</td>
<td>can be with modifications</td>
<td>can be with modifications</td>
<td>yes</td>
<td>can be with modifications</td>
</tr>
<tr>
<td>Type of testing</td>
<td>structural</td>
<td>structural</td>
<td>functional</td>
<td>structural</td>
<td>structural</td>
<td>functional</td>
<td>structural</td>
</tr>
<tr>
<td>Level of Integration</td>
<td>yes</td>
<td>can be applied with modifications</td>
<td>can be applied with modifications</td>
<td>can be applied with modifications</td>
<td>cannot be applied</td>
<td>can be applied with modifications</td>
<td>cannot be applied</td>
</tr>
<tr>
<td>Type of algorithm</td>
<td>coverage</td>
<td>minimization</td>
<td>minimization</td>
<td>minimization</td>
<td>coverage</td>
<td>safe</td>
<td>coverage</td>
</tr>
<tr>
<td>Global Variables</td>
<td>could be tested</td>
<td>could be identified</td>
<td>could be identified</td>
<td>could be identified</td>
<td>could be identified</td>
<td>could be identified</td>
<td>could be identified</td>
</tr>
</tbody>
</table>

Table 5.12. Properties of regression testing algorithms.
Chapter 6

Conclusions and Further Work

We have developed flow graph generation and test coverage display (FGTCD) tool and a framework for regression testing. We have also implemented a reduction oriented regression testing algorithm. Further, a comparative study, based on quantitative and qualitative criteria, for slicing, incremental, adapted firewall, reduction and minimization regression testing methods has been presented. These methods have been quantitatively evaluated based on efficiency, number of test cases that must be rerun, precision, and inclusiveness using a set of small and medium size modules for which the codes were written and the graphs were generated manually along with the test cases tables. They have been qualitatively compared based on the type of testing, level of testing, and type of maintenance, an algorithm can handle, requirements of user's parameter setting, and the ability to regression test global variables.

The FGTCD tool provides a graphical interface that is user friendly and facilitates the work of regression testing. The reduction regression testing algorithm has been tested and compared with other algorithms using small- and medium-size modules.

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

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

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

Furthermore, all algorithms, except the reduction, presented in this thesis apply to corrective maintenance and structural type testing. However, modifications where functional type of testing is needed can only be regression tested with the reduction and incremental algorithms at the unit level and with the firewall algorithm at the integration level.
Based on the work presented in this thesis, further research tasks can be pursued. Firstly, capabilities can be added or modified to the algorithms in order to handle the perfective type of maintenance. Secondly, modifications may be made to the algorithms so they can be applied at the integration level. Thirdly, the quantitative criteria experiments need to be done for large-size modules. This could be accomplished using tools that can generate the control-flow and data flow graphs and the associated test cases tables for a given large-size module, in order to be used as input for the algorithms. Fourthly, the global variable set/use matrix can be used to monitor the ripple effects of modifications by reflecting the status of each global variable, argument, parameter, and local variable. The global variables would be included in the matrix to facilitate the identification of modules in which the global variables are used. Then, applying the algorithms at the integration level, regression test global variables can be done using the set/use matrix.
Appendix A: Low Level Design of the Reduction Algorithm

This appendix describes the code for the menu that attaches the six tools for regression testing methods. It describes every procedure separately. The important functions which constitute the whole process are shaded. For the high level design, the reader may refer to Figure 3.2.

* FOR A COMPLETE DEFINITION OF THE VARIABLES AND INCLUDED FILES, SEE THE ATTACHED CODE FOR THE DOCUMENTATION

Function writeln(char message, int x, int y, int color)
1- set text color to passed color
2- set the coordinates to the passed coordinates
3- write the message character by character with delays
4- return to the calling function

Function window1(int x1, int y1, int x2, int y2, int fg)
1- set the color to the passed color
2- draw the rectangle with passed coordinates
3- return to the calling function

Function introduction
1- set text mode to 40 character
2- clear the screen
3- draw a window
4- display the name of the system
5- wait for keyboard input
6- read the keyboard input
7- return to the calling function

Function tg_mouseinrect(int left, int top, int right, int bottom)
1- get mouse position
2- if mouse is within the specified rectangle return(0)
3- else return(-1)
4- return to the calling function

Function.opengraph
1- open the graph mode
2- if fail to open graph mode
   2.1- notify the end user
   2.2- exit the program
3- return to the calling function

Function.helpmain
1- close the graph mode
2- call the utility smooth with help text file
3- open the graph mode
4- return to the calling function

