Structural testing techniques for the selective revalidation of software

Jean Zoren Werner Hartmann

University of Durham

School of Engineering and Computer Science
(Computer Science)

Structural Testing Techniques for the Selective Revalidation of Software

Jean Zoren Werner Hartmann

Ph.D.

1992

Abstract

The research in this thesis addresses the subject of regression testing. Emphasis is placed on developing a technique for selective revalidation which can be used during software maintenance to analyse and retest only those parts of the program affected by changes. In response to proposed program modifications, the technique assists the maintenance programmer in assessing the extent of the program alterations, in selecting a representative set of test cases to rerun, and in identifying any test cases in the test suite which are no longer required because of the program changes.

The proposed technique involves the application of code analysis techniques and operations research. Code analysis techniques are described which derive information about the structure of a program and are used to determine the impact of any modifications on the existing program code. Methods adopted from operations research are then used to select an optimal set of regression tests and to identify any redundant test cases. These methods enable software, which has been validated using a variety of structural testing techniques, to be retested.

The development of a prototype tool suite, which can be used to realise the technique for selective revalidation, is described. In particular, the interface between the prototype and existing regression testing tools is discussed. Moreover, the effectiveness of the technique is demonstrated by means of a case study and the results are compared with traditional regression testing strategies and other selective revalidation techniques described in this thesis.

Acknowledgements

This thesis is dedicated to Rosemarie. I would like to thank her for the love, support, and encouragement that she has given me over the years.

I wish to thank Dr. David J. Robson for agreeing to be my supervisor, and for the invaluable advice and encouragement that he has given me.

I would also like to thank British Telecom Research Laboratories, Martlesham Heath, Ipswich, United Kingdom for their financial support and, in particular, Mr. Stuart Birchall and Mr. Colin Archibald for their guidance and support during my numerous visits to the Laboratories.

I wish to express my sincere gratitude to Professor Keith Bennett and the members of the Centre for Software Maintenance. In particular, I would like to thank Mr. Chris Turner and Dr. Gerardo Canfora for their comments on the thesis.

This Ph.D. thesis has been produced using an Apple® Macintosh™ using the Microsoft® Word 5.0 wordprocessing package and the Claris MacDraw Pro™ graphics package.

® Apple is a registered trademark.

™ Macintosh is a registered trademark of Apple Computer Incorporated.

® Microsoft is a registered trademark of Microsoft Corporation.

™ MacDraw Pro is a registered trademark of Claris Corporation.

Copyright

Declaration

The work contained in this thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, and represents an independent contribution by the author. None of the work has been previously submitted by the author for a degree at this or any other University.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose of the Research

A technique for **selective revalidation**[1] [101] is described, which can be applied to the **regression testing** of programs during **software maintenance**. This technique is intended to help maintenance programmers analyse and retest their software in a systematic and efficient manner. The proposed technique, together with the supporting software tools, can lead to a reduction of resources required for regression testing and can improve the confidence of maintenance programmers by ensuring that their modifications have been adequately tested.

1.2 Motivation

Perhaps the most frequently quoted aphorism in the field of computer science is expressed by Dijkstra [58]:

> *"Program testing can be used to show the presence of bugs, but never their absence"*

Many theoreticians and practitioners agree with the substance of this observation and use it to justify their research into formal methods to assert the **correctness** of a program. However, there are several reasons why not to rely upon formal approaches alone: their inability to capture the non-functional requirements of software, such as its performance and usability; their implicit assumptions concerning the correctness of supporting software; the current lack of automated tools; and in most cases, the sheer size of the problems being addressed. The latter reason alone leaves sufficient scope for unstated assumptions, or simple reasoning errors, in the derivation of proofs.

[1] In this thesis, words in **bold typeface** are defined in the Glossary given in Appendix A.

Until such issues have been resolved, **software testing** [1, 52] remains a viable and effective technique for complementing formal methods during **software development**. Together, the two approaches ensure the correct operational behaviour of a system. However, this premise does not hold for software maintenance where it appears that testing *alone* is responsible for ensuring the correctness of software which may not have been formally specified or adequately documented. It is therefore essential that a systematic and efficient framework for maintenance testing is developed.

Software maintenance represents the most costly phase of the **software lifecycle**, and accounts for between 40%-80% of the total programming effort [163, 214]. Two important factors contributing to the high costs are the analysis and retesting of the modified software. Regression testing [282] is the testing strategy applied to ensure that no adverse side-effects have been introduced into the modified software, and that the modified software still complies with its requirements. However, the problems and uncertainty concerning which parts of the software are affected by the changes, and how thoroughly these parts need to be retested, have not yet been adequately resolved [98]. Different strategies have been developed in an attempt to solve these problems; each of them possesses some benefits, yet none can provide an effective and reliable solution. Therefore, it is the goal of the research described in this thesis to produce a systematic and efficient technique for regression testing.

1.3 Objectives of the Research

1.3.1 Assumptions

The work in this thesis is concerned with the development of a technique for selective revalidation. As such, the work is influenced by a number of important factors relating to the area of application, the quality of the existing test suite, and certain aspects of program analysis. The special requirements needed to consider some of these factors will not be addressed in this thesis.

Area of Application

Software maintenance activities may be classified into **corrective maintenance**, **preventive maintenance**, **adaptive maintenance**, and **perfective maintenance** [11, 214]. Corrective maintenance is aimed at rectifying program errors revealed during testing or operational use, and involves making corrections to the program code. In the case of preventive maintenance, alterations are made to the program code with the aim of improving the existing program so as to ease future maintenance. With adaptive maintenance, the program design and code are modified in order to reflect alterations in the software's operating environment. Perfective maintenance describes the changes made to the program specification, design and code in order to enhance the functionality of the software. These enhancements usually reflect user demands for new system capabilities, and result in the inclusion of additional system features, the deletion of old features, or the modification of existing ones. Table 1.1 summarises how the various maintenance activities affect a program's attributes, namely its specification, design and implementation.

	Corrective Maintenance	Preventive Maintenance	Adaptive Maintenance	Perfective Maintenance
Specification				•
Design			•	•
Implementation	•	•	•	•

Table 1.1 : Maintenance Activities and Affected Program Attributes[2]

The technique, described in this thesis, is discussed in the context of software maintenance. As such, it concentrates upon the analysis and retesting of modifications made to the program's implementation and is therefore relevant to every type of maintenance activity.

[2] Corrective maintenance and preventive maintenance may affect the low-level design requirements of a program.

Quality of Test Suite

Empirical research indicates that using a specific type of testing strategy alone cannot detect all errors in a program [116]. Thus, test cases are often created using a combination of techniques based on **functional testing** and **structural testing**. This leads to the concept of *grey-box testing* [267]. The use of functional testing, for example, may result in only 40%-60% of the program code being executed [1, 107]. Structural testing, however, can help to expose and alleviate potential problems, such as unreachable or island code [52], by identifying and exercising the remaining, untested code.

The aim of functional and structural testing techniques is to satisfy their respective **test coverage criteria**. The quality of a test suite is therefore measured by the degree to which these criteria are fulfilled. The application of the selective revalidation technique, described in this thesis, requires the existence of a high quality test suite which satisfies its criteria. However, the test suite quality is assessed and maintained only on the basis of the *structural* test coverage criteria.

Incremental Code Analysis

Program analysis forms an important aspect of the technique described in this thesis. In particular, **code analysis** is used to derive information about the structure of the program code, determine the dependencies between code **entities** and assess the impact of any proposed code changes by way of these entities. This is achieved by means of an *exhaustive* analysis of the program code. While such analysis can provide sufficient information about the existing program structure, it is inadequate for assessing its modified structure. Therefore, the technique assumes the existence of *incremental* code analysis techniques [224, 225, 226] to update the existing program structure and dependencies, in response to a program change, without needing to reanalyse the entire program.

1.3.2 Goals

In this thesis, a technique for selective revalidation is presented which is characterised by two important objectives: to increase maintenance programmers' confidence in the correctness of their modified software, and to reduce the overall cost of regression testing. In order to achieve these goals, ways must be found to systematically analyse the impact of any proposed program modifications and to efficiently select and update test cases in the test suite.

Initially, the technique must acquire information relating to the structure of the program code. This is accomplished by examining the software using code analysis techniques. The resulting dependencies between code statements are depicted by a graphical representation of the program. Techniques for dependency analysis have been successfully developed for a variety of programming languages [21, 38, 135]. In this thesis, however, the techniques are directed at the analysis of programs written in the C programming language [137].

The application of code analysis techniques also enables the **test requirements** for a program to be determined. The test requirements discussed in this thesis are dependent upon the type of *structural* testing technique being used and its corresponding test coverage criterion. During testing, an association is formed between each test requirement and the test cases exercising it; thus, a **testing history** emerges. This testing history later forms the basis for selecting and updating of test cases in the test suite.

In response to a proposed program modification, it is necessary to determine those parts of the program which are directly and indirectly affected by the changes. Therefore, a technique for change analysis is developed, which uses the graphical representation of the program established during code analysis in order to trace existing program dependencies from the point of modification and determine a set of affected program statements. These statements are subsequently used to identify affected test requirements.

Based on these test requirements and the set of test cases implicated by them, methods adopted from **operations research** can be used to select an optimal set of regression tests. After a modification has been implemented, the regression tests are rerun and the test suite is updated. Operations research methods are then applied again in order to determine any redundant tests.

When dealing with large and complex programs, time-consuming activities, such as code analysis and test suite management, need to be automated. Therefore, a prototype suite of tools is developed to provide support for these tasks, and this thesis describes the ways in which the tools are used to realise the technique for selective revalidation.

1.3.3 Anticipated Benefits

Given the time constraints, it is impossible to study all the applications of a technique for selective revalidation in this thesis. Some of the anticipated benefits of this work are briefly described below. It is hoped that some of these can be pursued in future research.

Path Selection Strategy

In software testing, a significant amount of time and effort is spent in analysing the program code in order to create suitable test data with which to exercise it. In structural testing, the development of an optimal path selection strategy [164, 212, 251] represents an important issue. Such a strategy can guide users in the establishment of an optimal set of test cases, which traverse paths through the program code, such that every test requirement is exercised at least once.

The technique for selective revalidation, described in this thesis, can be adapted for use as an optimal path selection strategy as it performs a similar task in the context of regression testing. It assists in the selection of an optimal set of test cases from an existing test suite instead of helping with the creation of a new set of tests. More

importantly, however, the technique concentrates on selecting a subset of test cases from the existing test suite which will traverse only those test requirements affected by the proposed modifications.

Revalidation

During software development, specification-based testing ensures that a program is assessed from an external point of view. Test cases are created to exercise the program's functionality and their execution is traced by means of a *feature-test matrix* [109]. In a similar manner, the program's design can be validated such that the conditions of each design function are related to a set of test cases in a *condition-test matrix* [37, 177, 223]. Finally, testing of the implementation involves the validation of individual, or a collection of, program **modules.** In this case, a test matrix is created in which different structural attributes of the program are exercised by a set of test cases.

During maintenance, however, revalidation is usually restricted to the analysis and retesting of the program code [101]. Few techniques are available to ensure that the corresponding changes to the program specification [151] or design [18] are tested. The development of a technique for selective revalidation, which is independent of any testing strategy, can provide a more consistent approach to regression testing. Changes at a given level can now be validated at all lower levels. For example, in the case of perfective maintenance, the same technique can be used to analyse and retest a change to the program's functionality which requires certain specification-based tests, as well as design-based and code-based tests, to be selected.

Metrics

Predicting the cost of regression testing is a difficult task. A change, which appears simple, may have a much larger impact on the software than originally anticipated, and thus requires extensive retesting. As a result, the resources required for regression testing

are often underestimated and project schedules are jeopardised. If, however, the factors affecting the cost of regression testing could be considered as part of the selective revalidation process, then maintenance programmers could more accurately predict the level of resources required to validate any program changes *before* actually making them. This, in turn, would have a significant influence on their choice of change implementation. It would also allow them to judge which, of possibly several, program changes could be implemented using the existing resources.

The technique described in this thesis is able to consider various cost factors as part of the selective revalidation process. For example, the *test execution costs* and *result analysis costs* [159], which represent the time and effort required to load the test cases, execute them and analyse their output can be considered. Therefore, in response to a change proposal, the selective revalidation technique can not only determine how many of the test cases need to be rerun, but also estimate the level of resources required to rerun them. As a result, it may be possible to define a relationship between a proposed program change and the extent of required retesting of the program. This relationship, along with other factors, such as program understandability [274] and modifiability [275], could then be used to define a maintainability metric [34, 86, 161].

1.4 Thesis Structure

Chapter 2 describes different testing phases and strategies used during software development, with emphasis being placed on structural testing techniques. Chapters 3-4 introduce the subject matter of this thesis. In Chapter 3, the role of testing during software maintenance is described, its problems are examined and the subject of selective revalidation is defined. For Chapter 4, existing techniques for selective revalidation are reviewed.

Chapters 5-6 describe the two aspects of a technique for selective revalidation. In Chapter 5, code analysis techniques are developed to systematically analyse the program

dependencies and assess the impact of proposed program changes. In Chapter 6, the problems of test suite management are addressed, with techniques adopted from operations research being used to define and efficiently solve them.

The development of a tool suite, which realises the technique for selective revalidation, is presented in Chapter 7. Brief descriptions are given concerning its design and the functionality of its components. In particular, the interface between the protoype and existing regression testing tools is discussed. Chapter 8 describes a case study in which the effectiveness of the technique for selective revalidation is analysed. Comparisons are then made with traditional regression testing strategies and other selective revalidation techniques described in this thesis. Concluding remarks, and future research, are outlined in Chapter 9.

Chapter 2

Testing During Development

2.1 Introduction

The software lifecycle describes the period of time which begins with the conception of the software and ends when it is no longer of use. It consists of two main phases: the development phase, which includes the requirements definition, specification, design, implementation and testing, and the operations and maintenance phase. Figure 2.1 illustrates the different activities within the software lifecycle [230].

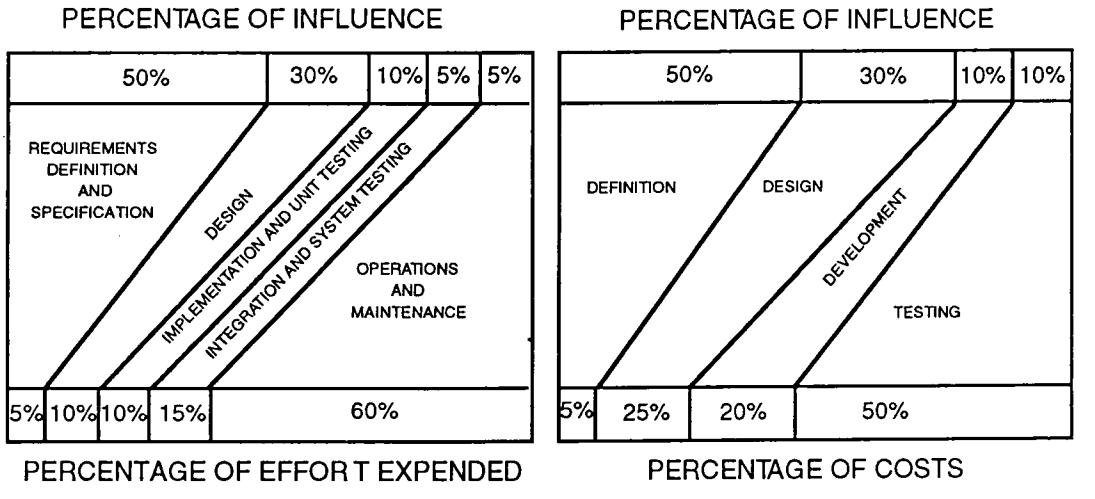


Figure 2.1 : Activities Within the Software Lifecycle

As seen in Figure 2.1, software testing represents one of the costliest, but also least influential, phases within the lifecycle; it accounts for as much as 25% of the overall programming effort and 50% of the costs [23, 216]. A need for testing arises from an inability to guarantee that tasks performed earlier in the software lifecycle have been satisfactorily completed; testing helps to increase confidence in the correctness of a piece of software by discovering any errors of omission and commission. Thus, testing forms an attempt to assess the *quality* of the completed work and to gain confidence in the

software [57]. As such, testing is regarded as part of Verification and Validation (V&V) process in which **verification** ensures the correctness of the software during each phase of its development and **validation** compares the software with its requirements [2, 249, 250]. The opportunity is taken here to discuss the testing activities occurring as part of software development.

2.2 Testing Activities

2.2.1 Testing Phases

The testing activities, which accompany the requirements definition and specification, help to ensure the adequacy of both phases with respect to their completeness, correctness, and consistency [54, 185]. They include the adoption of a general testing strategy, the selection of testing techniques, and the formulation of specific evaluation criteria. The resulting test plan provides the testing schedule, which contains the test case specifications and descriptions. Using scenarios of both expected *and* unexpected system usage, test data and its anticipated results are generated. These test cases now form the core of the final test suite. A test analysis summarises and documents the test results and their findings.

It is claimed that as many as 80% of errors detected in the software lifecycle arise from problems occurring during program design [248]. Testing activities need to exercise the functionality introduced during the design process, as well as the structure of a system. Their purpose is to show that the detailed design is consistent and complete. Apart from generating test cases, which are required for the implementation phase, the design itself must be examined and analysed for errors which may include missing test cases, faulty logic, interface mismatches and data structure inconsistencies.

During implementation, developers are presented with the program design from which they code the individual program modules. **Unit testing** ensures that each module correctly implements its design and is ready to be integrated into the system. This type of testing, which is also referred to as *intra*procedural testing, is performed in isolation from other modules and relies on the use of **test drivers** and **test stubs** [187].

After unit testing is concluded, the individual code modules are collated, integrated into a complete system and validated. The objective of **integration testing** is to ensure that all program modules interact and interface correctly with each other [20, 87]. This type of testing, across module boundaries, is also known as *inter*procedural testing. Two common testing strategies used during integration testing are: **bottom-up testing** and **top-down testing** [187].

System testing is concerned with validating the entire software system, or a major part of it, with respect to requirements such as performance, security and recovery capabilities. At this stage, software testing is concerned with revealing errors which might have been missed during the previous unit testing or integration testing. Until this stage, all testing activities are carried out using test data provided by the system developers.

Acceptance testing is the activity during which the entire system is evaluated in an operational environment using typical user data. It often demonstrates errors in the requirements of the system. In the case of erroneous software, for example, the requirements may not reflect the actual facilities and performance expected of the system. However, acceptance testing will highlight the problem by demonstrating that the system does not operate as was envisaged. Although acceptance testing is usually considered as part of system testing, it actually occurs *after* system testing and immediately *before* delivery of the software to customers.

The majority of activities associated with development testing lend themselves to automation. Software tools, for example, are available to support the generation of test data [13, 74, 200], its execution [44, 43, 182] and its documentation and management [166].

2.2.2 Testing Strategies

Software development includes a large number of testing strategies which, for all their apparent diversity, cluster or separate according to their underlying principles. However, two distinct classes emerge within the field of software testing: **static analysis** and **dynamic testing** as well as functional testing and structural testing. The degree to which the techniques use either static analysis *versus* dynamic testing, or functional testing *versus* structural testing, provides one possible way for classifying them[1].

In this thesis, emphasis is placed on describing the latter classification, functional testing versus structural testing. Functional testing is considered relevant, because it aids in understanding certain techniques for selective revalidation which are examined later on in this thesis. Structural testing has influenced the research direction of the thesis and forms the basis of a technique for selective revalidation. However, for completeness, both static analysis and dynamic testing are also mentioned.

Static Analysis and Dynamic Testing

Methods for static analysis attempt to analyse the software and detect errors without actually executing it. To achieve this, information is extracted concerning the structure of the program and represented by a program graph[2]. Dynamic testing techniques specify various test coverage criteria based on test requirements derived using the program graph.

---

[1] This type of classification is often complicated by the fact that some techniques may belong to several different categories.

[2] Graph terminology is defined in Chapter 5.

Test cases are then executed in order to exercise these test requirements and, subsequently, to satisfy the test coverage criteria. An indication of the achieved test coverage is given; if the test coverage has not been satisfied, then additional test cases are submitted. This process is repeated until the chosen test coverage criterion is fulfilled. Although errors may still exist after testing, the required degree of reliability in the code's correctness has been achieved according to the test coverage criteria which were used to validate it.

Functional Testing

Functional testing [119, 120, 198] considers the program's specification as a basis for testing. The program under consideration is effectively treated as a *black box*, where the contents are hidden and no consideration is given as to *how* the program performs the specified functionality. Thus, functional testing is also referred to as *black-box testing* [125, 222].

The most obvious, and generally intractable, functional testing technique is exhaustive testing. This form of testing is usually impractical as the range of input values for a program would result in an excessive number of test cases needing to be created. Therefore, the objective of functional testing is to maximise the number of errors detected, while selecting a finite set of test cases whose values are representative of the input domain of the program. Techniques, such as **cause-effect graphing** [16] and **revealing subdomains** [258], have been developed to assist in this selection procedure.

Cause-Effect Graphing

Cause-effect graphing, which was first defined by Elmendorf [60] and adopted for testing purposes by Myers [188], identifies individual system functions to be tested. For

each function, all significant causes and effects on the function's behaviour are determined. A graph is then constructed, which relates combinations of the causes to the effects they produce. Test cases are defined for each effect by considering all combinations of causes which produce that effect.

Although careful use of the cause-effect graph can produce effective tests, the technique is difficult to apply in practice. In particular, the cause-effect graph can become very complex when a function has a large number of causes. To reduce the complexity of the corresponding graph, intermediate nodes are added to represent logical combinations of several causes. However, the choice of appropriate intermediate nodes is not always obvious. Other problems related to cause-effect graphing include the difficulty of verifying its correctness, and the updating of the internal graph structure while the specification is being modified. In fact, the transformation of a specification into a set of cause-effect graphs only ensures that one complex representation is replaced by another.

Revealing Subdomains

Weyuker-Ostrand [258] propose the theory of revealing subdomains which represents a combined functional and structural testing strategy to derive a *partition* for a function's input domain into revealing subdomains. A revealing subdomain contains elements which are either all processed correctly or incorrectly. Once such a subdomain has been identified, executing the program on a single one of its elements is sufficient to test the entire subdomain. The existence of a correctly processed input from a subdomain, together with an indication that the subdomain is revealing, are equivalent to proving the program's correctness for all inputs in the subdomain. However, given the difficulty, in general, of proving program correctness, it should not be assumed that revealing subdomains can be easily produced.

The technique attempts to construct revealing subdomains by identifying the most likely places for errors to occur. A *problem partition* is created from the specification by examining classes of inputs which should be treated as equivalent by the program. A *path partition* is then created whose equivalence classes contain inputs that actually are treated the same way by the program. The partition to be used for functional testing, known as the *testing partition*, is then created by intersecting the problem and path partitions. This results in a set of equivalence classes, whose elements are equal and are treated as such by the program. During testing, therefore, test cases are generated by choosing one element from each of the testing partition's classes.

The main difficulty lies in the implementation of the revealing subdomain method. No formal, or systematic, guidelines exist for creating a problem partition. However, the category-partition method proposed by Ostrand-Balcer [200] may provide a partial solution to this problem. It describes a systematic approach to creating test sets on the basis of the specification and could conceivably help in creating such a problem partition.

Other functional testing techniques include *equivalence partitioning* [188, 219, 220, 231], *boundary-value analysis* [188] and *condition tables* [83]. These techniques are mentioned for completeness, but will not be further considered in this thesis.

2.3 Structural Testing Techniques

With structural testing [197, 257], or *white-box testing*, the testing effort is not directed at assessing the program's functionality, but rather at identifying errors in its actual implementation. Thus, test data is developed and executed on the program code.

The aim of detecting *data-flow anomalies* [123, 127, 199, 244] is to highlight any anomalous circumstances within the program code which may indicate errors. For example, the repeated definition of a program variable without any intervening reference,

a variable reference which is undefined, or undefining a variable, which has not been referenced since its last definition, can indicate possible errors.