Function.buildface
1- fill the whole screen with lightgray color
2- clear the button of the screen for messages
3- draw the 3-D effect lines for the menu
4- return to the calling function

Function.message(char message)
1- clear the old message
2- set the message color
3- display the message in the message window
4- return to the calling function
Function projectName
1- prepare a 3-d effect box for the project name
2- display the project name in the box
3- return to the calling function

Function writetext(int color,int x,int y,char str[])
1- set the color to the passed color
2- go to the x-y coordinates specified
3- display the text passed to the function
4- return to the calling function

Function mousemove(int oldx,int oldy,int newx,int newy)
1- check the change in y-position
2- check the change in the x-position
3- if no change in both return(-1)
4- else return(0)
5- return to the calling function

Function checkmousechange(int &oldx,int &oldy,int &mx,int &my)
1- prepare the old x and y position of the mouse
2- get the new position of the mouse
3- call mousemove to check
4- return the result of function mousemove
5- return to the calling function

Function getchoice(int maxchoice)
1- while choice is not entered or valid
   1 1- if mouse in a button then display the help for this button
   1 2- wait till a keyboard was hit or a mouse is clicked
   1 3- if keyboard was hit
      1 3 1- read the keyboard
1.3.2. choice = read input
1.4. else /* mouse was pressed */
1.4.1. get the mouse position
1.4.2. if mouse in button then choice = button choice
end while

Function genetic
1. change to the genetic directory
2. Prepare the constants files
3. compile the new hga
4. run the new hga
5. return to the parent directory
6. return to the calling function

Function sa
1. change to the sa directory
2. Prepare the constants files
3. compile the new sa
4. run the new sa
5. return to the parent directory
6. return to the calling function

Function incremental
1. change to the incremental directory
2. run the incremental algorithm
3. return to the parent directory
6. return to the calling function

Function firewall
1. change to the firewall directory
2. run the firewall algorithm
3. return to the parent directory
6- return to the calling function

Function slicing
1- change to the slicing directory
2- run the slicing algorithm
3- return to the parent directory
6- return to the calling function

Function reduction
1- change to the reduction directory
2- run the reduction algorithm
3- return to the parent directory
6- return to the calling function

Function main menu
1- initiate the mouse
2- call build face for building the background
3- call writetext for each menu option with eligible coordinates
4- call projectname to display the project name
5- call get choice to have the choice
6- if (choice == 0) /* choice is for help */
   helpmenu();
   if (choice == 1) /* choice is genetic algorithm */
   genetic();
   else if (choice == 2) /* choice is simulated annealing */
   sa();
   else if (choice == 3) /* choice is firewall */
   firewall();
   else if (choice == 4) /* choice is incremental */
   incremental();
   else if (choice == 5) /* choice is slicing */
   slicing();
else if (choice == 6) /* choice is reduction */
    reduce();
7- if choice == 7 exit and return to the calling function

Function main
1- call intro
2- call opengraph
3- call mainmenu
4- call closegraph
5- return to DOS

This part of the appendix contains the pseudo code for the reduction regression testing method implemented and integrated into the regression testing main menu for regression testing methods. Each procedure is described separately. The important functions which constitute the main of the software are shaded

* FOR A COMPLETE DEFINITION OF THE VARIABLES AND INCLUDED FILES, SEE THE ATTACHED CODE FOR THE DOCUMENTATION

Function tg_mouseinrect(int left, int top, int right, int bottom)
1- get mouse position
2- if mouse is within the specified rectangle return(0)
3- else return(-1)
4- return to the calling function

Function opengraph
1- open the graph mode
2- if fail to open graph mode
2.1- notify the end user
2.2- exit the program
3- return to the calling function

Function helpproduce
1- close the graph mode
2- call the utility smooth with help text file
3- open the graph mode
4- return to the calling function

Function buildface
1- fill the whole screen with lightgray color
2- clear the button of the screen for messages
3- draw the 3-D effect lines for the menu
4- return to the calling function

Function message(char message)
1- clear the old message
2- set the message color
3- display the message in the message window
4- return to the calling function

Function projectname
1- prepare a 3-d effect box for the project name
2- display the project name in the box
3- return to the calling function

Function.writetext(int color,int x,int y,char str[])
1- set the color to the passed color
2- go to the x-y coordinates specified
3- display the text passed to the function
4- return to the calling function
Function mousemove(int oldx, int oldy, int newx, int newy)
1- check the change in y-position
2- check the change in the x-position
3- if no change in both return(-1)
4- else return(0)
5- return to the calling function

Function checkmousechange(int &oldx, int &oldy, int &mx, int &my)
1- prepare the old x and y position of the mouse
2- get the new position of the mouse
3- call mousemove to check
4- return the result of function mousemove
5- return to the calling function

Function getchoice1(int maxchoice)
1- while choice is not entered or valid
   1.1- if mouse in a button then display the help for this button
   1.2- wait till a keyboard was hit or a mouse is clicked
   1.3- if keyboard was hit
       1.3.1- read the keyboard
       1.3.2- choice = read input
   1.4- else /* mouse was pressed */
       1.4.1- get the mouse position
       1.4.2- if mouse in button then choice = button choice
end while

Function getchoice2(int maxchoice)
1- while choice is not entered or valid
   1- if mouse in a button then display the help for this button
   1.2- wait till a keyboard was hit or a mouse is clicked
   1.3- if keyboard was hit
1.3.1- read the keyboard
1.3.2- choice = read input
1.4- else /* mouse was pressed */
1.4.1- get the mouse position
1.4.2- if mouse in button then choice = button choice
end while

Function menu1
1- call buildface to prepare for the menu
2- call writetext for each option of the menu
3- call project name to display the project name
4- call get choice1 to get the choice
5- return choice to the calling function
6- return to the calling function

Function menu2
1- call buildface to prepare for the menu
2- call writetext for each option of the menu
3- call project name to display the project name
4- call get choice2 to get the choice
5- return choice to the calling function
6- return to the calling function

Function timing
1- start the timer of the computer
2- save the start time
3- return to the calling function

Function endtiming
1- stop the clock of the computer
2- get the end time
3- compute the difference in time
4- display the result in milliseconds
5- return to the calling function

Function initialize
1- initialize the requirements
2- initialize the test cases
3- initialize the representative set
4- initialize the visiting array
5- initialize the maximum cardinality
6- return to the calling function

Function reading_file
1- read the maximum number of requirements from the file
2- read all the requirements into the arrays from the file
3- set the maximum cardinality
4- return to the calling function

Function reading
1- read the maximum number of requirements from the screen
2- read all the requirements into the arrays from the screen
3- set the maximum cardinality
4- return to the calling function

Function read
1- get the choice of the user
2- if choice is read from screen call function reading
3- else call function reading_file
4- return to the calling function

Function findcasereq(short int require[],short int testcase)
1- found = -1
2- search for the test case in requirement array
3- if test case is found found=0
4- return found to the calling function
5- return to the calling function

Function findreqcard(short int cardinal, short int retreq[])
1- initialize the retreq array to no requirements
2- for each requirements in the requirements array
    2.1- if requirement cardinality = cardinality passed
        2.1.1- add to the retreq array
3- return to the calling function

Function countcase(short int require[], short int testcase)
1- initialize test case count to 0
2- for each requirement in the require array do
    2.1- if test case in require[1] increment count
3- return count to the calling function
4- return to the calling function

Function counttestcases(short int retreq[], short int rettest[],
short int retcount[])
1- initialize the test cases array
2- for each test case in each requirements
    2.1- if test case is already in increment its associated count
    2.2- else add it to the list and put its count to 1
3- return to the calling function

Function allmarked
1- satisfied<= true
2- for each requirement in the list
    1.1- if requirement is not marked satisfied<= false
3- return satisfied to the calling function
4- return to the calling function

Function getmaxcount(short int counter[])
1- index<- -1
2- max<- 0
3- for each test case
   3.1- if test case counter > max then max<- counter
   3.2- index<- counter index
4- if more than two test cases have the same index then index<- -1
5- return index to the calling function
6- return to the calling function

Function marking(short int testcase)
1- for each requirement
   1.1- if test case in requirement
       1.1.1- mark requirement as satisfied
2- return to the calling function

Function writers(short int rs[])
1- print the representative set on the screen
2- wait for the user to press new line
3- return to the calling function

Function writersfile(short int rs[])
1- print the representative set on the file
2- wait for the user to press new line
3- return to the calling function

Function choosewrite(short int rs[])
1- get the choice of the user for writing
2- if choice = write on screen call writers
3- else call writersfile
4- return to the calling function

Function main
1- clear screen
2- initialize values
3- call read for reading information
4- call reduce test suite
5- return to DOS
Appendix B: Further Examples on the Application of The Reduction Algorithm.