For *symbolic execution* [39, 118, 138], three inputs are accepted: a program to be interpreted, the symbolic input for a program and a program path to be followed. Correspondingly, two outputs are produced in the form of the symbolic output, which describes the computation of the selected program path, and the path condition for that path. While the symbolic output is used to prove the program correctness with respect to its specification, the path condition assists in generating test data to exercise the desired path. The symbolic inputs refer to variable names instead of actual values, and the program outputs are expressed as logical or mathematical expressions involving these names.

With *domain testing* [40, 262, 263], test data is selected on, or near, the boundary of the path domain, which represents the set of values which cause a program path to be executed. During *mutation testing* [56, 139, 195, 269], errors, or *mutants*, are purposely planted in a program which is considered to be error-free. The program is then executed with the existing test data, and its adequacy is measured in terms of the number of mutants which are detected by the test cases.

However, this thesis concentrates on examining structural testing techniques [122, 194], or path selection criteria [41], which guide the selection of test cases for traversing paths through the program code and ensure that the program is adequately tested. Two types of techniques, namely **control-flow testing** and **data-flow testing**, are described, which rely upon the control structure and data structure of a program as the basis for developing test cases.

2.3.1 Control-Flow Testing

Control-flow testing involves the testing of program statements, branches, and paths throughout the program. Test coverage criteria, such as *Test Effectiveness Ratios* (TER_n) [106], have been defined to provide increasing degrees of test coverage which become progressively more difficult to achieve as n increases. For example, $Ter_1 = 1$ and $Ter_2 = 1$ represent the situation in which every program statement and program branch has been tested, respectively. Thus, the two test coverage criteria are related via the subsumption relation, $Ter_2 = 1 \Rightarrow Ter_1 = 1$.

Executing each program statement at least once during testing constitutes **statement testing**. It represents the most basic structural testing technique and is regarded as a minimum testing requirement. Statement testing is often associated with *segment testing*, or *DD-path testing* [181, 267], in which a linear sequence of statements is tested instead of just a single program statement. However, the problem with statement testing is that achieving satisfactory test coverage does not ensure that each transfer of control (branch) is exercised. Consider the following fragment of program code. (Note that the line numbers in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
1:   program example
2:   begin
3:     read(x,y);
4:     if (x > 0) then
5:        begin
6:          write(x);
7:        end
8:     write(y);
9:   end
```

Figure 2.2 : Branch Testing - An Example

Test data can be generated to execute these program statements. For example, if variables x and y are assigned values 1 and 2, respectively, all statements in the program would be traversed. However, the FALSE outcome of the predicate expression `(x > 0)` in the conditional statement (if-statement) at line 4 remains unexercised.

With **branch testing**, both the correct and incorrect outcomes of the predicate expression contained in a conditional statement need to be tested. In the above example, therefore, a further test case with inputs: x = -1 and y = 2, would need to be generated and executed. Tools for branch testing annotate the predicate expression of every conditional statement in order to evaluate its overall outcome during the execution of a test case [8, 24, 29, 278]. However, a limitation of branch testing is that predicate expressions, which consist of compound conditions, are not adequately tested.

Consider the following fragment of program code. (Note that the line numbers in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
1:   program example
2:   begin
3:     read(x,y);
4:     if (x > 0 AND y < 0) then
5:        begin
6:          write(x);
7:        end
8:     write(y);
9:   end
```

Figure 2.3 : Condition Testing - An Example

Test data can be generated to execute both program branches. For example, if variables x and y are assigned values 1 and -1, respectively, then the overall correct outcome of the conditional statement is executed. In contrast, a test case with variables x

and y being assigned values -1 and -1, respectively, would cause the overall incorrect outcome to be executed. For these cases, however, the condition ($y \geq 0$) of the predicate expression (`x > 0 AND y < 0`) is never tested.

A predicate expression contains either a *simple condition* or a *compound condition.* A simple condition is represented by a logical variable or a relational expression of the form: <arithmetic expression> <relational operator> <arithmetic expression>, where the relational operator is one of $<, >, \leq, \geq, \neq, =$. A compound condition is composed of two, or more, simple conditions, logical operators (`OR`, `AND`, `NOT`), and parentheses. A compound condition without relational expressions is referred to as a *logical expression.*

With **conditional testing** [188], test cases must be generated to exercise the compound condition, and every simple condition within it, at least once. Therefore, conditional testing is more difficult to achieve than branch testing. Its purpose is to guarantee the detection of logical and relational operator errors in a condition, provided that all logical variables and relational expressions in the condition occur only once and have no common variables [241, 242]. The notion of condition-constraints is introduced for specifying the outcome of a logical variable (`TRUE`, `FALSE`) and a relational expression (>, <, =). Consider the above example in which the constraint set would be given by {(`TRUE`, `TRUE`), (`FALSE`, `TRUE`), (`TRUE`, `FALSE`)}; only the first two condition-constraints are satisfied and, therefore, a third test case with inputs: x = 1 and y = 2, must be created to satisfy the test coverage criterion. If the compound condition is incorrect due to a logical operator error, then at least one of the three test cases will produce an incorrect outcome. A similar approach is taken for the testing of relational expressions, where condition-constraints are created using suitable combinations of relational operators. Tools for conditional testing [240] decompose the compound condition within each predicate expression into a set of simple conditions and nested conditional statements before testing it.

The testing of a **Linear Code Sequence and Jump** (LCSAJ) has been shown to be more effective, but also more difficult to achieve, than either statement testing or branch testing [105]. An LCSAJ is defined in terms of the program text and represents a subpath through the program code. It consists of a sequence of consecutive statements in the program text, starting at an entry point, or after a control-flow jump, and terminating with a jump or at an exit point. The entry and exit points include the beginning and end of a program. Consider the following fragment of program code. (Note that the line numbers and text in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
START      1:   program example
           2:   begin
           3:     read(x,y,z);
START      4:     while (
           5:              y > 5
FINISH     6:            ) do
           7:       begin
           8:         z = z + 1;
FINISH     9:         x = x + 2;
           10:      end
START      11:    b = 1;
           12:    if (
           13:         x > 10
FINISH     14:      ) then
           15:      begin
           16:        b = b + 1;
           17:      end
FINISH     18:    else
START      19:      begin
           20:        c = c + 3;
           21:      end
FINISH     22:    write(b,c,x);
           23: end
```

Figure 2.4 : Testing of LCSAJs - An Example

A total of six LCSAJs (`1:6:11, 1:9:4, 11:14:19, 11:18:22, 4:6:11, 19:22:23`) can be identified from the program. Each LCSAJ is represented by a start statement, an end statement, and the target statement for the control-flow jump. In this case, the Test Effectiveness Ratio measures the number of LCSAJs exercised by the test data and is expressed by Ter_3. By combining LCSAJs, progressively longer subpaths may be tested and it becomes progressively more difficult to satisfy the test coverage criterion. For example, the combination of two LCSAJs, `1:6:11` and `11:14:19`, would constitute the Test Effectiveness Ratio, Ter_4, while the combination of three LCSAJs, `1:6:11`, `11:14:19` and `19:22:23` would represent Ter_5. Therefore, Test Effectiveness Ratios, Ter_{n+2}, can be applied, where n specifies the number of concatenated LCSAJs required to exercise the subpaths through the program.

However, the most stringent control-flow testing technique and test coverage criterion is **path testing** [268], which requires the execution of every possible path through a program. By experimentation, Howden [117] showed that path testing is the single best method for exposing errors in a program. However, due to the presence of program loops, the number of paths through a program can be extremely large (possibly infinite), even for the most trivial programs [188, 192].

Boundary-interior path testing [116] is a restricted version of path testing in which the potentially infinite number of paths are partitioned into a finite set of equivalent paths based on the program loop characteristics. Testing is then conducted using a few representative paths from each partition. In boundary-interior testing, two classes of paths are considered from each group of similar paths with respect to each loop. Paths in the first class enter the loop, but do not iterate it (boundary tests), while paths in the second class iterate the loop at least once (interior tests). Among the boundary tests, those are selected which follow different paths within the loop. Among the interior tests, those are selected which follow different paths through the first iteration of the loop.

Consider the following fragment of program code. (Note that the line numbers in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
1:   program example
2:   begin
3:     read(x,y);
4:     while (x > 0) do
5:        begin
6:          if (y > 3) then
7:            begin
8:               y = y - 1;
9:            end
10:         else
11:           begin
12:              y = y + 2;
13:           end
14:         x = x - 1;
15:       end
16:     write(y);
17:  end
```

Figure 2.5 : Boundary-Interior Path Testing - An Example

Two boundary tests must be developed to exercise the `while` loop at statement 4 and then immediately exit the loop. Thus, a test case is created to exercise each branch of the `if-then-else` statement; test cases with inputs: x = 1 and y = 2, and x = 1 and y = 4 would fulfill these conditions. In addition, four interior tests need to be created, all of which execute the body of the loop for a second time. Test cases with inputs: x = 2 and y = 5, x = 2 and y = -3 , x = 2 and y = 4, and x = 2 and y = 3 cause all four possible permutations of the `if-then-else` statement to be exercised; they have the following outcomes: (TRUE, TRUE), (FALSE, FALSE), (TRUE, FALSE), and (FALSE, TRUE). According to the testing technique, interior test paths may exit the loop or iterate it an

additional number of times taking any of the branches after the second execution of the body of the loop.

However, path testing is also faced with another problem: path infeasibility. An infeasible path is impossible to execute with test data due to contradictions in the predicate expressions of some conditional statements which may lie on the same program path. This problem is highlighted in a study conducted by Hedley-Hennell [104] where a number of sample programs, containing as many as one thousand paths, were found to contain only eighteen feasible paths. Although current path testing strategies and tools [182, 265] cannot identify path infeasibility and need manual guidance, research into the subject is ongoing [35, 53, 117, 268].

2.3.2 Data-Flow Testing

The first data-flow testing criterion, known as *reach coverage*, was proposed by Herman [108]. It requires the testing of a program path between a variable definition and its corresponding use. This type of interaction is also referred to as a *2-dr interaction* [193]. The main shortcoming of this strategy is that it does not guarantee that branch testing is achieved. The *required pairs* strategy developed by Ntafos [193] ensures that at least one required pair is produced for each 2-dr interaction. If a 2-dr interaction involves a reference within a branch predicate, a required pair for each outcome of that predicate is produced. The *required k-tuples* strategy [193] is an extension of the required pairs technique in which sequences of 2-dr interactions, or *k-dr interactions*, are tested.

Laski-Korel [147, 148] proposed two data-flow testing strategies, the first of which is equivalent to the reach coverage criterion. The second testing technique requires the *elementary data context* of every program statement to be tested at least once. The elementary data context of a statement includes those variables definitions which reach the given statement and correspond to referenced variables in it. A more extensive strategy

called *ordered data context* requires that the definitions in each elementary data context are tested along all subpaths leading to their corresponding uses. In order to support these data-flow testing strategies, tools such as STAD [141, 149] have been developed.

However, the most recent work in data-flow testing has been conducted by Weyuker [72, 218] who proposes a number of test coverage criteria or *test adequacy criteria* [41]. The effectiveness of these data-flow testing criteria has been evaluated in a number of empirical studies [259, 260, 261] using tools such as ASSET [70, 71]. With this type of data-flow testing, each variable occurrence in a program is classified as either a *definition* (def), a *computation-use* (c-use) or a *predicate-use* (p-use). A *def* suggests that a program variable is assigned a value during a computation, while a *c-use* or a *p-use* represents a program variable being referenced in a computation or predicate expression, respectively.

Consider the following fragment of program code. (Note that the line numbers in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
1:   program example
2:   begin
3:     read(x,y);
4:     if (x > 0) then
5:       begin
6:         x = x * 2;
7:       end
8:     write(x,y);
9:   end
```

Figure 2.6 : Data-Flow Testing - An Example

Testing from a *def* to a *c-use* of program variable x requires that a definition-clear subpath can be found from statement 3, which contains the *def,* to statement 6 which

contains the corresponding *c-use*. Similarly, testing from a *def* to a *p-use* of program variable x requires that definition-clear subpaths must be found from statement 3 to both of the successor statements 6 and 8 of the *p-use* at statement 4. A *def-use pair*, or **def-use association**, therefore, is denoted by an ordered triple, where the first element identifies a given program variable, the second element depicts the program statement containing the *def* and third element represents the statement, or ordered pair of statements, containing the corresponding *c-use* or *p-use*. Thus, for the above example, a total of five definition-use associations can be determined: (x, 3, (4, 6)), (x, 3, (4, 8)), (x, 3, 6), (x, 6, 8), (y, 3, 8).

The degree of test coverage varies depending on the data-flow criteria which are chosen. Table 2.1 summarises the different data-flow criteria proposed by Weyuker. For the all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses and all-uses criteria, associations consisting of the definition and use of a variable are tested by *some* subpath. For the all-du-paths criterion, *every* subpath from a definition to its use must be tested. As with the control-flow criteria, the data-flow criteria provide increasing degrees of test coverage, while being progressively more difficult to achieve.

CRITERIA	DESCRIPTION
all-p-uses	Each p-use in the program is tested
all-c-uses/ some-p-uses	Each c-use is tested; if there are no c-uses, then some p-use is tested
all-p-uses/ some-c-uses	Each p-use is tested; if there are no p-uses, then some c-use is tested
all-uses	Each use in the program, both c-uses and p-uses, is tested
all-du-paths	Every subpath to each use in the program, both c-uses and p-uses, is tested

Table 2.1 : Summary of Data-Flow Criteria

2.3.3 Limitations

It is important to realise that for control-flow testing and data-flow testing, there is not necessarily any correlation between high test coverage and low defect rates. The different test coverage criteria are meaningless if testing is not conducted properly; high test coverage will not expose errors. Structural testing possesses weaknesses in that it simply measures what is present in the program code. Thus, no structural testing technique can expose missing-code errors which occur when part of the functionality described in the specification is overlooked and has not been implemented.

Another problem is caused by the fact that test coverage criteria becomes increasingly difficult to satisfy. A non-linear relationship is established whereby the higher the current level of test coverage, the more difficult it is to improve on it. This often requires elaborate test cases to be created in order to marginally increase the level of test coverage. One reason for the increased difficulty is that as the test coverage increases, the untested sections of code tend to become widely dispersed throughout the program.

A further shortcoming of structural testing techniques is highlighted during the testing of concurrent systems [239, 245]. As structural testing techniques cannot account for the concept of timing, program code containing any timing errors may be deemed tested without exposing these errors.

2.4 Summary

In this chapter, the importance of testing as part of the software development phase is emphasized. The testing activities and tools supporting the specification, design and implementation of a program are described. While examining the existing testing strategies, two distinct classes of techniques emerge: static analysis and dynamic testing, and functional testing and structural testing. Emphasis is placed on describing structural

testing techniques, in particular, control-flow testing and data-flow testing. In addition, the effectiveness of these techniques in detecting program errors through the application of increasingly stringent test coverage criteria is assessed. Moreover, the limitations of structural testing techniques are outlined. Structural testing techniques are considered relevant as they provide a basis for the selective revalidation technique described in this thesis.

Chapter 3

Testing During Maintenance

3.1 Introduction

After a software system is delivered to its customers, it may require further modification; it therefore enters the operations and maintenance phase of its lifecycle. Software maintenance can be classified into four distinct stages involving program comprehension, definition of a change proposal, analysis of the proposed changes, and the actual implementation and revalidation of these changes [272]. Figure 3.1 illustrates a typical maintenance cycle.

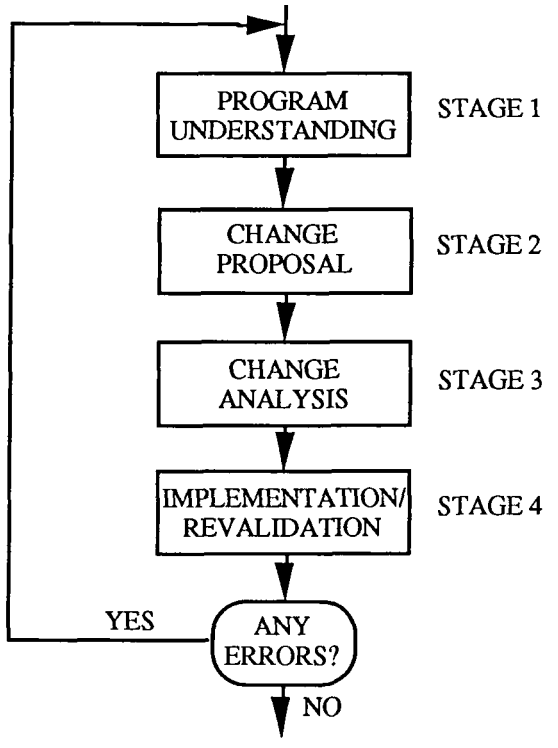


Figure 3.1: The Maintenance Cycle

After a change request has been received from users, maintenance programmers need to gain an understanding of the program; their comprehension of the software is

influenced by its complexity, documentation and self-descriptiveness. Once the program has been examined, maintenance programmers must generate a specification of the modification by means of a change proposal; this helps to specify the maintenance objectives. As part of this proposal, a set of basic modifications are described to satisfy the objectives. Before implementing any of the proposed changes, however, their impact upon the existing program needs to be assessed. It may be possible that a number of change implementations are being considered and so each one must first be examined in terms of its effect on the existing program. In the final phase, the change is implemented and the modified program is retested to ensure its functional consistency. If any errors are discovered during revalidation, a new change request may need to be generated and the entire procedure is then repeated.

Software maintenance is characterised by a predicament similar to that of software testing - it accounts for a large percentage of the overall programming effort and cost, yet possesses only negligible influence on the lifecycle. Two factors, which contribute to the high cost, are the analysis and retesting of the modified software [84]. While contemporary models of software maintenance often emphasize the need to formalise procedures for analysing and testing program changes, few of these models actually provide any guidelines [45]. Maintenance methodologies are therefore deployed with serious consequences. Unexpected side-effects, which inadvertently influence the existing functionality, are often introduced into the system. Recent studies [46] into maintenance practices substantiate these claims by revealing that as many as 53% of the exposed defects are related to side-effects which affect the unchanged portions of a system. It has also been suggested that the probability of invoking an error during maintenance is three to ten times greater than during development [180].

A possible reason for these problems may lie in the fact that practitioners often regard testing during maintenance simply as an extension of their approach to development testing. However, it should be emphasized that testing during development differs from

maintenance testing in a number of ways.

A major difference between the two types of testing usually lies in the availability of a test suite. Software development is concerned with the testing of software which constitutes the completed system. It involves the creation of test cases to satisfy the specification and the associated implementation. As testing proceeds, a test plan is slowly completed using functional and structural test cases which are added to the test suite. Therefore, the prime concern of testing during development is to ensure that the software is correct before its release to customers. During maintenance, testing begins with a possibly modified specification, a modified program and an existing test suite which requires updating. It involves selecting and executing some of the existing test cases, introducing new test cases into the test suite and deleting others. Thus, the testing activities focus on software that has already been released.

Testing during development and maintenance aims at validating a program to ensure its correctness and to locate errors. Although the two types of testing have similar goals, differences exist in the way in which these goals are achieved. During development, the individual components of a system are tested for correct implementation in accordance with the design. They are then integrated into the system in order to test their interconnections with the other components. With maintenance testing, only those parts of the program, which are affected by the maintenance modifications, need to be validated.

In general, adequate resources are allocated for development testing at the beginning of every software project. However, the resources required for maintenance testing activities are often underestimated, or simply ignored, in the project schedule and budget [155]. As a result, maintenance testing may be performed in a limited amount of time, and under the false assumption that testing only certain parts of a software system requires far less time than development testing.

Testing during development is based upon the knowledge that information about the software development process is accessible. It can be assumed that testers have an insight into the documentation concerning the specification and design. They may also access parts of the program code and execute test cases generated by the developers. Furthermore, they can approach developers to question them about specific aspects of the code, exploiting their expertise and intimate knowledge of the code and its structure. With maintenance testing, most, if not all, of this information is unavailable. In most cases, the development team will have been disbanded, or assigned to new projects, and the test cases will have been abandoned shortly after the completion of acceptance testing and the release of the product. In addition, the associated documentation may have been misplaced or destroyed. Thus, maintenance testing is usually approached in an *ad hoc* manner.

Differences between development and maintenance testing also occur with respect to the frequency of the activity. Development testing is a well defined activity, which is undertaken at regular intervals during the project, but it is completed when the product is released to customers. However, the time taken in maintenance testing often spans a time period, which far exceeds that of the development phase, and is conducted far more frequently, preferably after every program modification.

3.2 Regression Testing

3.2.1 A Definition

Activities relating to maintenance testing are characterised by regression testing [213, 228] which has also been referred to as redundancy testing [243]. The Standard Glossary of Software Engineering Terminology [282] defines regression testing as:

"Selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended, adverse side effects, or to verify that a modified system or system component still meets its requirements."

Two types of regression testing can be identified based on whether changes are made to the functionality of a software system or any of its subsystems. **Progressive regression testing** [155] involves retesting in the presence of an altered program specification, or design, where the changes can cause many of the existing test cases to no longer specify a correct input-output relationship. As a result, these test cases can no longer be used to revalidate the program. Typically, progressive regression testing would be conducted at regular intervals during perfective and adaptive maintenance [121].

Corrective regression testing [155] is applied whenever the program specification and design remain unchanged and modifications are restricted to the implementation. As no new functionality is being introduced, most of the existing test cases in the test suite can be reused. Typically, corrective regression testing would be invoked at irregular intervals during corrective and preventive maintenance [121].

3.2.2 Current Practices

At present, regression testing is only considered from a functional point-of-view where the objective is to ensure that the existing program functionality has not been damaged during modification. Regression testing is based on the use of already devised test cases, which minimises the effort required to create test cases, and allows the direct comparison of output from the modified software and the original test cases. This approach, however, relies upon the automation of the regression testing process. Tools have been developed to facilitate the creation, execution, and storage of test cases as well as their automatic replay and comparison of test results.

The use of regression testing tools has become increasingly widespread [48, 150, 162]. In 1983, a survey conducted by Beck-Perkins [17] reported that approximately 20% of the organisations being questioned performed regression testing. By 1986, this figure had risen to 53% according to a survey conducted by the Quality Assurance Institute [283]. In fact, regression testing tools are being used to test an increasing number of software systems, ranging from telecommunications equipment [140, 152, 207, 221] to embedded control systems [22, 25, 279].

The purpose of early regression testing tools, such as the Automated Unit Tester (AUT) [205] and the Fortran Test Procedure Language System (TPL/F) [206] was to serve as automatic software test drivers. While their primary application was in module testing, their facilities were equally well suited to regression testing. With the introduction of the graphical user interface (GUI) and alternative input devices for computers, a new generation of regression testing or 'capture-replay' tools has been developed [9, 133, 233, 234, 247]. Apart from validating the actual application program, these tools also process the interactions between users and the graphical user interface. Subsequently, all input data supplied via the keyboard and mouse, and output data in the form of windows, icons and menus can be captured and later replayed.

The growing importance of regression testing is also reflected in the number of dedicated software testing environments which have incorporated regression testing capabilities alongside their more traditional testing facilities. Examples of such test environments include the Virtual Terminal, Scaffold Test Package Automation Tool and Test Package Standard [73], and Buster [10].

Without doubt, the automation of regression testing has helped considerably in alleviating some of the tedious and time-consuming tasks associated with it. However, the reduction in testing effort is, in effect, confined to automating the capture of test *inputs* and the comparison of test *outputs*. As a result, maintenance programmers are still

faced with the problems and uncertainty concerning which parts of the program functionality and code are affected by the changes, and how thoroughly these parts should be retested. These problems have so far not been adequately resolved and there is neither an accepted technique, nor criterion, for program revalidation. This thesis, therefore, proposes the development of a systematic and efficient approach to regression testing known as *selective revalidation* [101].

3.3 Selective Revalidation

A single program modification is usually revalidated by rerunning *all* of the existing test cases, rather than a selected portion of them, to ensure that the change has not introduced any side-effects into the program [232]; this strategy is often referred to as the *retest-all* strategy. Alternatively, a modification cycle is imposed in which a series of changes are made over a period of time, and the changes are then collectively tested at the end of the cycle using the entire test suite. Other retesting strategies envisage the rerunning of a set of confidence tests aimed at exercising the major system features, the execution of a number of randomly, or intuitively, selected test cases or the exercising of all those test cases which traverse the modified program parts [89].

While rerunning an entire test suite can prove to be costly, and in some cases infeasible, the intuitive approaches to regression testing assume that maintenance programmers have an intimate knowledge of the program's purpose and its test suite. If such personnel are available, then it may be possible for them to select specific test cases which directly exercise the modified program functionality. In some cases, the maintenance programmers may possess sufficient knowledge of the program structure in order to select additional, structural tests which can be used to complement the already selected set of functional test cases. However, it is doubtful whether the extent of any given program change can be systematically analysed and an optimal set of regression tests selected to exercise the change.

The objective of a technique for selective revalidation, therefore, is to assist maintenance programmers in systematically analysing the modifications made to the program code, and in efficiently selecting and updating the test cases from the existing test suite [98]. The technique may thus be used to complement and enhance the current regression testing process. However, to achieve the objective, the technique must address the issue of **test suite classification** and solve the problems of **test suite management.**

3.3.1 Test Suite Classification

In response to changes in the program specification, design, or implementation, a test suite can be classified into four different categories of test cases. An accurate classification of the test suite is essential in order to later on define the problems of test suite management. Below, a description of each test case category is given.

Reusable tests constitute those functional and structural test cases which are associated with the unmodified portions of the program. These test cases traverse those parts of the software which are neither directly, nor indirectly, affected by the modifications and which continue to exercise the same functionality and code as before. Thus, the test cases should be retained in the test suite between testing sessions.

Retestable tests represent the set of test cases which need to be rerun after modifications have been made to the program functionality or code. They include functional and structural tests which exercise those sections of the specification, design and code which have been directly, or indirectly, affected by the changes. As a result of these changes, the retestable tests may now exercise different aspects of these program attributes.

New tests, both functional and structural, may need to be created in order to satisfy their respective test coverage criterion. Maintenance activities may include enhancements to the program's functionality which require the generation of new, functional test cases. However, these tests may not achieve the desired level of *structural* test coverage and therefore additional, structural test cases may need to be created to improve this level of coverage.

Unnecessary tests represent those test cases in the test suite which may be removed once the modifications have been validated. Structural and functional test cases are referred to as **redundant tests** if, after modifications, they exercise program functionality and traverse program code which has already been validated by other test cases in the test suite. Thus, their removal does not affect the respective test coverage, because other test cases provide the same measure of coverage for the program.

3.3.2 Test Suite Management

In response to program modifications, a test suite must be updated to reflect the changes. Although the test suite can now be classified into the different categories of test cases, problems still arise concerning how to determine these different categories of test cases during regression testing. Two problems, in particular, need to be solved. The problem of **test selection** [156] is concerned with the determination of the *retestable tests*; it is desirable to select a representative set of test cases from the test suite to ensure the integrity of the modified program. The problem of **test update** [156] is concerned with the updating of the test suite and involves the identification of the *redundant tests* as well as the design of *new tests*; it is desirable to maintain a representative set of test cases in the test suite to ensure its high quality. In this thesis, therefore, a technique for selective revalidation is developed which addresses the problems of test selection and test update and provides a solution to them.

3.4 Summary

The role of testing during the operations and maintenance phase of the software lifecycle is examined. It is found that apart from the analysis of program modifications, the testing of these modifications contributes to the high cost of maintenance. By examining the different stages which occur in a typical maintenance cycle, it is discovered that, while the activities related to the modification of the software have been clearly defined, the procedures associated with the validation of the modified software are still lacking the required formality and rigour. This has had a detrimental effect on the quality of the modified software with unexpected and undesirable side-effects being introduced into the software.

The lack of a formal approach to maintenance testing is examined and it is found that its cause may lie in the attitude of programmers - they regard maintenance testing simply as an extension of development testing. A number of differences between testing during development and maintenance are subsequently examined.

The subject of regression testing is introduced and a distinction is made between different types of regression testing according to whether, or not, the maintenance activities include changes to the program functionality. Current regression testing practices and tools are examined, and it is discovered that existing techniques promote a rather *ad hoc* approach to regression testing.

The subject of selective revalidation is defined and a corresponding technique is proposed which effectively complements and enhances current regression testing practices. As a result, maintenance modifications may now be analysed and retested with respect to both the program functionality and code. However, selective revalidation is faced with a number of problems which are addressed by the technique described in this thesis. The issue of test suite classification categorises the test cases in the test suite into

retestable, reusable, new and redundant tests, while the problems of test selection and test update require a set of retestable tests to be chosen in order to revalidate the modifications and redundant test cases to be determined during the updating of the test suite.

Chapter 4

Techniques for Selective Revalidation

4.1 Introduction

The work by Hartmann-Robson [101] defines the subject of selective revalidation. It discusses existing techniques and highlights five important issues concerning the subject: a) the majority of techniques for selective revalidation concentrate upon the analysis and retesting of individual program modules; b) few of the techniques recognise the importance of change analysis as part of selective revalidation; c) most techniques do not select a minimal set of retestable tests and do not provide a way of identifying redundant tests; d) little experimental evidence is available to demonstrate the benefits of selective revalidation; and e) very few techniques can demonstrate a consistent approach to the selective revalidation of the program specification, design and code.

In this chapter, each of the above issues is examined. In Section 4.2, the capabilities of existing techniques with respect to code and change analysis are investigated. Section 4.3 describes the revalidation criteria which are applied by individual techniques in order to select and update test cases in the test suite. Section 4.4 assesses any experimental evidence which has so far been presented. Finally, Section 4.5 analyses two approaches to selective revalidation, which are capable of retesting changes to a program's specification and design, as well as to its implementation.

4.2 Code Analysis

Code analysis forms an important aspect of every selective revalidation technique and is used to examine the dependencies which exist between different entities in a program.

Code analysis techniques[1] obtain this dependency information by static analysis of the program code. In particular, the *quality* of these techniques needs to be assessed as it reflects upon the type of testing strategy and change analysis which can be used for selective revalidation.

4.2.1 Testing Strategies

Code analysis is used to determine the test requirements of different structural testing strategies. Therefore, it influences the *applicability* of selective revalidation techniques by restricting their use to certain testing strategies. Two types of code analysis are used in conjunction with selective revalidation, namely *intraprocedural code analysis* [6, 19, 136] and *interprocedural code analysis* [5, 15].

Intraprocedural code analysis confines itself to analysing individual program modules and makes a number of assumptions concerning the use of global variables and module parameters for called modules. Its main advantage, however, lies in the fact that it is easy to implement. Consequently, intraprocedural code analysis forms the basis of *most* selective revalidation techniques. For example, the techniques developed by Harrold-Soffa [90], Taha *et al.* [238], and Ostrand-Weyuker [201] are based on data-flow testing of individual program modules. Moreover, Leung-White [154] describe a technique for selective revalidation which is based on statement testing. Other techniques, such as those developed by Benedusi *et al.* [18] or Fischer *et al.* [65, 67] are based on the path testing of program modules.

Interprocedural code analysis removes the restrictions imposed by intraprocedural code analysis to account for the effects of global variables and module parameters when another module is called; it can be categorised into *flow-insensitive* interprocedural code analysis [49] and *flow-sensitive* interprocedural code analysis [30, 93, 131]. While the

[1] Code analysis techniques are described in Chapter 5.

former approach summarises the data-flow information pertaining to each parameter and global variable in a called module and *ignores* the control structure of the called module. The latter approach, however, takes into the account the control structure of the called module. Subsequently, flow-sensitive interprocedural code analysis is far more accurate than either flow-insensitive interprocedural or intraprocedural code analysis, but it is also more costly to implement. This fact is reflected by the small number of selective revalidation techniques based on interprocedural code analysis.

The work by Harrold-Soffa [91, 96] describes a technique for selective revalidation which is capable of retesting modifications made to a collection of program modules and is based on data-flow testing. It relies upon the use of flow-sensitive interprocedural code analysis [93] for ensuring that definition-use associations, which correspond to module parameters and global variables, are examined and tested. However, this technique has yet to be implemented and applied in practice.

The work by Leung-White [157] describes a technique for selective revalidation which allows the analysis and retesting of a collection of program modules written in Pascal. The technique does not incorporate flow-sensitive or flow-insensitive interprocedural code analysis, but instead examines the program's calling hierarchy or **call-graph** in order to establish its test requirements. In response to module changes, a *fire-wall* [158] is defined for focusing the retesting effort on those program modules which may directly or indirectly be affected by the modified module. As revalidation is limited to the calling interaction between program modules, neither global variables, nor module parameters, are examined by this technique.

Apart from affecting the *applicability* of selective revalidation with respect to different testing strategies, the quality of code analysis also influences the *accuracy* with which program dependencies and test requirements are determined. The majority of techniques for selective revalidation [67, 90, 160, 201, 238] are based on code analysis techniques which examine programs written in Pascal [129] or Fortran [281]. With these techniques,

conservative assumptions are made concerning the analysis of composite variables and pointer variables. However, a lack of accuracy in analysing these variables as well as their aliases can significantly influence the number of dependencies, which are identified, and affect the number of test requirements being generated; this is especially true of the analysis and testing of programs written in the C programming language [113, 202]. Later on in this thesis, language-specific program dependencies are described using C programs.

4.2.2 Change Analysis

Another important aspect of code analysis is change analysis. With respect to selective revalidation, change analysis is used to identify the impact of the proposed program modifications and select those test requirements from a program's testing history which correspond to the affected program statements. Thus, change analysis helps to focus the retesting effort on those program paths traversing the affected portions of the program and influences the process of choosing a set of retestable tests during test selection. As change analysis is characterised by the accurate analysis of a program's control and data structure[2], it is directly affected by the quality of the code analysis being conducted.

The majority of selective revalidation techniques consider only the *direct* effects of a modification. Very few techniques utilise change analysis in order to identify the *indirect* effects of a program change. One selective revalidation technique, which *does* use change analysis to identify affected test requirements and program paths, is described by Fischer *et al.* [65]. Their algorithm for change analysis is defined solely in terms of the control structure of a program. With this technique, a program is represented in terms of a control-flow graph[3] and the reachability between program statements is examined.

---

[2] This is explained further in Section 5.5.

[3] Control-flow graphs are described in Section 5.3.1.

Consider the following fragment of program code. (Note that the line numbers in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
1:   program example
2:   begin
3:     read(a);
4:     a = a + 2;
5:     b = 10;
6:     c = 20;
7:     if (a > 0) then
8:         begin
9:           a = b;
10:        end
11:    else
12:        begin
13:          a = c;
14:        end
15:    write(a);
16:  end
```