This appendix contains some of the requirements transformed files. The files were transformed from their respective testing coverage file. In the test coverage file, for each test case there is information about the segment that it covers. Reversing the order, and taking the file into the form of which for each segment we will have the test cases that passes through it. The conversion was done using a C software programmed to accomplish this special purpose. This form of transformation will tell us whether all the segments are covered at least once or not. If a segment is not covered then its line will be empty data. Let us assume that each segment passing will constitutes a requirements for our reduction technique. Therefore, our objective is to discover a minimal representative set that will provide the same segment coverage. The results shows a high degree of reduction in the normal conditions that the program was put under. The input files transformed were the same used in the experimentation for section 5. A list of all the files and their explanation is provided in the following table:

<table>
<thead>
<tr>
<th>File</th>
<th>Output File</th>
<th>NR</th>
<th>NTC</th>
<th>NTCR</th>
<th>SNTC</th>
<th>PR</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFR8.DAT</td>
<td>RSFR8.DAT</td>
<td>8</td>
<td>36</td>
<td>184</td>
<td>2</td>
<td>95</td>
<td>0.01</td>
</tr>
<tr>
<td>SFR10.DAT</td>
<td>RSFR10.DAT</td>
<td>10</td>
<td>30</td>
<td>192</td>
<td>3</td>
<td>90</td>
<td>0.01</td>
</tr>
<tr>
<td>SFR14.DAT</td>
<td>RSFR14.DAT</td>
<td>14</td>
<td>32</td>
<td>296</td>
<td>3</td>
<td>90</td>
<td>0.01</td>
</tr>
<tr>
<td>SFR22.DAT</td>
<td>RSFR22.DAT</td>
<td>22</td>
<td>18</td>
<td>182</td>
<td>5</td>
<td>72</td>
<td>0.01</td>
</tr>
<tr>
<td>SFR25.DAT</td>
<td>RSFR25.DAT</td>
<td>25</td>
<td>23</td>
<td>179</td>
<td>8</td>
<td>65</td>
<td>0.01</td>
</tr>
<tr>
<td>SFR35.DAT</td>
<td>RSFR35.DAT</td>
<td>35</td>
<td>126</td>
<td>19179</td>
<td>13</td>
<td>90</td>
<td>0.05</td>
</tr>
<tr>
<td>SFR41.DAT</td>
<td>RSFR41.DAT</td>
<td>41</td>
<td>80</td>
<td>952</td>
<td>10</td>
<td>87</td>
<td>0.01</td>
</tr>
<tr>
<td>SFR45.DAT</td>
<td>RSFR45.DAT</td>
<td>45</td>
<td>84</td>
<td>918</td>
<td>12</td>
<td>85</td>
<td>0.01</td>
</tr>
<tr>
<td>SFR46.DAT</td>
<td>RSFR46.DAT</td>
<td>46</td>
<td>96</td>
<td>768</td>
<td>18</td>
<td>81</td>
<td>0.10</td>
</tr>
<tr>
<td>SFR57.DAT</td>
<td>RSFR57.DAT</td>
<td>57</td>
<td>108</td>
<td>1225</td>
<td>2</td>
<td>98</td>
<td>0.54</td>
</tr>
<tr>
<td>SFR61.DAT</td>
<td>RSFR61.DAT</td>
<td>61</td>
<td>145</td>
<td>1686</td>
<td>16</td>
<td>89</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table B.1- The result table for files of requirements transformed as segment level coverage reduction
The table B.1 legend is given in table B.2 below:

<table>
<thead>
<tr>
<th>Legend Name</th>
<th>Legend Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFN</td>
<td>Input file name for the file of requirements and associated testing sets</td>
</tr>
<tr>
<td>OFN</td>
<td>Output file name for the selected test cases</td>
</tr>
<tr>
<td>NR</td>
<td>Number of the requirements in the input file</td>
</tr>
<tr>
<td>NTC</td>
<td>Number of test cases (How many test cases)</td>
</tr>
<tr>
<td>NTCR</td>
<td>Number of test cases for all requirements</td>
</tr>
<tr>
<td>SNTC</td>
<td>Selected Number of test cases by the reduction method</td>
</tr>
<tr>
<td>PR</td>
<td>Percentage of reduction. computed as (NTC/SNTC* 100)</td>
</tr>
<tr>
<td>TIME</td>
<td>The time taken in seconds by the computer to compute the result.</td>
</tr>
</tbody>
</table>

Table B.2- This table explains the legend used in table B.1.

For a complete review of the input file, the user can review them with the supplied tool since they are provided with it.
Appendix C: Low Level Design of the Flow Graph generation and Test Coverage Display Tool

This appendix describes the code for the flow graph generation and test coverage tool. The functions are described in detail to allow the user to grasp as much as he can from the pseudo code. Again, the important functions that do most of the work are shaded to denote their importance. High level design of the tool is presented in Figures 4.2 and 4.3.

* For a definition of all variables used, please refer to the documented code supplied with this documentation.

Function `scanoe0n`
1- reads all the characters in a file till you reach end of the line
2- set the file pointer to the next line
3- return to calling function.

Function `Initialize`
1- initialize the segment connection nodes
2- initialize the segment connection edges
3- return to calling function

Function `Opengraph`
1- initialize graphics drivers to VGA display
2- Attempt to open the graph mode in VGA mode
3- if the attempt failed
3.1 - give an error message
3.2 - exit the program
4- return to calling function

Function Closing Graph
1- Close graph mode
2- Restore CRT mode
3- return to calling function

Function tg_mouseinrect( rectangle coordinates ):integer
1- get the mouse current position
2- if mouse is within rectangle return (0) to the calling function
3- else return (1) to the calling function

Function drawrectangle(rectangle coordinates)
1- if put choice = 1 then
   1.1-draw the white line effect of the rectangle
   1.2-draw the darkgray line effect of the rectangle
2- else
   2.1- draw the darkgray effect first
   2.2- draw the white effect second
3- return to calling function

Function warning( message)
1- save background for warning message window
2- display the warning window
3- display the warning message passed
4- wait for a key to be pressed
5- read the key pressed
6- return to calling function

Function echocharacter( int xposition, char ch):integer
1- prepare the character into a string
2- echo the character in the x-position
3- increment the x-position for the next character to be displayed
4- return(new x-position) to the calling function

Function question(int x, char message)
1- allocate the memory for saving the screen
2- save the screen behind the question window starting at position x
3- display the question window
4- display the message passed in the question window
5- return to the calling function

Function erasequestion(int x)
1- restore the screen to its previous status
2- free the memory allocated for saving the screen
3- return to the calling function.

Function clearinput
1- set view port to the sub window of input
2- clear the view port
3- return to the calling function

Function intro
1- open the introductory text file for reading
2- if file is not found
   2.1 - call warning("intro, file not found")
3- else
   3.1- prepare the help screen
   3.2- read the help text into the window
   3.3- get the read key for window
   3.4- if not end of file and key is not exit go to 3.1
4- return to calling function
Function DrawBoard
1- set the view port to the drawing window
2- prepare the drawing rectangles
3- fill the drawing rectangle with the white color for drawing
4- return to the calling function

Function draw buttons
1- clear the old buttons representation
2- for each button in the menu list do
   2.1- check if button is active
   2.2- if active button draw button as active
   2.3- else draw button as non active (shadowed)
3- return to the calling function

Function draw menu
1- prepare the menu for drawing buttons
2- call draw buttons for drawing the buttons
3- return to the calling function

Function showmessage(message)
1- clear the button of the drawing board
2- display the passed message in the button of the board
3- return to the calling function

Function drawname
1- clear the button of he drawing board
2- display the name of the program and version at the button
3- return to the calling function

Function preparefilelist
1- open the file filelist.txt for output
2- list all the files in the current directory to that file
3- return to the calling function

Function save directory
1- allocate memory for the size of directory screen
2- save the screen behind the directory screen
3- return to the calling function

Function Restore Directory
1- restore the screen to its previous state using memory saved image
2- free the memory allocated to save the directory screen
3- return to the calling function

Function directory list
1- open the file filelist.txt for reading
2- if file not found
   return to the calling function
3- prepare the directory window
4- read file names into the directory window
5- return to the calling function

Function Prepare Blink
1- allocate the memory for saving the screen
2- save the screen behind the blinking screen
3- set the view port for blinking in the middle of the screen
4- clear the view port
5- return to the calling function

Function blink line
1- display the text on the screen
2- wait for a time
3- erase the text on the screen
4- return to calling function

Function endBlink
1- set the view port for blinking
2- restore the screen to its previous image
3- free the memory allocated for blinking
4- Return to calling function

Function Processing(percentage processed)
1- set a rectangle equal to percentage processed
2- fill the drawn rectangle
3- return to calling function