Figure 4.1: Fischer's Algorithms - An Example

According to Fischer's algorithm, a modification to statement 6 would result in *all* computational statements being reported as affected; the reason for this being that in terms of the program's control structure all other program statements either reach to or can be reached from statement 6. As the program contains two possible paths, both of which traverse the modified statement and represent test cases, then *both* of these test cases need to be rerun. However, careful inspection of the program's control and data structures reveals that only statements 13 and 15 would be affected by a change to statement 6, resulting in only *one* of the two test cases needing to be rerun. Thus, the usefulness of Fischer's algorithm is limited as change analysis based solely on a program's control structure results in an overestimate of the number of affected program statements and subsequently, test cases.

Further work by Fischer *et al.* [67, 68] resulted in the definition of two new algorithms, simply referred to as A and B. These algorithms trace the impact of proposed code changes using the program's control structure and variable information, which describes the assignments and uses of variables within each program statement. Algorithm A identifies those program statements which reach to the modified statement in terms of control structure *and* contain variable assignments and references which affect those in the modified statement - the algorithm therefore attempts to solve the **backward data-flow problem** [136]. In a similar manner, algorithm B determines those statements which can be reached from the modified statement in terms of control structure *and* contain variable assignments and references that are affected by those in the modified statement - the algorithm therefore attempts to solve the **forward data-flow problem** [136].

Applying these two algorithms to the previous modification would now result in statements 6, 13 and 15 being flagged as affected. However, the algorithms contain inconsistencies in their definitions which still cause them to select an incorrect set of affected program statements; this may affect the number of paths and retestable tests being selected. For example, the application of algorithm A to a modification of statement 15 results in statements 3, 4, 5, 6, 9, 13 and 15 being flagged as affected. However, careful inspection of the program code reveals that statements 3 and 4 should not be flagged as variable a is redefined on every program path leading to its use in statement 15.

In constrast, a change to the computation in statement 4 would cause algorithm B to indicate that only statements 7 and 15 are affected by the modification. Apart from neglecting the fact that variable a is redefined at statements 9 and 13, the algorithm does not consider the possibility that a modification to the computational statement 4 could affect the value of the predicate expression in statement 7 and cause the path condition to be changed. Thus, the algorithm should also flag those statements as affected which contain dependencies straddling the affected predicate expression and lie on the same control-flow path as the modified statement. For a modification to statement 4, this would

result in statements 5/9 and 6/13 being flagged. In addition, the algorithm should examine those statements, which are control-dependent upon the evaluation of the predicate expression and subsequently affect statements throughout the program. For a modification to statement 4, this would result in statements 9 and 13 causing statement 15 to be flagged.

An alternative approach to change analysis, proposed by Leung-White [154], is based on the concept of a *retestable unit*. The principles of a retestable unit are closely linked to those of **program slicing**. Two algorithms are defined by Leung-White in order to provide solutions to the backward data-flow problem and forward data-flow problem. While the former problem is adequately addressed, the algorithm responsible for solving the latter problem contains inconsistencies which may result in an underestimate of the number of affected program statements. Consider the following fragment of program code. (Note that the line numbers in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
1:   program example
2:   begin
3:     read(x,y);
4:     if (x > 0) then
5:         begin
6:           x = x + y;
7:         end
8:     else
9:         begin
10:          y = y - x;
11:        end
12:    if (x ≥ 10) then
13:        begin
14:          a = x;
15:        end
16:    else
17:        begin
18:          a = y;
19:        end
20:    write(a);
21:  end
```

Figure 4.2: Retestable Unit - An Example

According to the algorithms defined by Leung-White, a modification to the predicate expression in statement 12 would cause program statements 3, 6, 12, 14, 18 and 20 to be reported as affected. For the backward slicing algorithm, any definitions associated with the use of variable x in statement 12 are traced; this implicates statements 3 and 6. For the forward slicing algorithm, statements 14, 18 and 20 are examined; statements 14 and 18 are implicated because they are control-dependent upon statement 12, while statement 20 is flagged as it is data-dependent upon statements 14 and 18[4]. This approach, however, fails to implicate those statements which form dependencies that straddle the modified predicate expression. Thus, in response to the above modification, the forward slicing algorithm should also consider statement 10 as affected, because it is data-dependent upon statement 18 which was included in the original set of affected statements and lies on one of the control-flow paths traversing the modified statement. This thesis later on describes a technique for change analysis, which extends the work by Leung-White, and is based on the concept of program slicing.

4.3 Revalidation Criteria

The advantage of selective revalidation over current regression testing practices lies in its ability to distinguish between the different categories of test cases in a test suite and determine the sets of retestable and redundant test cases. To achieve this, however, the individual techniques must first develop and then apply suitable revalidation criteria to the problem of test selection and test update. In the following section, emphasis is placed on examining the *quality* of these criteria.

4.3.1 Test Selection

A number of techniques for selective revalidation [90, 96, 201, 238] apply their test selection criteria to programs which have been validated using data-flow testing. These

[4] Control dependency and data dependency are described in Chapter 5.

criteria require only those test requirements to be identified which correspond to modified or deleted definition-use associations. This approach leads to the selection of a set of retestable tests which ensure that those program parts, which are *directly* affected by the modifications, are validated. However, it does not consider those definition-use associations and corresponding program code which may be *indirectly* affected by the modifications. This may lead to situations in which the test selection criteria underestimate the number of affected test requirements and thus cause too many retestable tests to be selected.

The work by Leung-White [154], which is based on statement testing, introduces an *all-essential assumption* in order to simplify the problem of test selection. It assumes that every statement on a program path contributes to the overall path computation. Thus, if a program statement is executed by n test cases, then its subcomputations are also exercised by those n test cases. A change to a given program statement would therefore be affected by computations on all program paths leading to the modified statement. Similarly, the modified statement itself would perform a computation which may affect statements on paths reaching from the changed statement. Therefore, the revalidation criterion developed by Leung-White simply selects *all* test cases for rerun which traverse the modified statement. However, this assumption is not valid for all cases as the statement being modified may only rely upon subcomputations on certain program paths leading to the modified statement and affect computations on some of the paths reaching from the changed statement. This may lead to situations in which the test selection criterion overestimates the number of affected test requirements and thus causes too many retestable tests to be selected.

To implement their approach, Leung-White [154] define a bit vector which is associated with each program statement or its corresponding node in the control-flow graph of the program[5]. If a test case i traverses a statement, then the *ith* bit of the

[5] Control-flow graphs are described in Section 5.3.1.

corresponding node's bit vector is set to one. Whenever a node is modified or deleted, the corresponding bit vector determines the set of test cases which needs to be rerun.

The technique, described by Benedusi *et al.* [18], selects its test cases based on the automatic analysis of those program paths which have been modified or deleted during maintenance. To achieve this, the program's control structure is first transformed into its equivalent algebraic expression using an algebraic representation known as exp(prog) [33]. This algebraic expression consists of operands and operators which depict the different control structures and sequencing operations found within a program. By applying the distributive law[6] with respect to the operators, a series of expressions can be derived for the paths through the program. Problems, such as loops, are overcome in this technique by grouping program paths, which execute the loop zero or more times, into an *exemplar path* (e-path).

For test selection, the path expressions of the program, before and after the modifications, are determined and then automatically compared using difference operators. Subsequently, the test suite can be categorised into *unmodified paths*, *modified paths*, *cancelled paths* and *new paths*. Once the modified paths have been determined, the corresponding test cases are selected from the testing history. However, the accuracy of test selection is limited by the algebraic representation, exp(prog), which can only depict the control structure, but not the data structure, of a program. Thus, a situation may arise whereby the first statement in a program is modified and all paths are flagged as being modified. This would result in all test cases in the test suite being rerun and constitute an overestimate in the number of retestable tests.

Apart from overestimating the number of retestable tests due to the lack of an integrated mechanism for change analysis, the test selection criteria described above also fail to account for the possibility of extraneous tests being introduced into the set of

[6] The distributive law for an algebraic expression a(b + c) = ab + ac.

retestable tests. Thus, whenever a set of retestable tests is identified, the possibility exists for considerable overlap, that is redundancy, in the set of test cases required to exercise the affected test requirements. In fact, the majority of techniques described above do not determine an *optimal* set of retestable tests. However, the work by Hartmann-Robson [103] describes an approach to test selection which *can* provide such an optimal set, and relies upon the combined application of operations research and change analysis to do so. This work is described later on in this thesis.

4.3.2 Test Update

For test update, Leung-White [154] define a set of bit-vector operations which are used to update the test cases associated with modified program statements[7]. They include a replacement operation for updating a test case execution whenever it traverses a statement, a bit shifting operation for deleting old test cases that no longer exercise the statement and an appending operation for the addition of new test cases. In the technique, described by Benedusi *et al.* [18], the comparison of path expressions, before and after the modifications, results in the generation of new test cases to exercise the modified paths and the deletion of old test cases which no longer exercise valid paths. Those techniques for selective revalidation, which are based on data-flow testing [90, 96, 201, 238], determine whether additional test cases must be created to exercise any new definition-use associations, or test cases need to be deleted should the definition-use associations, which they previously exercised, no longer exist.

These selective revalidation techniques, however, fail to consider the possibility of redundant tests being introduced into the test suite during test update. This can lead to an uncontrolled increase in the size of the test suite as successive maintenance changes are implemented. The approach described by Leung-White [155] characterises most other selective revalidation techniques with respect to this problem. It proposes that those test

[7] Recall that these statements are defined as the nodes of the program's control-flow graph.

cases, which are identified during test selection, should be executed until the imposed test coverage criterion is satisfied. Any remaining test cases should then be classified as redundant and eliminated.

However, the random choice of test case order can have a substantial impact on the number of redundant test cases which are introduced. A number of selective revalidation techniques [94, 160] have therefore suggested the use of operations research to determine redundant test cases in an updated test suite. Their approach to the problem of test update is based on the earlier work conducted by Hartmann-Robson [99].

4.4 Experimental Evidence

The concept of selective revalidation has been adopted by relatively few prototype maintenance environments and testing tools. This may give an indication of the difficulty associated with implementing a technique for selective revalidation. For example, the Test Inc. incremental data-flow testing tool [170] forms the basis of a selective revalidation technique described by Harrold-Soffa [92]. However, the tool confines itself to analysing small programs or modules written in a subset of Pascal.

Benedusi *et al.* [18] describe their selective revalidation technique as part of the MAINT_DB maintenance environment which retains information such as the specification, design documentation and code for a program. The *Post-Maintenance Testing* subsystem of the MAINT_DB environment is then responsible for coordinating the regression testing activities and utilises testing and dependency information supplied by the environment. However, the environment is restricted to analysing and retesting program modules implemented in COBOL or Pascal [129]. Other techniques for selective revalidation [65, 273, 238] are used in the analysis and retesting of program modules written in Fortran [281] or a subset of Ada© [280].

© Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

None of the tools described above have been used to evaluate the concept of selective revalidation. The resulting lack of experimental evidence means that any claims concerning the benefits of selective revalidation remain largely theoretical. However, in a recent experiment conducted by Leung-White [158], a sample program consisting of over five hundred lines of Pascal source code, and implemented in thirty-two modules, was analysed, modified and retested. Initial validation resulted in a total of 120 unit tests, 235 integration tests and 104 system tests were executed. A series of modifications, which included changes to the program call-graph, were implemented. Leung-White then applied their technique for selective revalidation and compared its effectiveness against the retest-all strategy in which the entire test suite needed to be rerun. It was found that selective revalidation achieved a similar level of confidence to the retest-all strategy in terms of its error detection capabilities. More importantly, however, it indicated that with the help of selective revalidation, significantly fewer, on average 66% fewer, test cases needed to be rerun.

While these results are encouraging, a number of important issues should be highlighted. First, all activities relating to the evaluation of the actual program including its instrumentation, code analysis, and test suite management were conducted manually. This could give rise to a considerable margin of error and have a significant influence on the outcome of any experimentation. Second, the revalidation criteria, which were applied, neither considered the possibility of selecting an optimal set of retestable tests, nor applied a mechanism for identifying redundant test cases. Third, the technique only compared its results with those obtained for the retest-all strategy. It did not provide a comparison with other selective revalidation techniques.

However, Hartmann-Robson [102] propose the development of a tool suite based on the technique for selective revalidation described in this thesis. This tool suite, which is described later on in the thesis, allows the tasks associated with selective revalidation to be automated and ensures that both an optimal set of retestable tests and any redundant tests can be identified. More importantly, however, it enables software, which has been

validated using various structural testing strategies, to be retested and allows comparisons between different selective revalidation techniques to be made.

4.5 Alternative Approaches

The majority of techniques for selective revalidation are applicable only to the analysis and retesting of modified program code. They do not consider the impact of modifications on the program specification or design. In the following section, however, two strategies are examined which revalidate a program's specification and design as well as its implementation.

4.5.1 Specification-Based Revalidation

Yau-Kishimoto [273] have developed a technique for selective revalidation based on information derived from the program specification and code. The technique assumes that the program specification is defined in terms of a cause-effect graph and that the program code is represented by its control-flow graph[8]. It then applies the theory of revealing subdomains[9] in order to define two types of partitions for the program.

The first partition is created by identifying different combinations of input conditions from the *modified* program specification based on the cause-effect graph. Another partition, referred to as the *path partition*, is established by considering each executable program path through the graph as corresponding to an input partition class[10]. The two existing partitions are then intersected to form a third partition or *testing partition.* In practice, the derivation, application and updating of revealing subdomains and cause-effect graphs is difficult.

---

[8] Control-flow graphs are described in Section 5.3.1.

[9] Revealing subdomains are described in Section 2.2.2.

[10] The exception is formed by those paths that differ in the number of loop iterations.

For the purposes of test selection, those paths in the control-flow graph, which correspond to the *reaching set* of a modified program statement, are marked. Data-driven symbolic execution is then used to evaluate the predicates encountered during program execution. This form of symbolic execution requires no program path to be specified as input data in the form of previous test inputs is used to evaluate the predicates. The symbolic execution tree, which is generated for the modified program, contains an entry for each node in the control-flow graph. Every entry maintains certain state information pertaining to the path constraints. The predicates, which are encountered at the decision points along the path, are then evaluated; the combination of these predicates dictates the input values for which the program path can be executed. Each predicate results in a constraint on the input data which is then conjoined with all previously evaluated constraints for this path to form the path condition.

During symbolic execution, a path through the modified program is evaluated using a depth-first search, and is subject to a number of outcome selection criteria. Thus, symbolic execution is continued for as long as: it follows any path in the reaching set of the modified nodes, and it selects, at every decision point, an outcome whose constraint is satisfied. The symbolic execution is then repeated until an exit point for the program is encountered. Any test cases, whose input values do not select the correct outcome for a given predicate, are stored at that particular entry in the tree. These are processed later on, after an exit point of the program is reached, by performing a backtracking operation to the root of the symbolic execution tree.

Once the symbolic execution terminates, those test cases which have satisfied all path predicates and traversed the reaching set of the modified program statement are collected. This also includes any test cases which may have been identified as part of the backtracking operation. Together, they are collated in a *test information table* which for each test case contains its number, a tree entry at which symbolic execution terminated and a boolean outcome to indicate its suitability for rerun.

The revalidation criteria are then applied by examining the path conditions at the leaf entries of the symbolic execution tree. These entries represent the valid input conditions for the modified program. Subsequently, test cases, which are attached to these leaf entries and cover the modified input partitions, are rerun. If any of these input partitions remain unexercised, then new test cases need to be generated to exercise the partition and satisfy the test coverage criterion.

For the technique described by Yau-Kishimoto [273], the derivation of the input partition, which represents the program code, is difficult as program logic can be complex and equivalent classes of program paths are difficult to identify. In addition, symbolic execution is more effective in testing of numeric, rather than non-numeric programs. Its application to a complex, numeric program may result in large amounts of output representing lengthy, symbolic expressions which must be correlated with the input partition constraints. A further disadvantage is that symbolic execution requires a large amount of computing resources and can really only be justified when applied to safety-critical systems, where the gain outweighs the cost of using such a technique.

4.5.2 Design-Based Revalidation

Benedusi *et al.* [18] describe a technique for selective revalidation which ensures the consistency of both the program design and code. For their strategy to be applicable, a program design must be represented in the form of low-level design documentation such as Jackson diagrams [128] or Warnier-Orr diagrams [252]. The work by Benedusi *et al.* exploits the fact that a detailed design specification can be expressed in terms of pseudocode or flowcharts and that it is directly related to the underlying source code. It applies the algebraic representation, exp(prog), to the design specification in order to transform it into its equivalent set of algebraic path expressions. For test selection, the path expressions before and after the design modifications are determined and automatically compared using difference operators. The modified, or deleted, paths through the design specification are identified and the corresponding test cases are

selected for rerun. These test cases then exercise affected constructs in both the program design and code.

4.6 Summary

In this chapter, a review of existing techniques for selective revalidation is conducted. The state-of-the-art is examined based on five important criteria which address the code analysis and change analysis capabilities of different techniques, their revalidation criteria, the presentation of experimental evidence, and the use of these techniques in analysing and retesting of modifications to the program specification or design.

A number of important conclusions are drawn from the review. The majority of selective revalidation techniques are based on code analysis which allows them to be used in conjunction with structural testing strategies and be applied to the unit testing of modified software. Few techniques have been defined for integration testing; this is due to a lack of suitable code analysis techniques. Those selective revalidation techniques, which *do* propose such approaches, have either not yet been implemented or make a number of assumptions which limit their use. Moreover, the majority of selective revalidation techniques are based on code analysis techniques which make conservative assumptions in their analysis of composite variables and pointer variables.

Few techniques for selective revalidation recognise the importance of change analysis in influencing the number of retestable tests chosen during test selection. As a result, the revalidation criteria being applied may tend to overestimate the number of retestable tests. More importantly, however, the majority of these techniques are incapable of selecting a minimal set of retestable sets and determining any redundant test cases in the test suite.

Very little experimental evidence is available which can be used to justify the claims made by existing techniques concerning the benefits of selective revalidation. Therefore, the concept of selective revalidation remains largely theoretical. However, encouraging

results have been presented by a recent experiment in which a sample program was initially validated, a set of program modifications applied and a technique for selective revalidation used to determine the number of retestable tests. These results indicate that with the help of selective revalidation, significantly fewer test cases in the test suite needed to be rerun.

The majority of techniques for selective revalidation are applicable only to the analysis and retesting of modified program code. However, two strategies are examined which consider the impact of modifications on the program specification or design and can be used to revalidate them. As part of the conclusion, this thesis suggests ways of extending the existing technique for selective revalidation to include the retesting of the program specification and design.

Chapter 5
Code Analysis Techniques

5.1 Introduction

Code analysis is a generic term used to denote those activities where the primary emphasis is on *examining* a piece of program code. Code analysis techniques can assume various forms, for example, the error and anomaly detection of Osterweil-Fosdick [199], Huang [123] and Jachner-Agrawal [127], the work by Burke-Cytron [26], Ferrante *et al.* [63] and Cooper *et al.* [50] on the optimisation and parallelisation of compilers, and the work by Burke-Ryder [27], Lu-Qian [168] and Zadeck [277] in incremental data-flow analysis.

Two aspects of code analysis which contribute to selective revalidation are: determining the dependencies existing between different entities in a program and using these dependencies to assess the impact of proposed program modifications. For any maintenance operation, programmers need to gain a general understanding of how the program works, together with knowledge about which sections of code are important to the maintenance operation and how these have been tested. Maintenance programmers need to assess the extent of the modifications by considering both the direct *and* indirect influences of any changes. It is, therefore, important that suitable code analysis techniques are developed as part of a technique for selective revalidation.

To be able to discuss these code analysis techniques, it is important that a suitable structure be used to represent the particular properties of a program. The graph has been found to be a suitable structure for other code analysis techniques [32, 271], so it will be used in this thesis to record and describe the dependencies within a program.

This chapter concentrates on developing code analysis techniques for *dependency analysis* and *change analysis*. Section 5.2 explains some important graph terminology. Section 5.3 describes dependency analysis in which control dependency and data dependency within a program are defined in terms of graph relations and may be combined to form a Program Dependency Graph. Section 5.4 examines data dependencies induced through the use of composite variables as well as pointer variables and their aliases. Emphasis is placed upon describing these dependencies with respect to programs written in the C programming language. As a result, an updated definition of data dependency is presented. In Section 5.5, a technique for change analysis is developed, which makes use of the Program Dependency Graph, in order to identify the direct and indirect effects of a proposed program modification. This technique is then used to extract those test requirements from the program's testing history which correspond to the parts of the program affected by the changes.

5.2 Graph Terminology

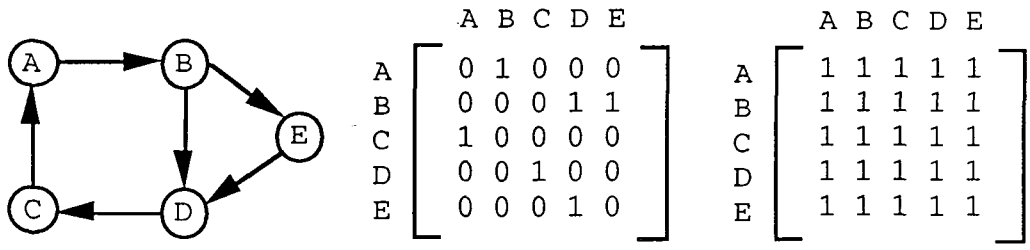
Graph theory suffers from a lack of a standard terminology. It is, therefore, necessary to give a brief explanation of the terminology used in this thesis. This terminology is taken from [82, 179].

Formally, a graph G is represented by a pair (N,E), where N is a finite set of vertices, or nodes, $\{n_1, ..,n_E\}$, and E is a finite set of ordered pairs called edges, $\{(n_1,n_2), ..., (n_{E-1},n_E)\}$.

Figure 5.1(a) illustrates a **directed graph** structure where the direction of the edges between nodes is of importance[1]. An ordered pair (n_x,n_y) denotes an edge connecting node n_x with node n_y. Two nodes are classed as being *adjacent* if there exists an edge connecting the two nodes. With a directed graph, the existence of the edge (n_x,n_y) cannot

[1] This is in contrast to *undirected graph* structures where no importance is placed upon the order of the nodes of an edge.

be used to infer the existence of an edge (n_y,n_x). Therefore, when determining adjacent nodes, the existence of edge (n_x,n_y) implies that n_y is adjacent to n_x, but does not mean that n_x is adjacent to n_y. Therefore, node n_x is described as the *predecessor* of node n_y and node n_y is termed the *successor* of node n_x.



(a) A Directed Graph (b) Connectivity Matrix (c) Reachability Matrix

Figure 5.1 : Graph Representations

The *indegree* of a node n_x represents the number of edges entering at node n_x, while the *outdegree* of a node n_y represents the number of edges leaving n_y. A node with an indegree of zero is referred to as a *source node*. Similarly, a node with an outdegree of zero is known as a *sink node*.

With respect to software, a *path* is a sequence of nodes such that successive pairs of nodes are adjacent. It starts with a source node and may end with one, or more, sink nodes. The path $n_1,n_2, ..., n_i$ is a path of length (i-1) from node n_1 to n_i. Correspondingly, a *subpath* represents a path of any length between $n_2, ..., n_{(i-1)}$. If, in a given path, each node and edge appears only once, then the path contains no cycles and is referred to as a *simple path*. In contrast, a *simple loop path* signifies a path in which a node may appear twice, but each edge appears only once.

Apart from the pictorial representation of a directed graph, shown in Figure 5.1(a), a directed graph can be described by its *connectivity matrix*. It represents a boolean matrix in which adjacent nodes, that is edges, are identified by a TRUE (1) value, and all other

values are set to a FALSE (0) value. The connectivity matrix corresponding to the directed graph is shown in Figure 5.1(b).

The *transpose* of a matrix describes the operation whereby the rows and columns of a matrix are interchanged so that the first row of the matrix forms the first column, and so forth. However, in terms of the pictorial representation of a directed graph, the transpose operation implies that the directions of the edges in the directed graph are simply reversed.

The set of edges E in a directed graph represents a specification of subpaths of length unity. Based on the composition of relations, it is possible to show that the composition of E with itself can produce subpaths of length two; the composition of this result with E again can produce subpaths of length three, and so forth. This leads to an induction argument which represents the process of computing the *transitive closure* of E [276]. The *kth* power of E is defined as $E^k = E^{k-1} \circ E$ where $\circ$ represents the compositional operator and k lies in the range 2 to N, the number of nodes in the directed graph. Subsequently, a path exists between two nodes n_x and n_y in the graph iff $n_x E^k n_y$ for $1 \leq k \leq N$. Hence, the transitive closure of E is given by $E^* = \cup_{(k \in N)} E^k$. The matrix E^* is referred to as the *reachability matrix* of the directed graph and is illustrated in Figure 5.1(c). Transitive closure can be computed either by the repeated self-multiplication of E, which requires order $O(N^4)$ operations or by applying one of the numerous different transitive closure algorithms [59, 186, 215, 253, 254] which can usually reduce the computational complexity of computing transitive closure to order $O(N^3)$ operations.

5.3 Dependency Analysis

Dependency analysis is the process of determining the control dependency and data dependency in a program; it involves analysing its control structure and data structure. A set of graph relations can be defined in order to express these dependencies. The formal

notation used to describe these graph relations has been adopted from the programming language ML [183]. Consider the following generic use of `let`:

let <declarations> *in* <expression>

Here, <declarations> consists of a sequence of name bindings that may be used inside <expression>. The scope of these bindings is limited to <expression>. The result of evaluating <expression> is returned as the value of the `let` construct. For example, the following expression evaluates to 3.

```
let a=2, b=1 in a+b
```

Names may also be bound using 'pattern matching' between two sides of the symbol =. For example, if the complex number X + Yi is represented by the tuple (X,Y), then the sum of two complex numbers $complex_1$ and $complex_2$ may be defined as follows:

```
sum(complex1,complex2) =
let     complex1 = (real1,imaginary1),
        complex2 = (real2,imaginary2)
in      (real1 + real2, imaginary1 + imaginary2)
```

In the above expression, $real_1$, $imaginary_1$, $real_2$, and $imaginary_2$ were all defined using pattern matching. The symbol $\cup$ is used to denote set union. For example, if $S = \{x_1, x_2, ..., x_n\}$, then:

$$\bigcup_{x \in S} f(x) \equiv f(x_1) \cup f(x_2) \cup \ldots \cup f(x_n)$$

Set unions may also be composed. For example, if $S_1 = \{x_1, x_2\}$ and $S_2 = \{y_1, y_2\}$, then:

$$\bigcup_{x \in S1} \bigcup_{y \in S2} g(x,y) \equiv \bigcup_{\substack{x \in S1 \\ y \in S2}} g(x,y) \equiv$$

$$g(x_1,y_1) \cup g(x_1,y_2) \cup g(x_2,y_1) \cup g(x_2,y_2)$$

5.3.1 Control Dependency

The concept of control dependency is introduced to model the sequence of statement executions within a program. Control dependency is entirely a property of the program's control structure in that it can be defined in terms of a **control-flow graph**. Thus, a program P can be depicted as a directed graph G(N,E) where N is a set of nodes corresponding to program statements and a set of directed edges E which illustrate the transfer of control through the program[2].

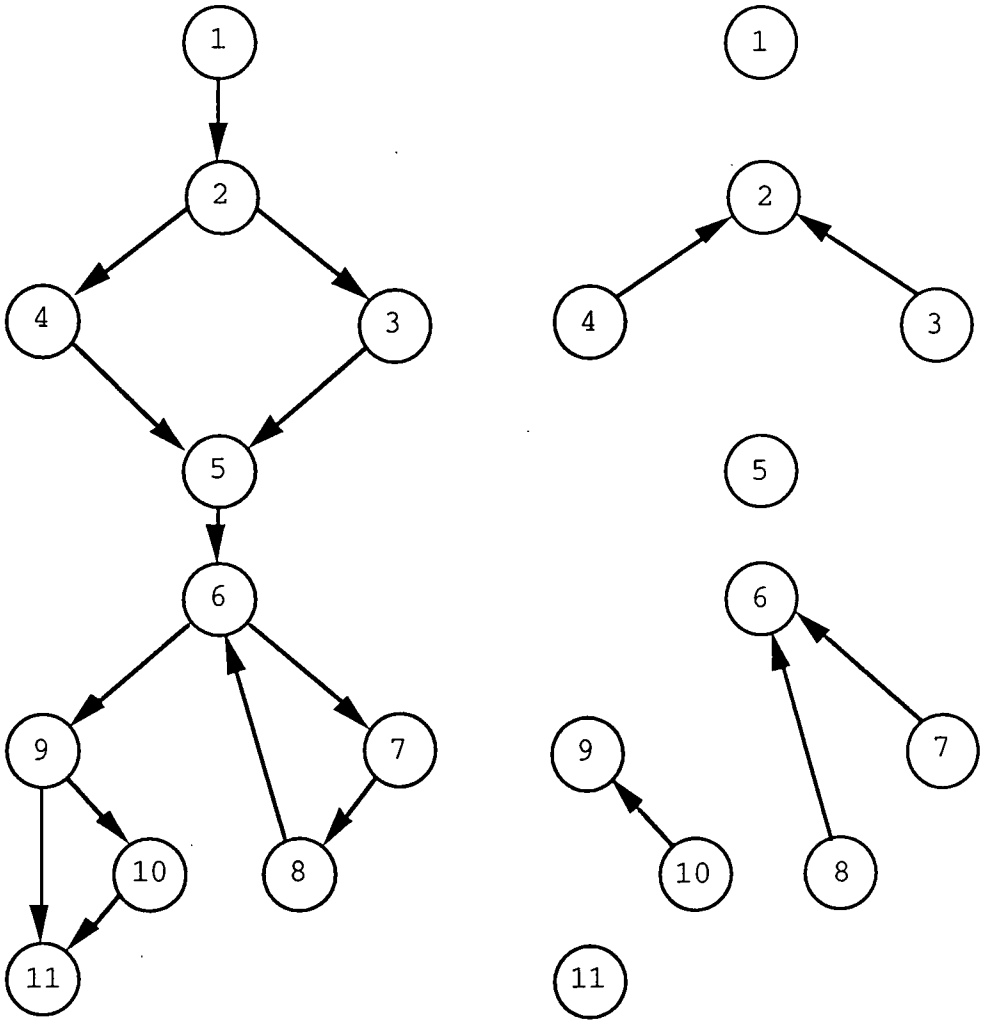
Consider the sample program in Figure 5.2, whose purpose is to compute the function $z=x^y$ where both x and y are integers. (Note that the line numbers in the leftmost column are to facilitate the discussion of the program and do not form part of the actual program.)

```
        program power
          begin
1 :          read(x,y);
2 :          if (y < 0) then
                   begin
3 :                     p = -y;
                   end
             else
                   begin
4 :                     p = y;
                   end
5 :          z = 1;
6 :          while (p <> 0) do
                   begin
7 :                     z = z * x;
8 :                     p = p - 1;
                   end
9 :          if (y < 0) then
                   begin
10 :                    z = (1/z);
                   end
11 :         write(z);
          end
```

Figure 5.2 : Dependency Analysis - An Example

[2] In some applications, the nodes of the control-flow graph correspond to **basic blocks**. For our purposes, it is more convenient to associate nodes with individual program statements.

Control dependency is defined as a relation on the graph G whereby an edge between node n and node m indicates that the execution of the statement at node n is dependent upon the outcome of the conditional statement at node m[3]; node n is said to be *dominated* by node m. This dominance relation, which has been formally defined by Ferrante *et al.* [63], is denoted in this thesis by the relation `ControlPred(n)`. Thus, for example, nodes 7 and 8 of the control-flow graph illustrated in Figure 5.3(a) are dominated by, and strongly control dependent upon, node 6.



(a) Control-Flow Graph (b) Control Dependency Graph

Figure 5.3 : Control Dependency

[3] It should be noted that node *n* possesses an outdegree of one.

A Control Dependency Graph or CDG of a program P is a tuple (N,C) where N is the same set of nodes as found in the control-flow graph G, but C represents the set of edges described by the the graph relation `ControlPred(n)`. Figure 5.3(b) illustrates the Control Dependency Graph and its formal definition is given in Figure 5.4.

Control Dependency Graph(P) =

let G(P) = (N,E),

$$C = \bigcup_{\substack{n \in N \\ m \in ControlPred(n)}} \{(n,m)\}$$

in (N,C)

Figure 5.4 : Formal Definition of a Control Dependency Graph

Consider, for example, the Control Dependency Graph illustrated in Figure 5.3(b) in which the `ControlPred(5)` = {} indicates that program statement 5 is executed unconditionally. In contrast, the relation `ControlPred(3)` = {2} shows that program statement 3 is dependent upon the boolean outcome of the predicate expression in program statement 2 and thus in its scope of control influence.

5.3.2 Data Dependency

The concept of data dependency is introduced to model the interaction of variables in programs. Although several types of data dependency are discussed in the literature, the description given in this thesis relates to *data-flow dependency* [7, 19, 62] which is dependent upon both the control structure and data structure of a program. The **data-flow graph** of a program P is established by augmenting the nodes of the control-flow graph G with variable usage information. Therefore, each node of a data-flow graph F(N,E) has a *use* set and a *def* set associated with it. The *use* set of a node in F consists of all program variables referenced during the computation associated with the node, while a *def* set consists of the variable computed at the node, if any. The data-flow graph

with its *def* set D and *use* set U is illustrated in Figure 5.5 and corresponds to the sample program presented in Figure 5.2.

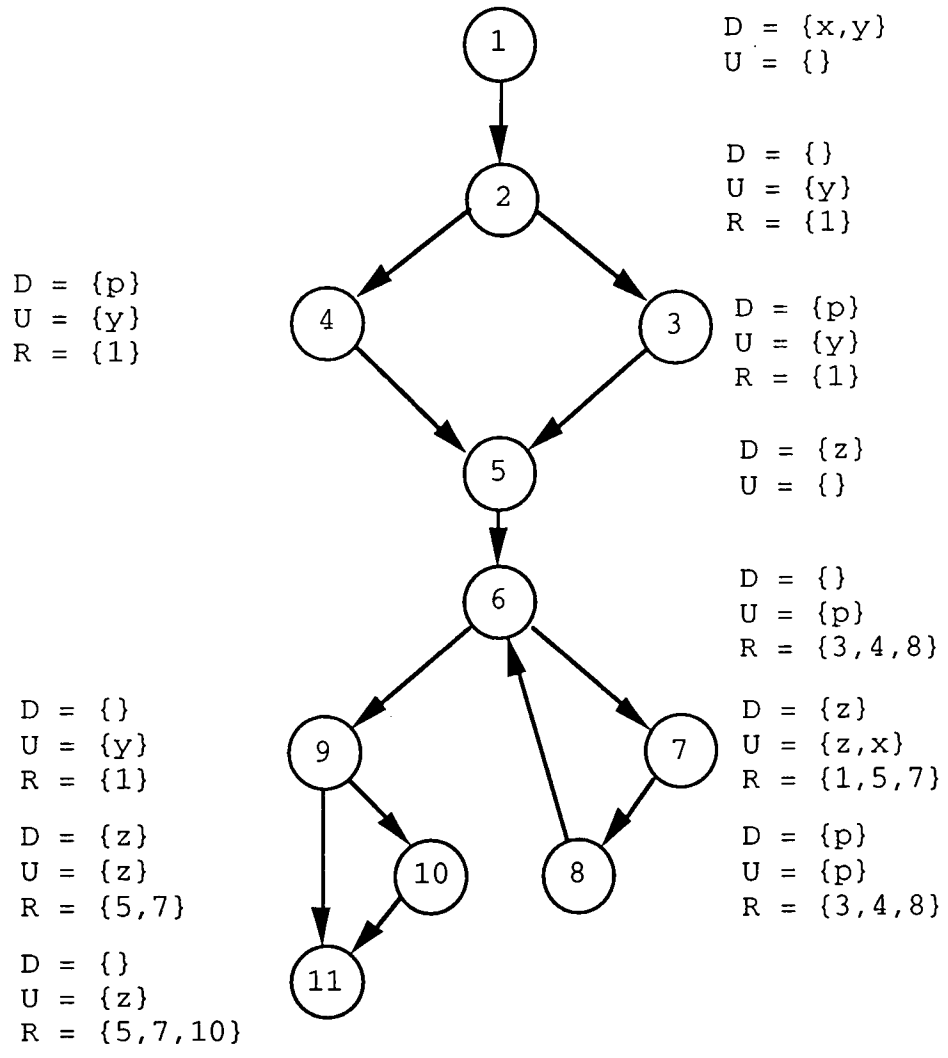


Figure 5.5 : The Data-Flow Graph

Data dependency is defined as a relation on the graph F whereby an edge between node n and node m indicates that the value of a variable *var* computed at node n is directly dependent upon the computation performed at node m. It assumes that the flow of control can transfer from node m to node n without encountering an intervening redefinition of variable *var* along any control-flow subpath between the two nodes in graph F.

In order to determine a set of nodes in F containing a variable definition which can *reach* the variable use at node n, a graph relation `ReachingDefinitions (var,n,F)` must first be defined; it is described formally in Figure 5.6.