<table>
<thead>
<tr>
<th>Function readconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1- call question window</td>
</tr>
<tr>
<td>2- call prepare file list</td>
</tr>
<tr>
<td>3- while ( not eof() )</td>
</tr>
<tr>
<td>3.1- read the character</td>
</tr>
<tr>
<td>3.2- store the character in the file name</td>
</tr>
<tr>
<td>3.3- call echo character to echo input</td>
</tr>
<tr>
<td>end while</td>
</tr>
<tr>
<td>4- open the file with the entered name</td>
</tr>
<tr>
<td>5- if file can not be opened</td>
</tr>
<tr>
<td>5.1- call warning(&quot;file not ready&quot;)</td>
</tr>
<tr>
<td>5.2- return to the calling function</td>
</tr>
<tr>
<td>6- else</td>
</tr>
<tr>
<td>6.1- while not end of the file do</td>
</tr>
<tr>
<td>6.1.1- read the segment connection into the array</td>
</tr>
<tr>
<td>6.1.2- call scaneco</td>
</tr>
<tr>
<td>end while</td>
</tr>
<tr>
<td>7- close the file</td>
</tr>
<tr>
<td>8- activate the drawing button and deactivate other buttons</td>
</tr>
</tbody>
</table>
Function insertnode(nnodel)
1- set the pointer to the head of drawing list
2- go until you reach before the null pointer
3- insert the node at the end of the list
4- let the next pointer of node points to null
5- Return to the calling function

Function findpl(int key)
1- set the pointer to the head of the predecessor list
2- do searching until null is reached or key is found
3- if not found return null
4- else return a pointer to it
5- return to the calling function

Function findnl(int key)
1- set the pointer to the head of the drawing list
2- do searching until null is reached or key is found
3- if not found return null
4- else return a pointer to it
5- return to the calling function

Function find(int pred, int list[]):int
1- found <- 1
2- search the whole list for the key
   2.1- if pred = list[key] found=0;
3- if found = 0 return(0) to the calling function
4- else return 1 to the calling function
Function countpred(int seg, int list[]): integer
1- initialize the list to no predecessors
2- count <- 1
3- initialize a pointer to the pred list
4- for every node in the predecessor list do
   4.1 - if node key = segment add to the list (increment counter)
   4.2 - go to the next predecessor
5- return to the calling function with count

Function getxx
1- minx <= 0
2- for every node that is a predecessor of the segment
   2.1- find position in the drawing list
   2.2- adjust the minimum and maximum
   2.3- add the position pointer
3- divide the position pointer by the number of predecessors
4- return the position as position pointer to the calling function

Function getdown
1- find the node that is the only predecessor of the segment
2- adjust the starting position and adjust the y-axis
3- return to the calling function

Function insertpred(pnode)
1- initialize a pointer to the head of predecessor list
2- do searching until you reach the end pointer
3- append the node at the end of the list
4- let the next pointer of the node points to null
5- return to the calling function

Function getdownleft
1- find the node that is the only predecessor of the segment
2- adjust the x-position to the left and adjust the y-axis
3- compute the new starting x-position and ending x-position
4- return to the calling function

Function getdownright
1- find the node that is the only predecessor of the segment
2- adjust the x-position to the right and adjust the y-axis
3- compute the new starting x-position and ending x-position
4- return to the calling function

Function free predecessors
1- for every node in the predecessors list
   1.1- free the memory allocated for it
   1.2- get to the next node in the list
2- return to the calling function

Function free nodes
1- for every node in the drawing list
   1.1- free the memory allocated for it
   1.2- get to the next node in the list
2- return to the calling function

Function buildlist ------> main function:
1- draw node 1 (adjust its starting x-position and y-position)
2- curr_node <- 1
3- while the last segment is not drawn
   3.1- compute the successors for the curr_node
   3.2- if successors = 1
      3.2.1- compute all predecessors
   3.2.2- if all predecessors are drawn
      3.2.2.1- call getxx to compute new position
      3.2.2.2- draw the node
1- set the view port for drawing
2- set text style for drawing
3- prepare the drawing board
4- for each node in the drawing list
   4.1- draw the node
   4.2- if node is visited the fill it with color
5- call connectlines
6- return to the calling function

Function drawzoomlines
1- draw the vertical line
2- draw the horizontal line
3- return to the calling function

Function connectlineszoom
1- for every segment in segment array
2- find the parent node position
3- find the first node connection
4- connect the parent to the connection node using zoom mode constant
5- if the segment has another node connected to it
   5.1- find the new right node connection position
   5.2- connect the parent to the connection node using zoom constant
6- return to the calling function