```
ReachingDefinitions(var,n,F) =
let F = (N,E)
in ⋃(m,n) ∈ E if var ∈ def(m) then
                    {m}
                  else
                    ReachingDefinitions(var,m,(N,E-{(m,n)}))
```

Figure 5.6 : Formal Definition of the Reaching Definitions

In Figure 5.5, the *reaching definitions* set R for all nodes with nonempty *use* sets is shown alongside the *def* and *use* sets for each node. For example, the program statement 3 contains one reaching definition for the use of variable y without any intervening redefinitions of the variable. Thus, `ReachingDefinitions(y,3,F)` = {1}.

A Data Dependency Graph, or DDG, of a program P is a tuple (N,D) where N is the same set of nodes as in the data-flow graph F, but D describes the set of edges which reflect data dependency between the nodes in F. Figure 5.7 illustrates the Data Dependency Graph of the program and a formal definition is given in Figure 5.8. Data dependencies, which occur as a result of a program loop, are known as *loop-carried* data dependencies [63, 203]. They arise from the iteration of a loop; a variable referenced on the current iteration may have been assigned a value in the previous iteration[4]. For example, the program illustrated in Figure 5.2 contains a loop-carried dependency with respect to variable z at statement 7. In contrast, *loop-independent* data dependencies [63, 203] occur due to the program execution order and regardless of any loop iteration. An

[4] This is based on the assumption that the loop is executed at least twice.

example of this type of dependency exists between program statements 1 and 2 with respect to variable y.

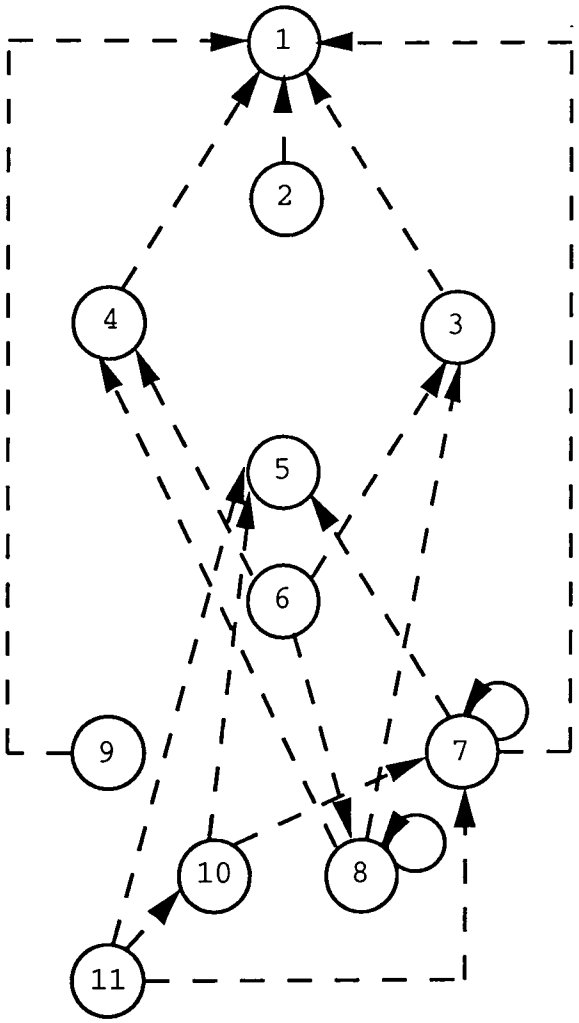


Figure 5.7 : The Data Dependency Graph

```
Data Dependency Graph(P) =
let F(P) = (N,E),

D = ∪n ∈ N                    {(n,m)}
        var∈use(n)
        m∈ReachingDefinitions(var,n,F(P))

in (N,D)
```

Figure 5.8 : Formal Definition of a Data Dependency Graph

5.3.3 Program Dependency Graphs

In Figure 5.9, the Program Dependency Graph, or PDG, for the sample program in Figure 5.2 is formed by combining its Control Dependency Graph (CDG) shown in Figure 5.3(b) and its Data Dependency Graph (DDG) shown in Figure 5.7.

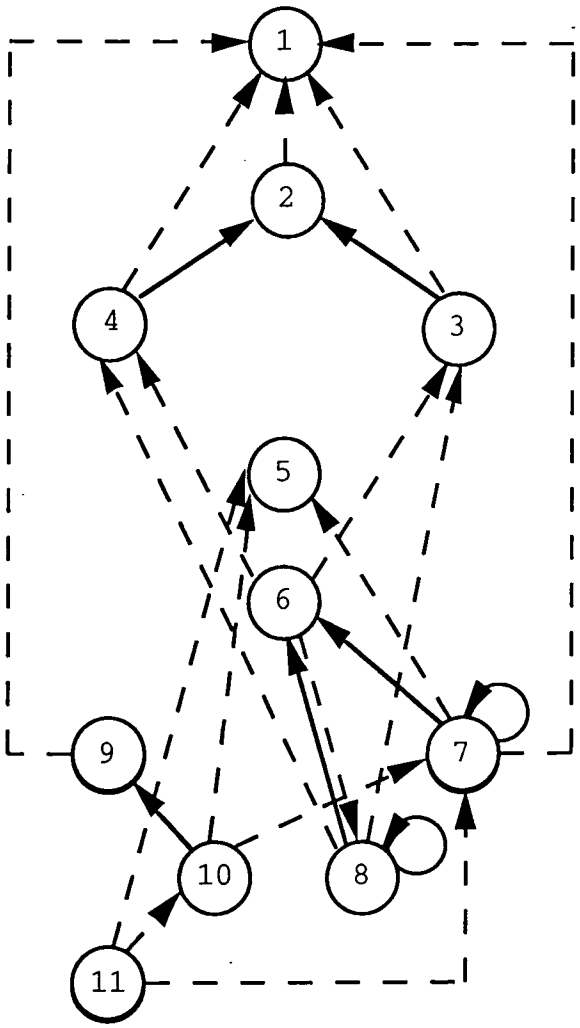


Figure 5.9 : The Program Dependency Graph

```
Program Dependency Graph(P) =

let Data Dependency Graph(P) = (N,D),
    Control Dependency Graph(P) = (N,C)
in (N, D ∪ C)
```

Figure 5.10 : Formal Definition of a Program Dependency Graph

The Program Dependency Graph, which is formally defined in Figure 5.10, can be used to describe the control and data dependencies which exist within individual program modules and small programs. However, dependency analysis could be extended to complete systems consisting of a single (main) module and a collection of supporting modules. By establishing the Program Dependency Graph for each module in the system and linking them together via their respective *calling dependencies* and *parameter dependencies*, a System Dependency Graph could be defined [115, 97]. However, considering the objectives of this thesis, which are to develop a change analysis technique based on the Program Dependency Graph, it is hoped that the development of a System Dependency Graph can be pursued in future research.

5.4 Language-Specific Dependency

The definition of data dependency, presented above, has provided a language-independent view of dependency analysis which accounts for data dependencies arising from the use of scalar variables. However, additional data dependencies may need to be considered when programs include the use of pointer variables and composite variables such as structures, unions and arrays. In the following section, therefore, such data dependencies are examined with respect to the C programming language. Consequently, the existing definition of data dependency is updated.

Modern programming languages, such as C, encourage the use of abstraction. For this purpose, they define a set of simple data types such as pointer variables, structures, unions, and arrays, from which programmers can then create their own structured data types. As in most programming languages, the C programming language enables variables to be defined which are nested, or contained, within each other. It defines a `struct` variable which consists of a set of fields or members. A similar construct is the `union` variable, which also contains a list of fields or members, but whose members actually occupy the same memory locations; they are effectively overlapping. In Pascal, the equivalent variables would be known as fixed and variant records.

In C, a pointer variable may be associated with a standard or abstract data type. Consider the C code fragment in Figure 5.11, where the pointer variable `p` of type `int` is declared alongside the integer variables `a` and `b`. In the course of program execution, the pointer variable `p` is assigned to the address of the integer variable `a`, and later the dereferenced pointer value `*p` is assigned to integer variable `b`. Although this is a rather contrived example in which a simple assignment `a=b` would have sufficed, it illustrates the effect of the dereferencing operator (*) and address operator (&) in C. Figure 5.11 shows the memory allocation made by a typical compiler for all variables along with their contents at each stage of program execution[5].

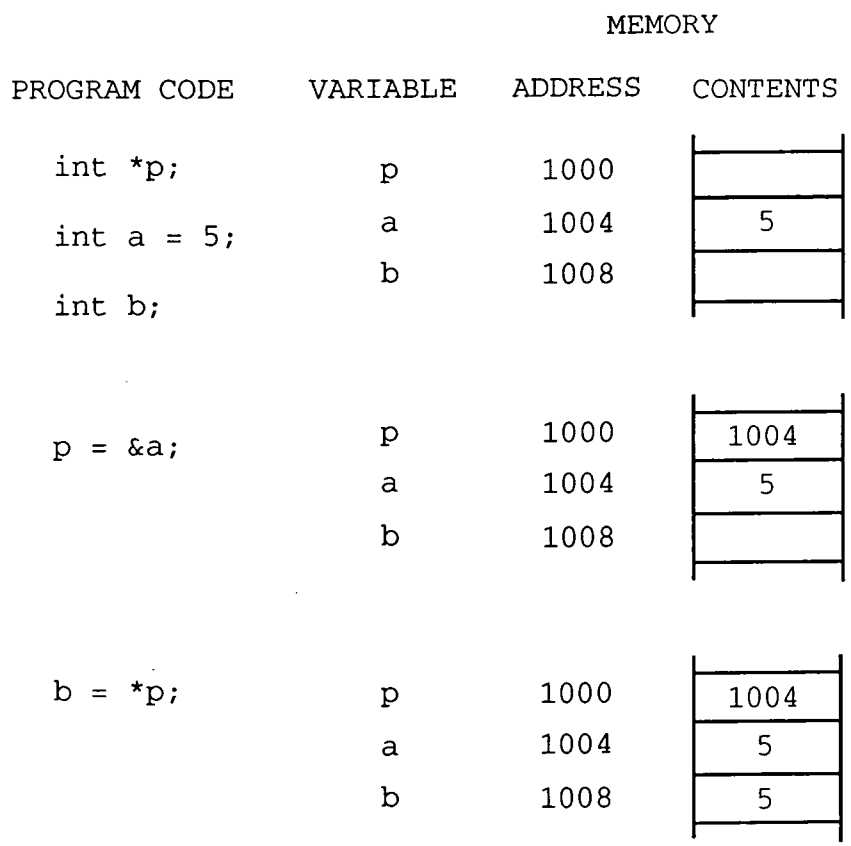


Figure 5.11 : Dereferencing a Pointer Variable

The use of pointer variables creates the situation in which code analysis cannot determine a unique memory location referred to by a pointer during program execution.

[5] Memory allocation may vary from compiler to compiler.

The *alias set* of a pointer expression at a program point is the set of all program variables to which the expression could refer at that point, as determined by code analysis. In Figure 5.11, for example, the assignment, `p = &a` implies that `a` is a member of the alias set of `p`. Thus, Figure 5.11 provides an indication of how the value of `a` could be indirectly modified, or accessed, via the dereferenced pointer variable `p`.

To facilitate the discussion of language-specific dependency, the notion of an *intersection* between two l-valued expressions is introduced. An expression is said to be an l-valued expression if a memory location can be associated with it. A simple check to determine whether an expression is an l-valued expression, or not, is to see if it can appear on the left hand side of an assignment statement. For example, expressions `var`, `A[i]`, `s.f`, `B[i].g`, `*p`, are all l-valued expressions. In contrast, none of the expressions `200`, `x + y`, or `a < b` is l-valued. The presence of pointer variables and composite variables, such as arrays, structures, and unions in a programming language requires that both *use* and *def* sets of statements be defined in terms of l-valued expressions.

A use expression e_1 is said to intersect with a def expression e_2 if the memory location associated with e_1 may overlap with that associated with e_2. To this end, three types of intersections can be defined: *complete intersection*, *partial intersection*, and *possible intersection.*

5.4.1 Complete Intersection

A *use* expression e_1 completely intersects a *def* expression e_2, if the memory location associated with e_1 is totally contained within that associated with e_2.

Consider the C code fragment in Figure 5.12(a), where the use of scalar variable `x` at program statement 2 completely intersects with its definition at statement 1. (Note that the

line numbers in the leftmost column are to facilitate the discussion of the programs and do not form part of the actual programs.)

```
void example()
{
     int x, y;

     ...
1:   x = 3;
     ...
2:   y = x;
     ...
}
```

(a)

```
void example()
{
     union {
          struct {
               short a,b;
          } s;
          long c;
     } test;

     ...
1:   test.c = 5;
     ...
2:   printf("%d",test.s.b);
     ...
}
```

(b)

Figure 5.12 : Examples of Complete Intersection

Complete intersections may also occur during the use of composite variables. In the case of `union` variables, for example, where the number of memory locations is governed by the member with the largest storage requirements, a complete intersection results when a member occupying fewer memory locations is referenced after a member with greater storage requirements has been defined. Figure 5.12(b) illustrates this relationship for the `union` members `test.c` and `test.s.b`.

5.4.2 Partial Intersection

If a *use* expression e_1 partially intersects with a *def* expression e_2, then the memory location associated with e_1 is partially contained within that associated with e_2. Thus, `PreExp(`e_1`,`e_2`)` describes the portion of the memory locations associated with e_1, which lie before those associated with e_2, and `PostExp(`e_1`,`e_2`)` refers to be the portion of memory locations which lie after those associated with e_2.

Consider the C code fragment in Figure 5.13(a), where the *ith* element of the character array x is defined at statement 1 and the entire array is then referenced at statement 2. (Note that the line numbers in the leftmost column are to facilitate the discussion of the programs and do not form part of the actual programs.)

```
void example()
{
     char x[4];
     int i;

     ...
1:   x[i] = 'a';
     ...
2:   printf("%s",x);
     ...
}
```

(a)

```
void example()
{
     union {
          char a,b,c,d;
          long e;
     } test;

     ...
1:   test.c = 's';
     ...
2:   printf("%ld",test.e);
     ...
}
```

(b)

Figure 5.13 : Examples of Partial Intersection

In this case, `PreExp(x, x[i]) = x[1..(i-1)]` and `PostExp(x,x[i]) = x[(i+1)..3]`. Another example of partial intersection is illustrated in Figure 5.13(b), where the union member `test.c` of type `char` is defined at statement 1, and correspondingly the union member `test.e` of type `long int` is used at statement 2. In this case, `PreExp(test.e, test.c) = test.a, test.b` and `PostExp(test.e,test.c) = test.d`.

5.4.3 Possible Intersection

A *use* expression e_1 is said to possibly intersect with a *def* expression e_2, if the memory location associated with e_1 partially, or completely, intersects with that associated with e_2. However, the decision as to whether or not they actually do intersect

is dependent upon program execution. In fact, possible intersections could only be resolved through the use of dynamic code analysis techniques [31, 123].

Consider the C code fragment in Figure 5.14(a), where the array element `x[i]` is defined at statement 1 and the element `x[j]` is used at statement 2. (Note that the line numbers in the leftmost column are to facilitate the discussion of the programs and do not form part of the actual programs.)

```
void example()
{
      char x[4];
      int i, j;
      ...
1:    x[i] = 'a';
      ...
      for(j=0;j<4;j++)
2:      printf("%c",x[j]);
      ...
}
```

(a)

```
void example()
{
      union {
            char a,b,c,d;
            long e;
      } test [10];

      int i, k;
      ...
1:    test[i].b = 't';
      ...
      for(k=0;k<4;k++)
2:      printf("%ld",test[k].e);
      ...
}
```

(b)

Figure 5.14 : Examples of Possible Intersection

Whether, or not, the use of `x[j]` at statement 2 intersects with the definition of `x[i]` depends on the actual values of program variables `i` and `j` at the time of program execution; the memory location accessed will vary from one loop iteration to another. However, if the two values should coincide, then the two expressions will form a complete intersection.

The situation may also arise, whereby a possible intersection and a partial intersection occur together. In this case, the possible intersections are analysed *before* partial intersections. This is illustrated in Figure 5.14(b), where the possible intersection

occurring between the array element `test[i]` at statement 1 and the array element `test[k]` at statement 2 takes precedence over the analysis of the partial intersection created between the two union members `test[i].b` and `test[k].e`.

However, the most significant influence upon the creation of possible intersections are pointer variables [146]. Consider the C code fragment in Figure 5.15(a). (Note that the line numbers in the leftmost column are to facilitate the discussion of the programs and do not form part of the actual programs.)

```
void example()
{
      int a, b;
      int *p;

      ...
1:    a = 3;
      ...
2:    b = 4;
      ...
      if(...)
           p = &a;
      else
           p = &b;
      ...
3:    *p = ...
      ....
4:       = a;
}
```

(a)

```
void example()
{
      int i, j;
      int *p;

      ...
      if(...)
      {
1:         y = ...
           p = &x;
      }
      else
           p = &y;
      ...
2:    *p = *p ...
      ...
3:       = y;
}
```

(b)

Figure 5.15 : More Examples of Possible Intersection

The intersection between the use of variable a at program statement 4 and its definition at statement 1 is dependent upon the dereferencing of pointer variable p at statement 3. At this statement, pointer variable p may or may not be accessing variable a; both variables a and b are considered to be members of the alias set of p. However, if the aliased variable a at statement 3 is redefined, an alternative intersection may arise with the use of a at statement 4.

A similar situation is illustrated in Figure 5.15(b), where a possible intersection exists between the definition of variable y at statement 1 and its corresponding use at statement 3 if pointer variable p were to be pointing to variable x; both variables x and y are considered to be members of the alias set of p. In addition, an intersection is possible between the same definition of variable y and the use at statement 2 due to y being an alias of pointer variable p.

The latter example, however, demonstrates the limitations of current algorithms for pointer aliasing, which compute the alias set of a pointer variable with respect to program statement [114, 202, 204], and shows the extent to which the accuracy of these algorithms can affect the number of possible intersections. If an analysis of program paths in Figure 5.15(b) is conducted, then it would be revealed that the program contains two paths which are mutually exclusive. The possible intersection with respect to variable y between statements 1 and 3 would, in fact, turn out to be a complete intersection as the pointer variable p would be pointing at variable x during program execution. It follows that the other possible intersection, which relates the definition of y at statement 1 to its use at statement 2, is erroneous.

The accuracy of dependency analysis could be improved through the use of path-sensitive aliasing algorithms. As a result, the number of possible intersections would be reduced and the number of complete intersections be increased. It is hoped that the development of path-sensitive aliasing algorithms can be pursued in future research.

5.4.4 Updated Definition

Let `CompleteIntersect`, `PartialIntersect` and `PossibleIntersect` represent boolean functions which determine if two l-valued expressions form a complete intersection, partial intersection, or possible intersection, respectively. With the definitions of control dependency and data dependency remaining the same, the graph relation `ReachingDefinitions` is updated to account for the language-specific

dependencies described above. Figure 5.16 defines the updated graph relation *ReachingDefinitions*.

```
ReachingDefinitions(var,n,F) =
let F = (N,E)
in ∪(m,n) ∈ E
     if def(m) = {} then
       ReachingDefinitions(var,m,(N,E-{(m,n)}))
     else
       let def(m) = {var}, E = E-{(m,n)}
       in if CompleteIntersect(var,var) then
                {m}
          else
          if PossibleIntersect(var,var) then
               {m} ∪ ReachingDefinitions(var,m,(N,E))
          else
          if PartialIntersect(var,var) then
               {m} ∪
               ReachingDefinitions(PreExp(var,var),m,(N,E)) ∪
               ReachingDefinitions(PostExp(var,var),m,(N,E))
          else
               ReachingDefinitions(var,m,(N,E))
```

Figure 5.16 : Updated Definition of the Reaching Definitions

5.5 Change Analysis

Schneidewind [229] notes that one of the difficulties experienced during software maintenance is the analysis of proposed program changes. Maintenance activities often require simple changes to be made to the program code which involve either the addition, deletion or modification of a program statement. They may, for example, include the alteration of a variable declaration, the modification of a computation in an assignment statement or the adjustment of a predicate value in a conditional statement. Such changes, however, can give rise to a number of problems.

A modification may need to be made to a variable use whose value has been defined using different declarations or assignments throughout the program. Thus, maintenance programmers may change the expression involving the use of the variable, while only being aware of *some* of the corresponding variable definitions. A change to the variable's value may then correct a problem occurring on one of the program paths leading up to the use, but at the same time introduce a new problem on another path.

Another frequent error made by maintenance programmers involves changing a variable assignment due to the assigned expression being incorrect and not ensuring that the new expression is appropriate for all the variable's uses. If the resulting variable assignment is used as part of several other computations, maintenance programmers may focus on only one or two of these uses and may not be aware of the value being used in other places.

These problems are compounded when the indirect effect, or **logical ripple effect** [270], of a program change is considered. For each set of statements involved in a maintenance operation (*primary error sources*), a further set of statements (*secondary error sources*) may be implicated; this is known as the *first-order* ripple effect. This process is continued when the secondary error sources become the primary error sources and implicate further statements, which causes a *second-order* ripple effect. The logical ripple effect therefore propagates throughout a program until no new secondary error sources are discovered.

In order to test modified program statements, the majority of selective revalidation techniques proceed with an inspection of the program's testing history. They aim to identify those test requirements, which are associated with the directly affected program statements[6], and then examine each of these test requirements to establish the corresponding set of retestable tests. Selective revalidation techniques based on data-flow

[6] Their approach to test selection is examined in Section 4.3.1.

testing examine only those test requirements (definition-use associations) in which either the variable definition or use forms part of the proposed modification. While this approach may ensure that the first-order ripple effect is analysed, the possibility that second-order ripple effects may arise as a result of the proposed modification is not considered. However, it is precisely these higher order ripple effects, which may cause further test requirements to be implicated and enable the retesting effort to focus on specific program paths affected by the changes.

In constrast, the selective revalidation techniques based on path testing require that whenever a particular program statement is modified, all test cases traversing that statement are rerun. This approach is based on the assumption that the computation performed in the modified statement is dependent upon the subcomputations on all paths leading to and from the statement. While this assumption may be valid in *some* cases, it does not hold for *all* cases. Thus, a computation may only affect or be affected by subcomputations on *certain* paths leading to or from the modified statement. To ensure that those test requirements affected by the proposed program modifications are systematically identified, a change analysis technique, based on the concept of program slicing [256], is developed.

5.5.1 Program Slicing

Program slicing is a form of program decomposition based on the extraction of information from the Program Dependency Graph. A program slice S, extracts from a program P, a sequence of statements in which the order of the statements in S is the same as in P. The slice S is obtained by selecting only those statements from P which conform to a slicing criterion C represented by an ordered tuple (*statement-range*, *variables*). For a particular slicing criterion, the value of *statement-range* is the range of statements over which a program is sliced and the value for *variables* represents some subset of variable identifiers which are visible in the given statement range. When a program is sliced in this

way, those statements which do not *affect* the value of one of the chosen *variables* are deleted, with the remaining statements forming the desired program slice.

Since the concept of a program slice was first introduced by Weiser [255, 256], two distinct forms of program slicing have evolved: *dynamic slicing* [3, 142, 143] and *static slicing* [171, 75]. Both forms of slicing have been applied to the testing [142, 144] and debugging [4, 47] of program code.

During testing and debugging, a program fault may occur and manifest itself as an error at the output of the program. Slicing techniques can then be applied to localise the fault by extracting those statements from the program which directly and indirectly affect variables *referenced* in an output statement; the techniques effectively trace program paths backward from an output statement. While dynamic slicing requires only those affected statements to be extracted which lie along the path traversed during the *actual* execution of a test case, static slicing extracts affected statements based on the *potential* execution of several test cases.

A number of static slicing algorithms have been defined in terms of graph reachability [203]. Conceptually, these algorithms are easier to understand than the one originally formulated by Weiser [256] and also more efficient when several slices at a time are required. For the purposes of fault localisation, a *static* slice can be extracted from a program *P* based on the knowledge that a variable *var* is referenced at a statement or node *n* in the Program Dependency Graph. Thus, a graph relation `Program Slice`(P,var,n) can be defined in terms of reaching definitions and reachable nodes; all reaching definitions of variable *var* at node *n* are identified and each reaching definition used to determine all other reachable nodes in the Program Dependency Graph. A formal definition of the program slice is presented below in Figure 5.17.

```
Program Slice(P,var,n) =
let F = (N,E), D = Program Dependency Graph(P)
in ⋃ m∈ReachingDefinitions(var,n,F) ReachableNodes(m,D)
```

Figure 5.17 : Formal Definition of a Program Slice

The above definition incorporates a graph relation `ReachableNodes(n,S)` for determining the set of nodes in a directed graph S, which can be reached from node *n* by following one or more edges in S. In fact, this relation represents the transitive closure of node *n* in graph S.

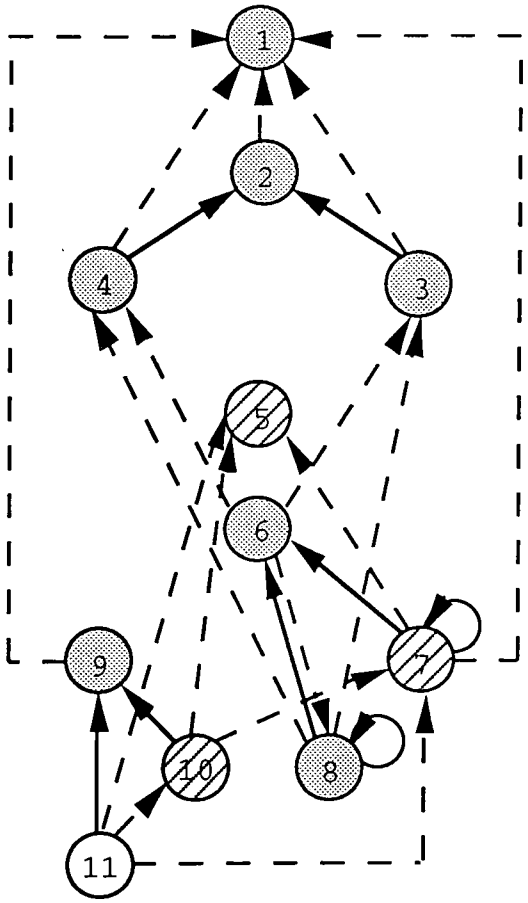
```
ReachableNodes(n,S) =
let S = (V,A)
in {n} ∪ ⋃ (n,m)∈A ReachableNodes(m, (V,A-{(n,m)}))
```

Figure 5.18 : Formal Definition of Reachable Nodes

Consider the sample program illustrated in Figure 5.19(a). If a slice is to be taken with respect to the variable z at the output statement (11) in the program, then according to the above definition all other program statements would be extracted as each one of them directly or indirectly affects the value of z at statement 11; `Program Slice`(P, z, 11) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. The statements, which are initially identified as reaching definitions of z, are depicted in Figure 5.19(b) as the *striped* nodes, while the remaining reachable nodes are indicated by the *shaded* nodes in the Program Dependency Graph.

```
program power
     begin
1:         read(x,y);
2:         if (y < 0) then
             begin
3:             p = -y;
            end
           else
            begin
4:             p = y;
            end
5:         z = 1;
6:         while (p <> 0)
do
            begin
7:            z = z * x;
8:            p = p - 1;
            end
9:         if (y < 0) then
            begin
10:            z = (1/z);
            end
11:        write(z);
```

(a)



(b)

Figure 5.19 : Example of a Program Slice

5.5.2 New Approach

The change analysis technique developed in this thesis defines a number of graph relations which utilise the Program Dependency Graph in order to identify the effects of a proposed modification. Their objective is to extract only those statements from a program which are directly and indirectly related to the proposed change. Based on these statements, a set of test requirements is chosen which influences the choice of retestable tests during test selection[7]. A number of benefits accrue from the development and use of this change analysis technique; fewer faults are introduced into the program during the changes and the growth rate of program complexity is reduced due to a greater

[7] A procedure for test selection is described in Section 6.5.

understanding of the effects of maintenance. More importantly, however, program changes can be implemented and *tested* more systematically as only those test cases in the test suite, which traverse the set of affected test requirements, are selected for rerunning.

A technique for change analysis is described, which is based on *static* slicing and can thus be used to examine the potential effects of a proposed program modification. Emphasis is placed upon developing a technique which allows a slice S to be extracted with respect to a *modified* statement n occurring anywhere in a program P. This is in constrast to the approach taken during program debugging where a slice is constructed with respect to an *output* statement. Slicing, as part of change analysis, also requires the variable definitions (*def* set) and uses (*use* set) of a modified statement to be examined instead of just the variable uses in an output statement.

A further consideration, when adapting static slicing for the purposes of change analysis, is the nature of its definition: a program slice must form an executable program in itself. Therefore, a slice needs to consider both the control dependency *and* data dependency of a program. As a result, the slice may contain program statements which are not directly relevant to a modified program statement, but are included as they represent a subset of the program's original control structure. For example, a slice taken with respect to the variable z during a proposed modification of statement 10 in the program in Figure 5.19(a) would include two reaching definitions from statements 5 and 7. Of these two statements, statement 7 is control-dependent upon statement 6; thus, statement 6 forms part of the slice. By examining the data dependencies involving statement 6, it is found that statements 3 and 4 are implicated which, in turn, are in the scope of control influence of statement 2; thus, statement 2 is also included in the slice, and so forth. Of a total of eight statements (1-8) in the slice, five statements (2, 3, 4, 6 and 8) are due to the control dependency of the program being considered.

Unlike slicing, the change analysis technique developed in this thesis is only concerned with the program's *data dependency*; the only control dependency information

being considered is expressed implicitly by the data dependency. Instead of examining both the affected statements *and* their associated control structure to determine a set of potentially affected program paths, the change analysis technique concentrates on extracting statements, which directly and indirectly affect the value of the modified variables, and assumes that these statements will be traversed by some subset of the affected paths. The corresponding graph relation thus identifies a considerably smaller number of affected statements; for example, it would only implicate statements 1, 5, 7 and 10 in response to a proposed modification of statement 10.

The traditional role of program slicing as an aid to debugging has meant that static slicing algorithms have focused on the *backward* data-flow problem. Therefore, any slicing algorithm adapted for the purposes of change analysis would only be able to identify those statements in a program which contain definitions, and subsequent uses and definitions, affecting a modified variable use. However, as part of developing a technique for change analysis, new graph relations are defined to also address the *forward* data-flow problem. These enable the technique to not only identify those statements which *affect* a modified statement, but also those statements which are *affected by* a modified statement. Therefore, the technique identifies those statements containing uses, and subsequent definitions and uses, affected by a modified variable definition.

In particular, the change analysis technique has to consider the effects of a proposed modification on conditional statements as any change to the value of an associated predicate expression will cause an alteration in the path condition. Whenever a predicate expression undergoes an explicit change or its value is affected by program modifications, those data dependencies, which straddle the modified conditional statement or lie on a path that traverses the affected conditional statement, need to be examined[8]. Therefore, program statements are considered as affected if the data dependency connecting them relates to a variable definition being encountered prior to the

[8] The variables associated with these data dependencies may not necessarily be related to the variables in the predicate expression.

conditional statement and the corresponding variable use lying beyond, but in the scope of control influence of, the conditional statement. Furthermore, those statements, which are control-dependent upon the conditional statement and are associated with any data dependencies, need to be analysed.

The change analysis technique is described by a new graph relation `Change(P,var,n)`, which can address both the forward and backward data-flow problems, and is formally defined in Figure 5.20(a). It enables a set of affected program statements to be extracted from a program *P* based on the knowledge that a modified variable *var* is either defined or used at the statement (node) *n* in the Program Dependency Graph. The set of program statements, which are obtained, can then be used to determine the corresponding set of test requirements.

The initial step of the change analysis technique determines whether the affected variable is being analysed in the context of a directly modified computational or conditional statement. To achieve this, a relation `ControlSucc` is used to determine whether a node m *dominates* a node n. This relation, which has been formally defined by Ferrante *et al.* [63], represents a transposed `ControlPred` relation. If a computational statement is being examined, for which the `ControlSucc` relation returns no control-dependent statements, then the change analysis technique establishes whether the modified variable is defined or used. It initialises the analysis of either the backward or forward data-flow problem by using the existing graph relation `ReachingDefinitions` and its transposed relation `ReachingUses`, respectively, to determine the corresponding definitions or uses. The latter relation, which is formally defined in Figure 5.21(a), is used to identify those variable uses that can be reached from a directly modified variable definition. In the case of a directly modified conditional statement, the technique analyses the backward data-flow problem as well as those data dependencies which may be associated with statements in its scope of control influence.

```
Change(P,var,n) =
let F   = (N,E),
    D   = Data Dependency Graph(P),

in if var ∈ def(n) then

        {n} ∪ ∪ m∈ReachingUses(var,n,F) Ripple(m,n)
    else
    if ControlSucc(n) = {} then

        ∪ m∈ReachingDefinitions(var,n,F) ReachableNodes(m,D)
    else
        {n} ∪

        ∪ m∈ReachingDefinitions(var,n,F) ReachableNodes(m,D) ∪

        ∪m∈ControlSucc(n) Ripple(m,n)
```

(a) The Change Relation

```
Ripple(a,b) =
let F   = (N,E),
    D   = Data Dependency Graph(P),
    var ∈ def(a),

in if ControlPred(a) = {} then
        if ControlSucc(a) = {} then

            {a} ∪ ∪ c∈ReachingUses(var,a,F) Ripple(c,b)
        else
            {a} ∪ ReachingNodes(a,b) ∪

            ∪c∈ControlSucc(a) Ripple(c,b)

    else
        if ControlSucc(a) = {} then
            {a} ∪ ReachingNodes(a,b) ∪

            ∪ c∈ReachingUses(var,a,F) Ripple(c,b)
        else
            {a} ∪ ReachingNodes(a,b) ∪

            ∪c∈ControlSucc(a) Ripple(c,b)
```

(b) The Ripple Relation

Figure 5.20 : Formal Definition of the Change Analysis Technique

The change analysis technique is supported by the new graph relation `Ripple` which recursively analyses the implicated statements. Emphasis is placed on distinguishing between computational and conditional statements as well as nested and unnested statements. During the analysis of the backward data-flow problem, the `ReachingNodes` relation, defined in Figure 5.21(b), ensures that only those statements are identified which lie on the same control-flow path as the originally modified statement and thus may be affected by any changes made to it.

```
ReachingUses(var,m,F) =
let F = (N,E)

in ∪(m,n) ∈ E  if var ∈ use(n) then
                  {n}
               else
                  ReachingUses(var,n,(N,E-{(m,n)}))
```

(a) Formal Definition of the Reaching Uses Relation

```
ReachingNodes(a,b) =
let F   = (N,E),
    D   = Data Dependency Graph(P),
    var ∈ use(a),

in if c ∈ ReachingDefinitions(var,a,F) andalso
      b ∈ ReachableNodes(c,F) orelse
      c ∈ ReachableNodes(b,F) then

        ReachableNodes(c,D)
   else
        {}
```

(b) Formal Definition of the Reaching Nodes Relation

Figure 5.21 : Formal Definitions of the Supporting Relations

5.5.3 Application

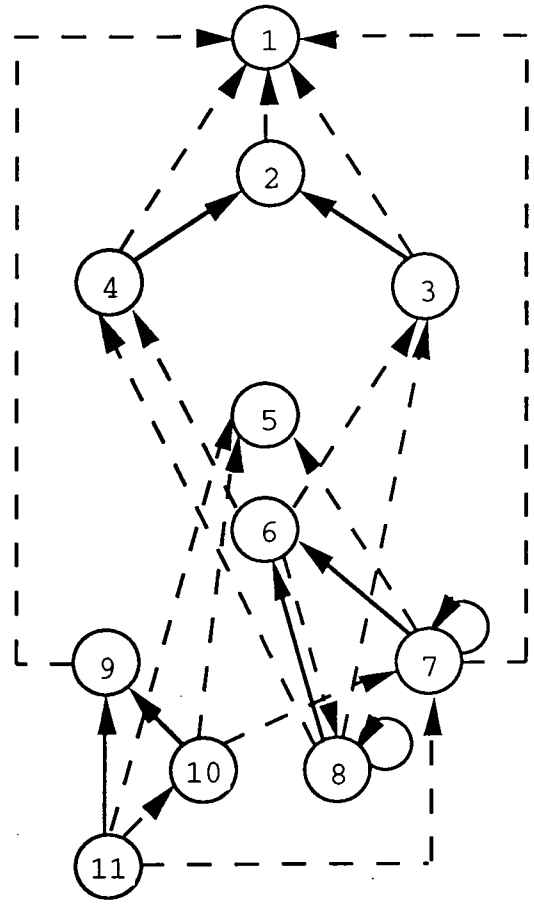
During the maintenance phase, change proposals detail the modifications to be made to the program specification, design and code. With respect to the implementation and, in

particular, individual program modules, a list of basic modifications comprises the addition, deletion and alteration of program statements. As a result of these modifications, the control and data dependency in the existing Program Dependency Graph may need to be updated, with some of the current dependencies being removed and new dependencies being created. With the help of *incremental* code analysis techniques, it would be possible to transform the existing Program Dependency Graph into a new Program Dependency Graph to reflect the updated program structure. While it is hoped that the development of incremental code techniques can be pursued in future research, this thesis assumes their existence for the purposes of demonstrating the application of the change analysis technique.

Whenever a program statement is added, deleted, or modified, the change analysis technique is applied with respect to each affected variable definition and use in that particular statement. While the analysis of any inserted statement requires the technique to utilise the *updated* Program Dependency Graph in order to examine its new variable definitions or uses, the latter types of modification require their deleted or affected definitions and uses, respectively, to be examined. Their analysis is based upon the *existing* Program Dependency Graph. The change analysis technique therefore ensures that those program statements, which directly or indirectly contribute to the computation performed by the modified statement and also rely upon the value computed by that statement, are determined. As a result, the retesting effort focuses on those test requirements, and subsequently test cases in the test suite, which previously traversed the modified statement and may now help to expose any errors induced by the program changes.

In order to demonstrate the effectiveness of the change analysis technique, modifications are suggested for the sample program and associated Program Dependency Graph illustrated in Figure 5.22. The list of definition-use associations are included so that comparisons with other selective revalidation techniques based on data-flow testing can be made in terms of the number of affected test requirements being selected.

```
program power
     begin
1:         read(x,y);
2:         if (y < 0) then
             begin
3:             p = -y;
            end
           else
            begin
4:             p = y;
            end
5:         z = 1;
6:         while (p <> 0) do
            begin
7:            z = z * x;
8:            p = p - 1;
            end
9:         if (y < 0) then
            begin
10:            z = (1/z);
            end
11:        write(z);
     end
```