Function drawlistzoom
1- set the view port for drawing
2- set text style for drawing
3- prepare the drawing board
4- call drawzoomlines
4- for each node in the drawing list
   4.1- draw the node using zoom constants
   4.2- if node is visited the fill it with color
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.</td>
<td>call connectlineszoom</td>
</tr>
<tr>
<td>6.</td>
<td>return to the calling function</td>
</tr>
</tbody>
</table>

**Function refresh**

1. redraw the drawing board in white background  
2. reinitialize nodes colors to natural  
3. reinitialize edge colors to natural  
4. return to the calling function

**Function drawlist**

1. call build list to form the linked list for drawing  
2. call drawlist to draw the list in the drawing board  
3. activate all the buttons  
4. call draw buttons to redraw the active buttons  
5. return to the calling function

**Function up**

1. check the value of the y-axis  
2. if y-axis is at zero then warning("can not go up")  
3. else increment y-axis to get to the next page  
4. if in zoom mode call drawzoomlist  
5. else call drawlist  
6. return to the calling function

**Function down**

1. check the value of the y-axis  
2. if y-axis is at maximum y then warning("can not go down")  
3. else decrement y-axis to get to the next page  
4. if in zoom mode call drawzoomlist  
5. else call drawlist  
6. return to the calling function
Function left
1- check the value of the x-axis
2- if x-axis is at zero then warning("can not go left")
3- else increment x-axis to get to the next page
4- if in zoom mode call drawzoomlist
5- else call drawlist
6- return to the calling function

Function right
1- check the value of the x-axis
2- if x-axis is at maximum x then warning("can not go right")
3- else decrement x-axis to get to the next page
4- if in zoom mode call drawzoomlist
5- else call drawlist
6- return to the calling function

Function zoom
1- if already in zoom mode
   1.1- return to natural mode
   1.2- call draw list to draw the list
2- else
   2.1- set zoom flag on
   2.2- call draw zoom to draw the list
3- return to the calling function

Function findpath(int seg, int path[])
1- found <- false;
2- while not found and not the end of list do
   2.1- if path[key] = segment passed found <- true
end while
3- return to the calling function with found
Function testcover
1- call prepare file list
2- call save directory
3- Open the file for reading (directory list file)
4- call question
5- get the test cover file name
6- open the test cover file
7- if file can not be opened
   7.1- display a warning("file not found");
   7.2- go to 5.
8- erase question
9- restore directory
10- read the test cover file
11- mark the edges and nodes covered
12- if in zoom mode call draw zoom list
13- else call draw list
14- return to the calling function

Function printing
1- initialize the printer
2- zoom the drawing to two pages
3- match the drawing with the printer
4- check if printer is ready
5- if printer ready
   5.1- send the graph node by node
   5.2- send the connecting line
   5.3- if end of page go to next page
   5.4- if not end of segments go back to 5.1
   5.5- send the variable segment information
   5.6- close the printer
6- return to the calling function
Function exiting
1- clear the drawing board
2- prepare the drawing board for exiting screen
3- open the exit text file
4- read the text from the exit file to the screen
5- return to the calling function

Function buildhelpface
1- prepare the drawing board by clearing it
2- set the background color for the help option
3- return to the calling function

Function help
1- call buildhelpface
2- open the help file for reading
3- if help file not found
   3.1- warning("Help file not found");
   3.2- return to the calling function
4- else
   4.1- read the help file into the screen
   4.2- wait for the keyboard input
   4.3- if input is not exit go to 4.1
5- restore the drawing board
6- return to the calling function

Function buildinfoface
1- prepare the drawing board by clearing it
2- set the background color for the information option
3- return to the calling function

Function drawcutline(int y)
1- draw the first line at y-position
2- draw the second line at y-position
3- return to the calling function

Function information
1- open the variable information file
2- call build info face to prepare the background
3- read the file information into the screen
4- read key from the keyboard
5- if key is not exit go to 3
6- return to calling function

Function segmentinfo
1- open the variable information file
2- call build info face to prepare the background
3- read the file information into the screen
4- if information is related to segment passed then display it
5- read key from the keyboard
6- if key is not exit go to 3
6- return to calling function