```
(y, 1, (2,3))   (y, 1, 3)  (y, 1, (9, 10))
(y, 1, (2,4))   (y, 1, 4)  (y, 1, (9, 11))
(p, 3, (6, 9))  (x, 1, 7)  (p, 3, (6, 7))
(p, 8, (6,9))   (p, 3, 8)  (p, 4, (6, 7))
(p, 8, (6,7))   (p, 4, 8)  (p, 4, (6, 9))
(p, 8, 8)       (z, 5, 7)  (z, 5, 10)
(z, 7, 7)       (z, 7, 10) (z, 5, 11)
(z, 7, 11)      (z, 10,11)
```

Figure 5.22 : Applying the Change Analysis Technique

For a proposed program modification to statement 10, the change analysis technique is applied with respect to both the definition and use of the variable z. In a backward slicing operation, those variable definitions, and subsequent uses and definitions, are considered which may possibly affect the value of z; thus, statements 1, 5 and 7 are identified. A forward slicing operation then determines those uses, and any subsequent definitions and uses, possibly affected by a change to the value of z; thus, statement 11 is flagged as affected. From this set of affected statements, produced by the graph relation

Change(Power, z, 10) = {1, 5, 7, 10, 11}, the following list of test requirements or definition-use associations can be identified: (z, 5, 10), (z, 7, 10), (z, 10, 11), (z, 5, 7), (z, 7, 7) and (x, 1, 7).

Another example demonstrates the application of the change analysis technique to a proposed modification of the predicate expression in statement 9. This analysis is more complex than for the computational statement being considered earlier as it involves the explicit change of a conditional statement. First, data dependencies are traced which are related to the affected variable y in the predicate expression. As a result, statement 1 is identified as affected. Second, those data dependencies are analysed in which a variable definition is encountered prior to the affected conditional statement and the corresponding variable use lies beyond, but in the scope of control influence of, the conditional statement. This reveals that, in particular, data dependencies relating to the variable z straddle the modified conditional statement. Subsequently, statements 5, 7 and 10 are included in the set of affected statements. Finally, statement 11 is identified during a forward slicing operation on statement 10 which lies within the scope of control of statement 9. From the set of affected statements, produced by the graph relation Change(Power, y, 9) = {1, 5, 7, 9, 10, 11}, the following list of test requirements or definition-use associations can be identified: (y, 1, (9,10)), (z, 5, 10), (z, 7, 10), (z, 10, 11), (z, 7, 7), (z, 5, 7) and (x, 1, 7).

If the program had initially been validated using data-flow testing techniques and revalidated using the corresponding selective revalidation technique[9], then the above examples would have resulted in the following definition-use associations being identified: (y, 1, (9,10)), (z, 7, 10), (z, 5, 10), (z, 10, 11) and (y, 1, (9,10)), (y, 1, (9,11)), (p, 3, (6,9)), (p, 4, (6,9)), (p, 8, (6,9)). This approach ensures that those test requirements, which are associated with the immediately affected data dependencies of the variables z and y, are considered for test selection. In both cases, the retesting effort

[9] This technique requires only those definition-use associations to be selected in which the modified statement contains either a definition or use with respect to the modified variable

is limited to certain subpaths along which the variable z and y are defined and used, respectively. However, with the change analysis technique, developed in this thesis, the scope of the retesting effort is extended to include those subpaths in the program along which the modified variables *and* any affected variables may possibly have been defined or used.

The above examples have highlighted the important role of a change analysis technique in systematically determining the extent of a proposed program modification. Although this approach often results in a larger number of test requirements being implicated than for other selective revalidation techniques, in particular, data-flow based techniques, it also ensures a more thorough approach to selective revalidation. Together with the operations research techniques described in Chapter 6, the change analysis technique can have a significant influence on the process of test selection.

5.6 Summary

In this chapter, code analysis techniques are developed based on a graph theoretic approach which depicts a program's control dependency and data dependency information in terms of nodes and edges in a directed graph. Two aspects of code analysis are examined in the context of selective revalidation: dependency analysis and change analysis.

For any maintenance operation, programmers need to gain a general understanding of how the program works, together with knowledge about which sections of code are important to the maintenance operation. Thus, dependency analysis is responsible for identifying the dependencies between different entities in a program. Apart from investigating the dependencies arising between scalar variables in a program, the analysis also examines the dependencies expressed by composite and pointer variables with particular reference to the C programming language.

Maintenance programmers need to assess the extent of the modifications by considering both the direct and indirect influences of any changes. Therefore, a new change analysis technique, which utilises the control and data dependency within a program in order to extract the affected statements, and subsequently test requirements from the program's testing history, is developed.

Chapter 6

Test Suite Management

6.1 Introduction

The management of a test suite during maintenance requires solutions to be found to the problems of test selection and test update. In response to these problems, existing strategies for selective revalidation have developed procedures based on revalidation criteria which neither determine a minimal set of retestable test cases, nor identify any redundant tests. Moreover, these procedures are usually restricted to a particular testing phase or testing strategy.

However, a systematic and efficient solution to the problems of test suite management may be found in the application of operations research. With operations research, the problems of test selection and test update are formulated as decision problems and solved using optimisation methods. As a result, procedures for test selection and test update can be developed which select a minimal set of retestable tests and identify any redundant tests.

A further benefit of operations research is that the decision models, established as part of the formulation, allow different testing objectives and constraints to be considered. The problems of test suite management can therefore be solved independently of any testing strategy or testing phase. In this thesis, emphasis is placed on developing procedures for test selection and test update to enable the selective revalidation of the program code. The procedures are applied with respect to a test suite which may consist of both functional and structural test cases, but where the selection and update of test cases is undertaken only on the basis of their *structural* test coverage.

In this chapter, solutions are presented to the problems of test suite management. Section 6.2 introduces the subject of operations research by explaining important terminology and describing its application with respect to software testing. In Section 6.3, the problems of test selection and test update are formulated in terms of decision problems. The corresponding decision models are described in terms of generalised objectives and constraints. Section 6.4 explores the algorithms which can be used to solve these decision models. Particular emphasis is placed upon examining approximation algorithms and describing them in terms of their solution size and run-time complexity. A new heuristic method is then developed to address the limitations of existing algorithms. Finally, Section 6.5 describes selective revalidation based on the technique developed in this thesis. Details are given, concerning procedures for test selection and test update, which rely upon the use of the new change analysis technique and new heuristic method described in this thesis.

6.2 Operations Research

The subject of operations research has been discussed extensively in the literature [110, 237]. Therefore, the aim of the following discussion is not to provide a complete reference to the subject, but instead to concentrate only on those aspects of operations research which have influenced the research directions of this thesis.

Although many different definitions of operations research exist, a set of common denominators can be established to describe it. Operations research defines: a) the solution of problems relative to the attainment of specified objectives or criteria, b) the identification of alternative solutions, c) the optimisation, or selection, of the best alternative for the stated criterion, and d) the provision of a system perspective in which a tendency exists to consider the interrelationship of components in their environment rather than as separate entities.

In recent years, significant advances in the development of computer hardware have made the practical application of operations research possible and the development of optimisation methods for solving of large and complex problems a viable proposition. A wide range of applications, including telecommunications networks, transportation systems, emergency services, and utilities have benefitted as a result [169]. However, an increasing number of disciplines within computer science have used operations research.

In the field of software engineering, the subject of software testing has seen the extensive application of operations research. For example, in structural testing, path selection strategies are needed to guide users in the selection of an optimal set of test cases. A number of these optimal path selection strategies have been developed based on techniques adopted from operations research [55, 145, 192, 211]. They result in a minimal set of test cases being developed in order to traverse paths through the program code and exercise every test requirement at least once; thus satisfying the associated test coverage criterion.

However, structural testing strategies require infeasible paths in the program to be identified. Thus, where a particular path cannot be executed, an alternative path must be sought to ensure that the associated test coverage criteria are satisfied. Coward [53] and White-Sahay [264] describe methods for the identification of such infeasible paths in the program code. Again, techniques adopted from operations research are used to solve the problem of path feasibility. The solution of the problem indicates whether, or not, a feasible path through the code exists and, if so, identifies which of the program variables must be used as input variables in order to exercise the feasible path.

Other problems in software testing involve the elimination of redundant test cases during development testing. For example, Ince-Hekmatpour [125, 126] describe an empirical evaluation of *random testing* in which operations research is used to select a subset of test cases from the original set of test cases and yet achieve the same structural test coverage as the original set.

The subject of operations research encompasses a large number of optimisation methods [110]. In this thesis, however, emphasis is placed on describing optimisation methods based on *linear programming* [112]. A linear programming model consists of two parts: a) an objective function **Z**, which forms a linear function f(**x**) with respect to a set of decision variables, $\mathbf{x} = x_1, x_2, ..., x_n$, and b) a set of constraints in **x** in which each constraint represents a linear function g(**x**) such that $g(\mathbf{x}) \leq b$, $g(\mathbf{x}) \geq b$, or $g(\mathbf{x}) = b$, where b is a constant. By solving a linear programming model, a set of decision variables in **x** is identified, which aim to either maximise or minimise the objective function **Z**. In this thesis, however, the aim of any linear programming model is to minimise the objective function. The standard form of a linear programming model (LP) is:

minimise	$Z = c^T x$	(6.1)
subject to	$Ax \geq b$	(6.2)
and	$x \geq 0$	(6.3)

where A is the constraint matrix of order [m x n], c is an [n x 1] cost matrix, b is an [m x 1] requirements matrix, and x represents the set of decision variables of size n components. The superscript T denotes the vector transpose.

In this thesis, a linear programming problem known as **integer programming** [184, 235, 236] is examined. Its corresponding model differs from linear programming in that it specifies an objective function, decision variables, and constraint coefficients, all of which are represented by integers. The standard form of an integer programming model (IP) is:

minimise	$Z = c^T x$	(6.4)
subject to	$Ax \geq b$	(6.5)
and	$x \geq 0, x \in Z$	(6.6)

Some integer programming problems have the special feature that their decision variables are restricted to only one of two values: zero or one. In effect, the decision variables represent *binary* variables. Therefore, this type of integer programming problem is referred to a binary (zero-one) programming problem [77]. The standard form of a binary programming model (BP) is:

minimise	$Z = c^T x$	(6.7)
subject to	$Ax \geq b$	(6.8)
and	$x \in \{0,1\}$	(6.9)

The *set-covering problem* refers to a special instance of the binary programming problem where the constant b, which is associated with every constraint, is represented as a vector of ones. In fact, the problems of test selection and test update can be described in terms of such a set-covering problem which can be stated as follows:

Given: a test suite TS containing a list of test requirements $R = \{r_1, r_2, ..., r_i\}$ for a program, and a collection of testing subsets $T = \{T_1, T_2, ..., T_i\}$, in which one T_i is associated with each test requirement r_i $(1 \leq i \leq m)$, and each T_i comprises a set of test cases $t = \{t_1, t_2, ..., t_j\}$ and associated cost factors c_j such that at least one test case t_j $(1 \leq j \leq n)$ from T_i can be used to exercise the test requirement r_i.

Find: the representative set of test cases t_j, which at minimum cost, exercise every test requirement r_i.

The test requirements r_i may represent those test requirements which correspond to the modified portions of a program (test selection) or all test requirements (test update) of a program. The set of test cases t_j must contain at least one test case from each subset T_i. Such a set is referred to as the *cover* of T. To find the representative set of test cases t_j requires a cover of minimum cardinality[1] to be found for the collection of subsets T_i.

[1] The cardinality of a finite set represents the number of elements contained in the set.

However, the problem of finding such a cover has been shown to be NP-complete. The notion of **NP-completeness** [76, 174] is the basis of a theory allowing a class of problems, such as integer programming and set-covering, to be identified for which no efficient, polynomial-time algorithm is likely to exist. As a result, it cannot be predicted with a degree of certainty whether a solution to such problems is attainable[2]. However, this does not detract from the fact that such problems need to be solved. In practice, two types of algorithms, which are discussed in Section 6.4, are examined to solve such problems.

In terms of the binary programming model, described by Equations 6.7-6.9, a set-covering problem is defined as: a) an objective function Z, which aims to obtain a cover of minimum cardinality; b) a cost vector c of size [n x 1], which represents a cost factor c_j associated with each test case t_j; c) a set of decision variables x of n components, which relate to each test case t_j in the test suite; and d) a constraint matrix $A = [a_{ij}]$, which represents an [m x n] matrix of test requirements r_i versus test cases t_j. Thus, the rows of A correspond to the elements of R, while columns of A correspond to the test cases t_j in each T_i. The value of a_{ij} is set to one, if test requirement r_i is exercised by test case t_j; otherwise it is zero. Thus, R_i denotes the *ith* row of A and C_j denotes the *jth* column of A.

6.3 Test Selection and Test Update

The idea of formulating the problems of test suite management as set-covering problems was first proposed by Fischer [65, 66]. However, his work concentrated upon the problem of test selection, with the resulting decision model being characterised by a limited set of objectives and constraints which did not consider all of the important aspects of test suite management. Furthermore, the algorithm used to solve the decision model was not suitable for solving large decision models. In this thesis, the work by

[2] This fact is ascertained regardless of the problem size.

Fischer has been extended to address not only the problem of test selection, but also that of test update. Decision models are developed in which objectives and constraints can be generalised in order to better reflect the important characteristics of the test suite management problem. These single-objective models are then evolved into multiple-objective models which reflect situations in which the selection or update of test cases in the test suite needs to be optimised with respect to a number of different objectives, simultaneously.

6.3.1 Generalised Objectives

The most common objective for test selection or test update requires an optimal set of test cases to be determined, either to validate the modified portions of the program or the entire program, respectively. Therefore, a decision model is formulated in which an objective function Z is specified with a *unit* cost vector c to indicate that the same resources are associated with each test case in the test suite.

However, alternative objectives can be specified to more accurately reflect the problems of test suite management. For example, a set of test cases may need to be selected or maintained at minimum cost. A number of cost factors [159] have been identified for use in the decision model; the *test execution cost* includes the cost, time, and effort required to set up the testing environment and execute each test case, and the *result analysis cost* involves collecting the output of each test case, comparing it with any previous output, and verifying it. As a result, a decision model can be formulated in which an objective function Z is specified with a cost vector c which reflects the test execution cost and result analysis cost associated with each test case in the test suite.

Selective revalidation may also be performed based on the subjective qualities of a test suite. Thus, a test suite may be associated with a priority scale in which those test cases exercising a particularly critical piece of program functionality or code are given a different priority from those test cases exercising less critical aspects. Consequently, a

decision model can be formulated in which an objective function Z is specified with a cost vector c which reflects the priority associated with each test case in the test suite; its aim being to select the most important test cases.

6.3.2 Generalised Constraints

Apart from being able to specify different objectives, a decision model includes a set of constraints which reflects the testing history of a program, with each constraint representing a test requirement and the test cases exercising it. While in this thesis, the test requirements are restricted to those specified by *structural* testing techniques, it should be noted that it is possible for the constraints to represent test requirements specified by other testing strategies.

For test selection, the set of constraints is confined to those test requirements which exercise statements directly and indirectly affected by the program modifications. In order to determine these constraints, the change analysis technique, described in Section 5.5, is used. It assesses the impact of the proposed changes on the existing program code and then selects the corresponding test requirements and test cases from the program's testing history. For test update, the set of constraints reflects the complete testing history of a program.

In each decision model, a set of constraints is established based on the constraint matrix A, which reflects the testing history of a program, and a requirements matrix b. The latter matrix usually consists of a unit vector to indicate that at least one test case in the test suite exercises the test requirement represented by the corresponding constraint. However, situations may arise where particular values in the requirements matrix need to be changed. Critical sections of program code can be identified in the testing history by the large number of test cases which traverse their corresponding test requirements. Whenever these sections of code are affected by the proposed modifications, they may need to be retested more thoroughly than other parts. Therefore, during test selection, the

affected test requirement is identified and its corresponding value in the requirements matrix increased to a value, which lies between one and the maximum number of test cases traversing that particular test requirement. This will force the decision model to increase the number of test cases traversing the affected piece of code to the new value indicated by the requirements matrix. Similarly, during test update, it may be desirable to retain a number of additional test cases to exercise particularly critical sections of program code.

Other situations can arise in which the test requirements may have not been implicated by the proposed modifications, but still need to be considered as part of a test selection model. Consequently, any additional test requirements must be appended to the selected set of constraints. This action will force the decision model to adjust its solution and as a result, extra retestable tests may be determined. Conversely, test requirements may be deleted from those selected during change analysis thereby possibly reducing the number of retestable tests.

During maintenance, a program is often enhanced through the incorporation of additional features, with new source code being added and existing code either being modified or deleted. As part of the maintenance operation, improved testing procedures may be introduced which ensure that the new program release is validated using more stringent structural testing techniques. Therefore, a situation may arise in which portions of the code have been validated using different test coverage criteria. The decision model developed in this thesis can accommodate different sets of constraints whereby each set of constraints represents a testing history generated through the use of a different structural testing technique.

6.3.3 Generalised Goal Programming

Previously, the problems of test suite management were described in terms of a decision model which consisted of a single objective function and a set of constraints.

Thus, test selection and test update could be solved by selecting or maintaining a test suite either at minimum cost, time or priority. A benefit of this type of decision model was that any decisions concerning the choice of objective and constraints could be clearly made.

In practice, however, the problems of test selection and test update may require *several* objectives to be satisfied at the same time. For example, a critical piece of program functionality may require correction and maintenance programmers wish to revalidate the modified software using a set of retestable tests which exercise its most important features in the minimum amount of time, with less emphasis being placed on the cost of regression testing. One way of addressing this problem using the existing decision model is to: a) list *all* problem objectives, b) choose *one* of these objectives as the single objective in the model while considering all other objectives to be rigid constraints, and c) solve the resultant model. The steps b) and c) are then repeated with the remaining objectives as the single objective in the model. Finally, a solution is chosen from those solutions found in step c), which appears to 'best' satisfy all the listed objectives.

However, there are at least three drawbacks with this approach. The first of these relates to the uncertainty as to which single objective is really the most important one amongst several objectives. The second problem concerns the treatment of objectives as rigid constraints. An equality such as $f(\mathbf{x}) \geq b$, $f(\mathbf{x}) \leq b$, or $f(\mathbf{x}) = b$ must be formed, whereby the right-hand side of the inequality must reflect an aspired level which the left-hand side *must* achieve. However, setting an incorrect aspired level may result in the decision model having an infeasible solution. The third problem relates to the quality of the resultant solution, whereby the above method may not represent the solution that 'best' satisfies all the objectives.

There are at least two primary approaches, or philosophies, which form the basis for nearly all the multiple-objective techniques that have been proposed. These include weighting or utility methods, and ranking or prioritising methods.

The weighting method refers to those approaches which attempt to express the problem objectives in terms of a single measure, for example, as unit cost. The basic aim of all such models is to transform a multiple-objective model into a single-objective model. This is an attractive proposition in that it may reduce the computational effort. However, the major disadvantage with this approach is that it is associated with actually developing truly credible weights. For example, if one objective is to minimise the cost of regression testing and another is to minimise the time required for regression testing, how much more, or less important, is the expense of testing as opposed to the time taken?

The ranking, or prioritising, methods attempt to circumvent this problem by requiring users to rank the list of single objectives according to their perceived importance in the problem. For example, in determining a minimum number of test cases, a procedure may be used to first rank the test suite according to its cost, time and priorities. The problem with this approach is how to associate the results of a given solution to the satisfaction of the ranking.

The two basic approaches, described above, also represent some of the extremes in multiple-objective approaches. However, the benefit of a generalised **goal programming** model [124] for test selection and test update is that a comprimise can be developed, which results in a working combination of the above approaches. A *goal* is a mathematical function of the decision variables **x**, which represents the combination of a single-objective function f(**x**) with a target value. The standard form of a goal is defined in a way similar to that of a constraint, that is either as $f(\mathbf{x}) \leq b$, $f(\mathbf{x}) \geq b$, or $f(\mathbf{x}) = b$, where b is a constant. An *aspiration level* is a specific value associated with a desired, or acceptable, level of achievement of an objective. Thus, an aspiration level is used to *measure* the achievement of an objective. The *goal deviation* is the difference between the value of the resulting solution and the aspiration level. In all, but the most trivial problems, or where the aspiration level has been incorrectly set, goal deviations will be encountered. It should be noted that a deviation can represent an *over*achievement, as well as an *under*achievement, of a given goal.

It should be emphasized that an objective differs from a goal in that, for a goal, a minimum acceptable or target value for its level of performance is described, whereas in the case of an objective, it is simply stated that its measure of performance is to be minimised. As a result, the objective is not represented as an inequality. The difference between a goal and a constraint is more subtle. While both form inequalities, the concept of a goal implies more flexibility and less rigidity than that of a constraint. For the goal, the right-hand side of the inequality is simply a target value to which it aspires. However, in the case of the constraint, the right-hand side must always be achieved; otherwise the constraint is considered violated with the further implication that the entire problem is infeasible.

Consider the objective function expressed in general terms as $f_i(\mathbf{x})$; thus, $f_i(\mathbf{x})$ is the mathematical representation of objective *i* as a function of the decision variables $\mathbf{x} = (x_1, x_2, ..., x_n)$, and b_i constitutes the value of the aspiration level associated with objective *i*. Three possible forms of goals may then result: a) $f_i(\mathbf{x}) \leq b_i$ - the value of the objective $f_i(\mathbf{x})$ is to be equal to or less than b_i; b) $f_i(\mathbf{x}) \geq b_i$ - the value of the objective $f_i(\mathbf{x})$ is to be equal to or greater than b_i; and c) $f_i(\mathbf{x}) = b_i$ - the value of the objective $f_i(\mathbf{x})$ is to be equal to b_i.

Regardless of their form, these relations are transformed into the goal programming format by adding a negative deviation variable ($\eta_i \geq 0$) and subtracting a positive deviation variable ($\rho_i \geq 0$). By considering the relationship between the original goal form ($\geq$, $\leq$, $=$) and the deviation variables, the following statements can be made: a) to satisfy $f_i(\mathbf{x}) \leq b_i$, the positive deviation variable (ρ_i) must be minimised; b) to satisfy $f_i(\mathbf{x}) \geq b_i$, the negative deviation variable (η_i) must be minimised; and c) to satisfy $f_i(\mathbf{x}) = b_i$, both the positive and negative deviation variable (ρ_i and η_i) must be minimised. Table 6.1 summarises these points.

Goal Type	Goal Programming Form	Deviation Variables to be Minimised
$f_i(\mathbf{x}) \leq b_i$	$f_i(\mathbf{x}) + \eta_i - \rho_i = b_i$	ρ_i
$f_i(\mathbf{x}) \geq b_i$	$f_i(\mathbf{x}) + \eta_i - \rho_i = b_i$	η_i
$f_i(\mathbf{x}) = b_i$	$f_i(\mathbf{x}) + \eta_i - \rho_i = b_i$	$\eta_i + \rho_i$

Table 6.1 : Generalised Goal Programming Formulations

As seen from Table 6.1, the three goal types form constraints, which in terms of goal programming, can be treated in the same as rigid constraints. They are transformed to include the negative and positive deviation variables, and an attempt is made to minimise appropriate deviation variable, or a combination of them, in order to achieve the relationship shown.

Once every objective and constraint has been transformed, a relationship is developed which indicates and measures the level of achievement of any solution proposed. This relationship is appropriately named the *achievement function.* As the satisfaction of the goals or constraints is obtained by minimising various deviation variables, this should be indicated by the achievement function.

However, a further question remains to be answered. It concerns the quality of the solution **x** which is obtained by the multiple-objective model as represented by the above goal formulations. Is, for example, the quality measured in terms of how well it minimises the sum of the weighted goal deviations or satisfies the maximum (worst) goal deviation? Does the solution lexicographically minimise an ordered, ranked or prioritised, set of goal deviations?

The achievement function, developed to address the problems of test suite management, is one that combines the points described above. Thus, achievement is measured in terms of a lexicographic minimisation of an ordered set of goal deviations.

Weights may then be assigned within each set of goals at a particular rank. Thus, this achievement function or vector can be stated as follows:

$$\mathbf{a} = (a_1, a_2, ..., a_k, ..., a_K) \qquad (6.10)$$

where **a** represents the achievement vector for which the lexicographic minimum is sought, k is the ranking or priority, and

$$a_k = g_k(\eta,\rho) \qquad k= 1, 2, ..., K \qquad (6.11)$$

where $g_k(\eta,\rho)$ is a linear function of the goal or constraint deviation variables, which are to be minimised at rank or priority k. Moreover, the first term in **a**, that is, a_1 is reserved for the deviation function associated with any rigid constraints. The lexicographic minimum can be described as follows: given an ordered array **a** of non-negative elements a_k's, the solution given by $\mathbf{a}^{(1)}$ is preferred to $\mathbf{a}^{(2)}$ if $a_k^{(1)} < a_k^{(2)}$ and all higher order elements, that is $(a_1, ..., a_{k-1})$ are equal. If no other solution is preferred to **a**, then **a** is the lexicographic minimum. Thus, if two solutions $\mathbf{a}^{(r)}$ and $\mathbf{a}^{(s)}$ are available, where $\mathbf{a}^{(r)} = (0, 10, 16, 33)$ and $\mathbf{a}^{(s)} = (0, 7, 5, 100)$ then $\mathbf{a}^{(s)}$ is preferred to $\mathbf{a}^{(r)}$.

Another term, which is used to describe the lexicographic minimum notion is the concept of *preemptive priorities*. A solution that provides a lexicographic minimum to **a** also satisfies the concept of preemptive priorities. Any goal at preemptive priority k will always be preferred to, that is, preempt any at a lower priority k+1, ..., K regardless of any scalar multiplier associated with these priorities. It should be emphasised that this concept was first used implicitly in the single-objective decision model where the first priority was to find a solution, which satisfied the constraints, and the second priority was to minimise a single objective *without* violating the constraints. Thus, the concept of preemptive priorities can effectively be used in decision problems as an iterative screening process.

Some of the criticisms being directed at goal programming concentrate on the use of the lexicographic minimum or preemptive priorities. Although the lexicographic minimum may not always result in the most desirable measure of achievement for a given problem, it generally forms a starting solution which can be improved upon by relaxing the strict interpretation of the lexicographic minimum. Thus, the lexicographic minimum or preemptive priority measure is used primarily in this thesis for its flexibility and general applicability to the problems of test suite management.

Apart from specifying the objective functions and set of constraints, the construction of a multiple-objective decision model requires the following, additional assumptions to be made: a) aspiration levels may be associated with each and every objective so as to transform them into goals; b) negative and positive deviation variables are included for each and every goal and constraint; c) goals are ranked in terms of importance whereby any rigid constraints are set at priority one; and d) an achievement function is established in which all goals within a given priority are either commensurable or can, by means of weights, be made commensurable. Once these steps are accomplished, the following linear goal programming model is formed:

$$\text{lexicographically minimise} \quad \mathbf{a} = (g_1(\eta,\rho), \ldots, g_k(\eta,\rho)) \tag{6.12}$$

$$\text{subject to} \quad f_i(\mathbf{x}) + \eta_i - \rho_i = b_i \tag{6.13}$$

$$\text{and} \quad \mathbf{x} \in \{0,1\},\ 1 \le i \le m,\ \eta \ge 0,\ \rho \ge 0 \tag{6.14}$$

The benefits of this multiple-objective model are that it better reflects the problems of test suite management than any single-objective model. In future research, it is hoped that algorithms for solving this type of model can be investigated and integrated into the technique for selective revalidation described in this thesis.

6.4 Algorithms

The two primary determinants of computational difficulty for an integer programming problem with a single objective function are the number of decision variables and the structure of the problem. This situation is in contrast to linear programming problems, where the number of linear constraints is more important than the number of decision variables[3]. Depending on the nature of an integer programming problem, it may be possible to first use the approximate procedure of solving the problem using a linear programming algorithm and then rounding the non-integer variables to integers in the resulting solution [153, 227]. The same problem is thus considered except that the restriction concerning the need for an optimal *integer* solution is removed. If, for example, linear programming produces rather large numerical values, then rounding them up or down to the nearest integer value may provide a perfectly acceptable result and cause relatively small errors.

However, there are two problems with this approach. First, the rounding procedure may result in solutions that are infeasible; it is often difficult to decide on which way the rounding should be done in order to retain feasibility. Second, the rounding of the decision variables may cause the solution to be far from its optimal integer value. In this thesis, however, two categories of algorithms are examined which are suitable for solving integer programming problems. They include optimal algorithms, such as *cutting-plane methods* and *enumerative techniques*, and approximation algorithms.

6.4.1 Cutting-Plane Methods

Gomory [81] developed the first finite algorithm called the cutting-plane method. It is characterised by the idea of using a cut, which is a derived constraint with the property of cutting off part of the set of feasible solutions while not excluding any integer solution. In

[3] In integer programming, the number of constraints is of some importance, but it is strictly secondary to the other two factors.

this method, the cut constraints are constructed and linear programming algorithms applied iteratively until an optimal integer solution is found.

The Gomory all-integer cutting-plane method has been used with some success in solving large set-covering problems. However, it is less effective for general integer programming problems. Even though a cutting-plane algorithm converges in a finite number of iterations, this finite number may prove to be large; thus, this algorithm is an exponential time algorithm. In general, the cutting-plane method tends to be unpredictable in terms of computer run-time. This has been attributed to the fact that a cutting-plane method can make substantial progress in the first few iterations of solving a problem and then tends to slow down dramatically. Given an integer programming problem, however, a cutting-plane method should be used in the first instance, because if a converging solution to the problem exists, then this method will quickly obtain a solution. Should this approach prove to be unsuccessful, it may be necessary to use enumerative techniques in order to solve the problem.

6.4.2 Enumerative Techniques

The most obvious approach to solving an integer programming problem is to enumerate all possible candidate solutions and select an optimal solution from them. However, this approach, which is known as *explicit enumeration*, is not considered practical for problems of reasonable size as it requires all 2^n possible solutions of a problem with n decision variables to be investigated. An alternative approach, therefore, is to examine only *some* sets of integer solutions which will produce optimal solutions.

Implicit enumeration methods [85] apply heuristic rules so that during analysis only a portion of the 2^n solutions are considered. In particular, the correct selection of next decision variable to be investigated may significantly influence the algorithm efficiency. This is particularly true in the initial stages of solving the problem, where the poor selection of such a variable could result in a needless enumeration of a large number of

nodes. A heuristic rule designed to direct the search toward a solution has been given by Balas [12].

Consider the zero-one integer problem specified by Equations 6.7-6.9. To solve this problem, search algorithms are used which enumerate either explicitly or implicitly all 2^n possible zero-one vectors in **x**. In such procedures, the vast majority of solutions are enumerated implicitly. The enumerative procedure can be illustrated by means of a search tree composed of nodes and branches. Figure 6.1 illustrates such a search tree.

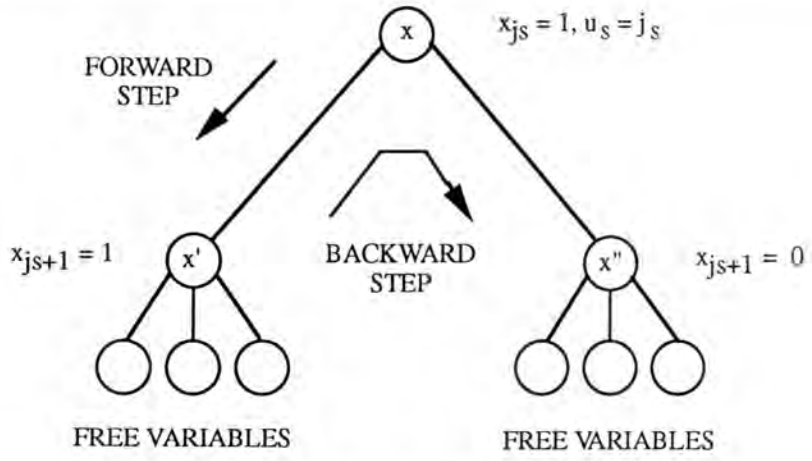


Figure 6.1 : Search Tree in Implicit Enumeration

A node corresponds to a zero-one candidate solution **x**. Two nodes connected by a branch differ in the state of one variable. Each variable can be in one of three states: *fixed at 1*, *fixed at 0*, or *free*. A new node is defined by fixing a variable to 1 (forward step) and a node is revisited by fixing a variable to 0 (backward step).

In order to explain how the above search tree is traversed, it is assumed that the current partial solution under consideration is given by vector $\mathbf{u} = (u_1, u_2, ..., u_s, 0, 0, 0)$ with entries interpreted as follows: a) $u_k = j_k$, $1 \le k \le s$, means that x_j has been fixed at 1 in accordance with the heuristic rule mentioned above. Its complement ($x_{jk} = 0$) has not

yet been considered; b) $u_k = -j_k$, $1 \leq k \leq s$, means that $x_{jk} = 1$ or 0, and that the complements of these values have already been considered; c) $u_k = 0$ for $k > s$.

Now suppose that an *incumbent solution* $\underline{\mathbf{x}}$ is available which represents the best feasible solution discovered so far. Furthermore, assume that $x_{js} = 1$, $u_s = j_s$ (the value of $x_{js} = 0$ has not yet been considered) and that at node $\mathbf{x}$, the partial solution $\mathbf{x} = (x_{j1}, x_{j2}, \ldots, x_{js}, 0, \ldots, 0)$ is infeasible. If a test is now used which by setting the variable x_{js+1} to 1 indicates that the infeasibility can be reduced and the objective function value $\mathbf{Z_0}$ can be improved, then a forward step is taken; thus, $u_{s+1} = j_{s+1}$ is added to the vector $\mathbf{u}$.

At the new node $\mathbf{x'}$, it is found that $\mathbf{x'}$ is feasible and it produces an objective function value $\mathbf{Z}$ which is better (less) than that associated with the incumbent solution. The incumbent solution $\underline{\mathbf{x}}$ is now replaced by $\mathbf{x'}$ and the objective function value $\mathbf{Z_0}$ is replaced by $\mathbf{Z}$. All *completions* in this partial solution with $x_{js+1} = 1$ have now been enumerated since there can be no improvement on the current feasible solution by fixing any new free variable at 1. Therefore, this partial solution has been *fathomed* and all completions of it have been implicitly enumerated.

Next, all completions of $\mathbf{x}$ with $x_{js+1} = 0$ are considered with a backtracking step being taken to node $\mathbf{x''}$. However, it is found that there is no attractive completion for this node, that is no completion of this partial solution can produce an optimal solution. At this point, all possible completions of $\mathbf{x}$ have been enumerated whereby $x_{js+1} = 1$ and $x_{js+1} = 0$ have been examined.

A backtracking step is now taken to consider $x_{js} = 0$. The element u_{js} of the vector $\mathbf{u}$ is now set to $-j_s$, assuming that previously $x_{js} = 0$ was not considered. If, however, $u_s = -j_s$ and $x_{js} = 0$, that is, $x_{js} = 1$ *was* previously considered, the rightmost element of $\mathbf{u}$ is found, the sign is changed to negative, and the new partial completions are examined. In this case, backtracking steps over more than one branch of the search tree are taken. The

block diagram in Figure 6.2 summarises the basic enumeration process for a general implicit enumeration scheme.

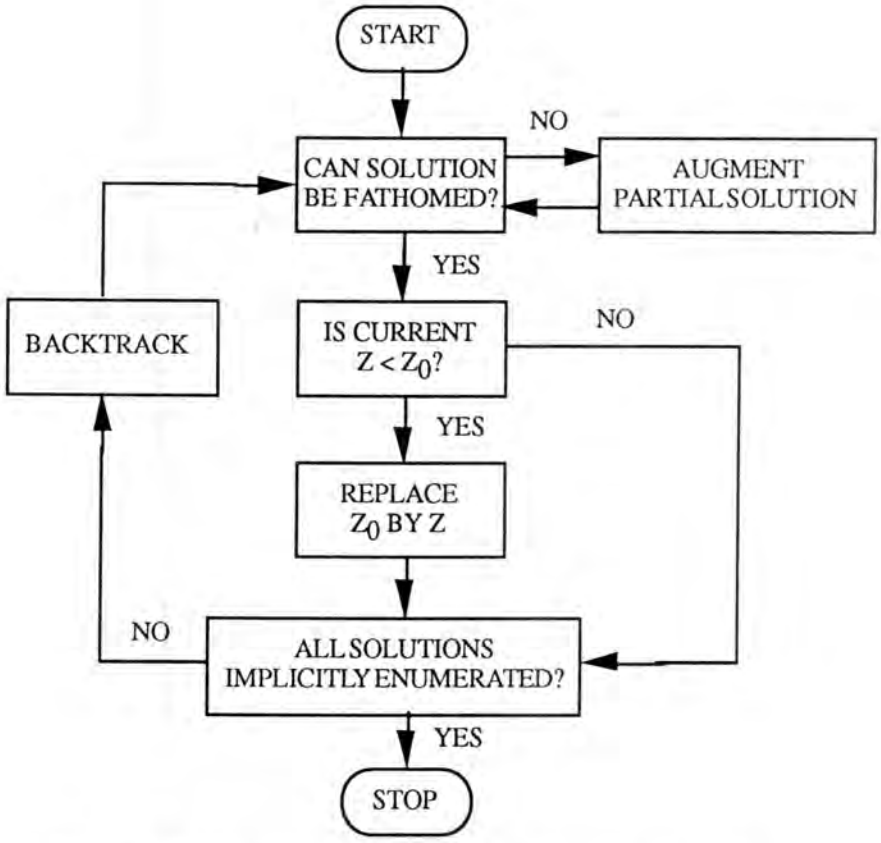


Figure 6.2 : Block Diagram of Implicit Enumeration Scheme

As mentioned earlier, the effectiveness of implicit enumeration algorithms is dependent on the strength of their exclusion tests. It has been found that the criterion used to detect whether or not a partial solution has a feasible completion is not very effective. This is especially true whenever the algorithm attempts to solve problems with a considerable number of constraints. In order to remedy this problem, the concept of a *surrogate constraint* has been introduced which effectively combines a set of constraints, but does not eliminate any of the original, feasible integer points of the problem [80, 79]. The new constraint is devised such that it has the potential to reveal information that cannot be conveyed by any of the original constraints considered separately. Consider the two constraints:

$$2x_1 - x_2 \leq -1 \quad (6.15)$$

and $$-x_1 + 2x_2 \leq -1, \quad (6.16)$$

where $$x_1, x_2 \in \{0,1\} \quad (6.17)$$

By considering each constraint separately, it cannot be concluded that the two constraints do not have a feasible solution. However, if the two constraints are combined, the resulting surrogate constraint, $x_1 + x_2 \leq -2$, shows decidedly that the problem cannot have a feasible solution. Empirical evidence indicates that the use of the surrogate constraint can be effective in reducing the computational time [28].

One particular observation about implicit enumeration is that the computation time is often data-dependent. Therefore, the specific ordering of the decision variables and constraints may have a direct effect on the efficiency of an implicit enumeration algorithm. For example, the constraints could be ordered such that the most restrictive constraint is the first one to be investigated by the algorithm, and variables could be arranged according to an ascending order of their objective coefficients. Both conditions are favourable to producing *faster* fathoming of the partial solutions.

A number of implicit enumeration algorithms have been designed to solve the binary programming problem [61, 78]. Comparative studies [64, 69, 173, 190, 208, 246] of their computational performance concluded that for the majority of samples, the Balas algorithm could solve problems in fewer iterations, be applied to denser matrices[4], and examine problems with a larger number of decision variables. However, as any enumerative technique must investigate either implicitly, or explicitly, all 2^n binary combinations, the solution time varies almost exponentially with the number of decision variables n.

[4] The density of a matrix is related to the ratio of non-zero to zero values in the constraint matrix A.

6.4.3 Heuristic Methods

An alternative approach to solving large integer programming problems, in particular set-covering problems, is to apply heuristic methods or approximation algorithms [14, 36, 42, 111, 132, 172]. Heuristic methods can be considered as incomplete enumerative techniques, which do not investigate branches of the search tree even though they may contain the optimal solution. As a result, these algorithms are not *guaranteed* to find an optimal solution, but they extremely *efficient* in solving large problems. More importantly, however, they are polynomial-time algorithms.

Effective heuristic methods should possess three important properties: a) they should yield solutions within a reasonable amount of computation time, b) the solutions that are produced should, on average, be as close as possible to the optimal solution, and c) the probability of solutions which vary widely from the optimal solution, should be low. The conditions b) and c) are not identical, although they are related, since obtaining a solution close to the optimal one does not preclude the possibility of a poor solution in certain instances. The effectiveness of a heuristic method, however, can only be judged after conducting extensive numerical tests.

Two approximation algorithms have been developed to solve the problems of test suite management and, in particular, the problem of test update. However, neither algorithm accomodates some of the important characteristics required to solve a decision model consisting of generalised objectives and constraints. For example, neither algorithm considers the fact that each test case t_j may be associated with a cost c_j. Instead, both algorithms implicitly assume that all test cases have the same cost. In addition, both algorithms concentrate on solving a set-covering problem for which the values in the requirements matrix $\mathbf{b}$ are restricted to one, as opposed to an arbitrary value between one and the maximum cardinality of each testing subset.

The work by Leung [160] describes a heuristic method of order O(mn), where m represents the number of testing subsets T_i for the program and n is the number of test cases t_j in the test suite. The method is described in terms of operations on the constraint matrix A[5]. The initial step of the algorithm involves the summation of all columns C_j in the constraint matrix A forming a column C_T whose elements indicate the cardinality of each testing subset T_i. In the next step, each test case t_j in the test suite is processed by subtracting its associated column, C_j, from the current value of C_T. The order for processing is either in ascending, or descending, order of test execution. After subtracting each column C_j from its current C_T value, the new value of C_T to examined to see whether any new zero values have appeared. According to the author, the appearance of a zero value in the summation C_T relates to the presence of a representative test case t_j which needs to be retained. The algorithm repeats this step for each test case t_j until all tests in the test suite have been processed.

Consider the following example, in which the testing information is presented in terms of the test requirements r_i and the testing subsets T_i, and a constraint matrix that illustrates test cases t_j exercising testing subsets T_i.

test requirement, r_i	testing subset, T_i
REQ1	{2,5}
REQ2	{5}
REQ3	{1,2,3}
REQ4	{3,6}
REQ5	{1,4}
REQ6	{1,6}
REQ7	{3,4,7}
REQ8	{2,3,4,7}

test case, t_j / testing subset, T_i	1	2	3	4	5	6	7
T_1	0	1	0	0	1	0	0
T_2	0	0	0	0	1	0	0
T_3	1	1	1	0	0	0	0
T_4	0	0	1	0	0	1	0
T_5	1	0	0	1	0	0	0
T_6	1	0	0	0	0	1	0
T_7	0	0	1	1	0	0	1
T_8	0	1	1	1	0	0	1

Figure 6.3 : Testing Information

[5] Recall from Section 6.2, that a binary programming problem consists of an objective function Z, a cost vector c, and a set of constraints whose coefficients are represented by a matrix A.

A set of count vectors, which correspond to test cases t_j in the test suite, can be derived from the columns of the constraint matrix; C_1 = [00101100], C_2 = [10100001], C_3 = [00110011], C_4 = [00001011], C_5 = [11000000], C_6 = [00010100] and C_7 = [00000011] with the resulting summation C_T = [21322234]. If the test cases are processed in an ascending order, then the first step requires $C_{T'} = C_T - C_1$ = [21221134]. The new summation $C_{T'}$ is then examined to see if any of the $C_{T'}$ elements contain zero entries. In this case, however, no new zero entries have appeared in C_T and, therefore, test case t_1 is deemed redundant; otherwise, the test case would have been retained. This operation is now repeated for test case t_2, whereby $C_{T''} = C_{T'} - C_2$ = [11121133], and so forth. A similar approach is taken with a descending order of subtraction, whereby $C_{T'} = C_T - C_7$ = [21322223], and so forth.

However, the algorithm developed by Leung [160] fails to determine a representative set of test cases and, therefore, does not eliminate a maximum set of redundant test cases. The application of Leung's algorithm to the above example yields a representative test set consisting of five test cases (t_3, t_4, t_5, t_6, and t_7); this is based on an ascending order of test execution. In reverse order, however, the same algorithm results in only four test cases (t_1, t_2, t_3, t_5) being chosen. In fact, the minimal set of test cases consists of test cases t_1, t_3, and t_5, with the test cases t_2, t_4, t_6 and t_7 being redundant.

The example illustrates the problems arising from the restrictive ascending, or descending, order of processing. Test cases are retained, or discarded, based on whether or not a testing subset is still exercised by a test case. No criteria are applied which could guide the algorithm in selecting a *specific* test case from amongst those test cases that comprise each testing subset. As a result, the algorithm tends to choose representative test cases from the latter half of the test suite when processing in an ascending order and from the former half of the test suite during a descending order.

The work by Harrold *et al.* [95] describes a heuristic method which determines a representative set of test cases in order O(nm(m+n)) where m represents the number of

testing subsets T_i for the program and n is the number of test cases t_j in the test suite. It defines a selection criterion which is based on the maximum cardinality of each test case. The algorithm consists of the following two steps. First, all subsets T_i are examined in order to determine which of them contain only a single test case. Each discovered test case immediately forms part of the representative set of test cases, or so-called *hitting set.* All testing subsets, which include this test case, are marked as processed. Second, all unmarked subsets with cardinality two are investigated. The test case which appears in the maximum number of testing subsets is added to the hitting set. If, at this stage, a tie arises between several of these test cases, unmarked subsets of next higher cardinality are examined using only these tests. The process of determining which test case is represented in the majority of testing subsets is continued until the tie is broken and a test case can be selected. Subsequently, all testing subsets, which include this test case, are marked as processed. This procedure may then be repeated using testing subsets of increasing cardinality.

Consider again, the testing information illustrated in Figure 6.3 where a test suite consisting of seven test cases and eight test requirements, that is testing subsets, is being examined. Test case t_5 is added to the hitting set since it is the only test case exercising testing subset T_2; therefore, testing subsets T_1 and T_2 are marked as being processed. Unmarked testing subsets of cardinality two are then examined. Each of test cases t_3 and t_4 appears in one of the testing subsets under consideration, while each of test cases t_1 and t_6 appears in two of those testing subsets. Since there is a tie between test cases t_1 and t_6 for the maximum, processing continues with unmarked testing subsets of cardinality three. Thus, testing subsets T_3 and T_7 are considered next. Only the tests cases involved in the tie are used to compute the maximum for cardinality three. Test case t_1 appears in testing subset T_3 while test case t_6 appears in neither of the testing subsets. Therefore, test case t_1 is chosen and added to the hitting set. Testing subsets T_3, T_5, and T_6 are marked since they contain test case t_1. Processing now continues with testing subset T_4, the only unmarked testing subset of cardinality two. Again, there is a tie between test cases t_3 and t_6 causing testing subsets of cardinality three to be investigated.

Test case t_3 appears in testing subset T_7 and thus it is added to the hitting set which allows the remaining testing subsets T_4, T_7, and T_8 to be marked. The resulting hitting set comprises test cases t_1, t_3, and t_5, with the test cases t_2, t_4, t_6 and t_7 being redundant. While the algorithm described by Harrold-Gupta [95] proves to be more precise than the one developed by Leung, it is complex in terms of computational effort.

6.4.4 New Approach

In this thesis, a new heuristic method is developed to address the limitations of the above algorithms. The new algorithm is characterised by its simplicity and flexibility in computing a minimum cover. It considers the situation whereby each test case t_j is associated with a cost c_j and each test requirement, that is testing subset, is associated with an arbitrary value in the requirements matrix b which lies between one and the maximum cardinality of each testing subset. The algorithm defines a selection criterion which is based on the ratio of maximum cardinality to minimum cost of each test case.

Step 1 of the algorithm `Minimum_Cover` consists of the required initialisation and preprocessing of the test cases. The cardinality of each test case t_j is established by inspecting the testing subsets T_i to see if the test case exercises them. The number of testing subsets, which contain the test case t_j, are then indicated by the corresponding value of element *j* in the array `card`. If a test requirement should specify a constraint on the number of test cases exercising it, then the array `init_requirements` reflects this number. After the test case cardinalities have been determined, the array `ratio` is used to reflect the value of test cardinality/test cost for each test case t_j. In Step 2, the sum of the elements in array `ratio` is determined to ascertain whether any of the test requirements remain to be exercised. If so, a test case is chosen, based on the test selection criterion, and included in the cover; otherwise the sum is zero and the algorithm terminates. If a test case is selected, a subtraction process ensures that all testing subsets containing this, and other test cases, no longer need to consider these test requirements if their constraints have been fulfilled. As a result, the array `card` and, subsequently, the

array `ratio` can be adjusted accordingly. This step is now repeated using the test case with the highest value in the array `ratio`. The algorithm which implements this heuristic method is given in Figure 6.4.