Function cinfo
1- get the user choice for information display
2- if the user want whole information call function information
3- else call segment information with segment number
4- return to the calling function

```c
Function choosefunction(int choice)
{
   switch (choice)
   {
     case 1: reading(); /* choice is read */
       break;
     case 2: refresh(); /* choice is refreshing */
   }
}
```
break,
case 3: drawing(); /* choice is drawing */
        break;
case 4: up(); /* choice is getting up in the picture */
        break;
case 5: down(); /* choice is getting down in the picture */
        break;
case 6: left(); /* choice is getting left in the picture */
        break;
case 7: right(); /* choice is getting right in the picture */
        break;
case 8: zoom(); /* choice is zooming the picture */
        break;
case 9: testcover(); /* choice is test coverage path */
        break;
case 10: printing(); /* choice is printing the graph */
        break;
case 11: information(); /* this choice provides information about the segment variables */
        break;
case 12: choice = exiting(choice); /* choice is to exit the program */
        break;
case 13: help(); /* choice is to get help */
        break;
};
2-return to the calling function

Function getchoice
1- wait for either a mouse strike or a keyboard
2- if mouse was strike get mouse position
3- else get keyboard choice
4- activate the corresponding choice by drawing its button pressed
5- call choose function
6- return to the calling function

Function password
1- display the password window
2- get the password
3- check the validity of the password
4- if password is valid then return to the calling function
5- else go to 2
6- return to the calling function

Function main
1- call opengraph
2- call drawname
3- call drawboard
4- call draw menu
5- call password
6- call intro
7- call get choice
8- call closing graph
9- return to DOS
Appendix D : Further Examples on the Application of the Flow Graph Generation and Test Coverage Display Tool.

This appendix contains several examples for input files that can be supplied to the flow graph generation and test traversal tool. Also, in addition to each input file there is the output file associated with it as a printout of the graph generated by the tool itself. The organization of the appendix is as follow:

1. The general outline of an input file
2. Some input files as associated with their output file

As for the first part of the appendix. The general outline of the connectivity file is given in Figure D.1.

<table>
<thead>
<tr>
<th>FUNCTION NAME</th>
<th>NUMBER OF SEGMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: 0</td>
<td>/* a needed line for starting segment */</td>
</tr>
<tr>
<td>segment #</td>
<td>segment#</td>
</tr>
<tr>
<td>Last segment#</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure D.1. Connectivity input format file.

The first line of the file should contains first the function name for which the connectivity is generated. Note that the function name should not exceed the limit of 32 characters long. After the function name on the first line, we have the number of
segments i.e. the last segment number. For example if the connectivity have 14 segment then the last segment number would be 13 since 0 is counted as a segment number even if it is not drawn by the tool. The second line of the file contains the standard connection of segment 1 and only. The second segment is given the value 0 to denote no connection. The 0 value would be used anywhere in the file to denote no connection. The third line of the file, a line repeated as many times as the number of segments each time incrementing the segment number on the left by 1 is used to denote the connection of that segment to other segments. The other segment in the line should not have a number that is lower than the number on the left since it will creates a loop, a case not found in the connectivity matrix. If such a case happens to exist the tool will give an appropriate error message: “Misplaced Segment. Segment Number Error” and skips to the whole file. The line should not contain both right segments to be 0 or else it won’t be connected to other segments of the file. The tool in this case won’t give an error message and it will continue drawing normally as if nothing has happened. However, when the user draws the connectivity he will sees a disconnected graph. The last line of the file is a special case line, where the last segment number is used on the left and the other two segment numbers are 0’s to denote that this line is the end line and there is no interesting line after it. Note that the user is allowed to put comment in its input file after the end of each line on condition that it is separated from the last field by at least one space and can add as many lines after the end segment line as he desires to explain the file itself. The second part of the appendix contains some examples with their associated output. The list of files is given in Table D.1.
<table>
<thead>
<tr>
<th></th>
<th>MAIN</th>
<th>Connectivity+Variables file</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>TEST1</td>
<td>Connectivity</td>
</tr>
<tr>
<td>3</td>
<td>TEST2</td>
<td>Connectivity</td>
</tr>
<tr>
<td>4</td>
<td>TEST3</td>
<td>Connectivity</td>
</tr>
<tr>
<td>5</td>
<td>TEST4</td>
<td>Connectivity</td>
</tr>
<tr>
<td>6</td>
<td>TEST5</td>
<td>NONE</td>
</tr>
<tr>
<td>7</td>
<td>TEST6</td>
<td>NONE</td>
</tr>
<tr>
<td>8</td>
<td>TEST7</td>
<td>Connectivity</td>
</tr>
<tr>
<td>9</td>
<td>TEST8</td>
<td>Connectivity</td>
</tr>
</tbody>
</table>

Table D.1. List of test files as occurring in the appendix.
References