```
algorithm Minimum_Cover

input       cost              : array [1..n] of test cost factors c_j,
            T_i               : set of testing subsets T_i containing t_j test cases,
            init_requirements : array [1..m] of constraints b_i.

output      cover             : set of test cases t_j representing the minimum
                                cover, initialised empty.

declare     card              : array [1..n] of current cardinality of test cases t_j,
            ratio             : array [1..n] of ratio card/cost,
            new_requirements  : array [1..m] of intermediate constraints b_i,
            next_test         : test case t_j.

begin
    /* Step 1: Initialisation */

    foreach j, j = 1 to n do
        foreach i, i = 1 to m do
            if t_j ∈ T_i then
                card[j] : = card[j] + 1/init_requirements[i]
            endif
            new_requirements[i] = init_requirements[i]
        endfor
        ratio[j] := card[j]/cost[j]
    endfor

    /* Step 2: Selection Process */

    while Sum(ratio) ≠ 0 do                  /* termination criterion */
        next_test := Max_Ratio(ratio)        /* test selection criterion */
        cover := cover ∪ next_test
        ratio := Substract(next_test, card, ratio)
    endwhile

    return(cover)

end Min_Cover

function Sum(ratio)

/* this function determines whether the sum of test case ratios is zero or not */

declare     sum               : sum of test case cardinalities, initialised to zero

begin
    foreach j, j = 1 to n do
        sum := sum + ratio[j]
    endfor
```

```
        return(sum)

end Sum.

function Max_Ratio(ratio)

/* this function determines the test case with the maximum cardinality/cost ratio */

declare     max             : maximum in a set of numbers, initialised to zero,
            index           : test case with maximum ratio, initialised to zero
begin
        foreach j, j = 1 to n do
            if ratio[j] > max then
                max := ratio[j]
                index := j
            endif
        endfor

        return(index)

end Max_Ratio.


function Substract(next_test, card, ratio)

/* this function adjusts the test case cardinalities after a particular test case is selected */

begin
        card[next_test] := 0
        foreach i, i = 1 to m do
            if next_test ∈ Ti then
                Ti := Ti - next_test
                new_requirements[i] := new_requirements[i] - 1
                if new_requirements[i] = 0 then
                    foreach j, j = 1 to n do
                        if tj ∈ Ti and not next_test then
                            card[j] := card[j] - 1/init_requirements[i]
                            ratio[j] : = card[j]/cost[j]
                            Ti := Ti - tj
                        endif
                    endfor
                endif
            endif
        endfor

        return(ratio)

end Subtract.
```

Figure 6.4 : New Heuristic Method - The Algorithm

Consider again, the testing information illustrated in Figure 6.3. In that example, the test costs and associated constraints are implicitly assumed to be one. The first step of the

algorithm examines each of the testing subsets T_i in order to establish the cardinality of each test case t_j; thus, `card` = [3343222]. With a cost factor of one for each test case, the array `ratio` reflects the array `card`, that is `ratio` = [3343222][6]. As the sum of the elements in array `ratio` is not zero, the algorithm selects the test case with the highest value element in the array; thus test case t_3 is identified and included in the cover[7]. Examination of the test requirements reveals that test case t_3 exercises the requirements {REQ3, REQ4, REQ7, REQ8}. Subject to the constraints specified by each test requirement, the test cases contained in the corresponding testing subsets are no longer required to exercise these test requirements. Consequently, the cardinality of each test case t_j associated with the test requirements {REQ3, REQ4, REQ7, REQ8} can be reduced and a new set of values calculated for array `card`, that is array `ratio` = [2101210]. During the second iteration, the algorithm ensures that the sum of the new elements in array `ratio` is non-zero and test case t_1 is subsequently selected. This implicates test requirements {REQ5, REQ6}, both of which are exercised by test case t_1. As a result, the cardinality of each test case t_j exercising these requirements is affected such that the array `ratio` = [0100200]. In the third, and final, iteration of the algorithm, test case t_5 is selected with test requirements {REQ1, REQ2} being considered. The subtraction process, which follows, causes all elements of array `ratio` to be reduced to zero values and the algorithm terminates as all test requirements have been exercised by at least one test case. The final cover consists of test cases t_1, t_3, and t_5, with the test cases t_2, t_4, t_6 and t_7 being redundant.

The worst case run-time complexity of the new algorithm has also been analysed to demonstrate its efficiency and suitability for incorporation in a test suite management procedure. Let **n** denote the number of test cases t_j in the test suite and **m** the number of test requirements, that is testing subsets. The heuristic presented involves two main steps: the computation of the number of occurrences of each test case in various testing subsets

---

[6] However, the values in array `ratio` will vary depending on the test costs and the constraints placed upon the test cases by the test requirements.

[7] Any ties for maximum value would be resolved by choosing the first test case encountered with the maximum value.

and the selection of the next test case. The latter step is performed repeatedly in order to find a minimal cover for the testing subsets. The computation of the number of occurrences of each test case in various testing subsets requires order O(**nm**) time since there are **m** testing subsets and they are examined **n** times. The selection of the next test case to be included in the minimal cover requires the complexity of each function `Sum`, `Max_Ratio` and `Subtract` to be examined; the respective worst case run-time complexities are order O(**n**), O(**n**), and O(**nm**). Therefore, each iteration during the selection of a test case takes at most order O(**n**(**m** + **1**)) time. Assuming that both **n** and **m** are large, the run-time complexity is order O(**nm**). The selection of a test case and the recomputation of the test case cardinalities is repeated at most **m** times since after the selection of a test case at least one testing requirement is satisfied[8]. Therefore, the overall run-time complexity is O(**nm** + **m**(**nm**)). Rearranging this expression, the time complexity is order O(**nm**(**1** + **m**)), and assuming that a reasonable number of iterations are performed by the algorithm, then the run-time complexity of the algorithm is order O($\mathbf{nm^2}$). This compares favourably with the run-time complexity of the algorithm proposed by Harrold *et al.* [95]. Although the above analysis provides the worst case time complexity of the algorithm, in practice, the algorithm may solve the given problem faster.

Two important questions arise from this analysis; the first concerns the size of the *problems* which may be solved by the new heuristic method, while the other concerns the size of the *solutions* generated by it. Lee-He [151] provides an indication of the problem size which can be addressed; they describe the solving of set-covering problems consisting of several *hundred* test requirements and test cases. It was also found that for the decision problems being examined, the size of the solution obtained by the heuristic method was comparable to that obtained using an optimal algorithm. However, in order to obtain a better measure of its effectiveness, a larger number of decision problems need to be considered. It is hoped that this work can be pursued in future research.

---

[8] This is based on the assumption that each test requirement needs to be exercised by no more than one test case.

6.5 Application

The following section describes selective revalidation based on the technique developed in this thesis. Details are given concerning procedures for test selection and test update, both of which rely upon the use of the new heuristic method developed in Section 6.4.5. The objective of the technique, described in this thesis, is to provide maintenance programmers with a set of formal guidelines which will enable them to perform the revalidation of individual program modules in a systematic and efficient manner. It also aims to integrate the concept of selective revalidation into the traditional maintenance cycle.

During the maintenance phase, programmers typically receive a number of change requests from users. Each of these change requests is then described, in detail, in a change proposal which contains a list of modifications to be made to the program specification, design and code. With respect to the implementation and, in particular, individual program modules, a list of basic modifications comprises the addition, deletion or alteration of program statements. The technique for selective revalidation examines each of these modifications assuming that a test suite and a set of test costs are provided by users. The technique, which is presented in Figure 6.5, consists of two steps: one for test selection, the other for test update.

---

procedure Selective_Revalidation

input	test_suite	: current testing history containing testing subsets T_i,
	costs	: set of test costs,
	modification	: list of basic modifications.
output	new_test_suite	: set of test cases reflecting updated testing history,
declare	Change	: function that returns the set of affected testing subsets T_i,
	Rerun	: function to reset test execution history, rerun retestable tests, and indicate test coverage,
	Update	: function to update the current testing history and indicate test coverage,
	requirements	: current set of testing subsets T_i,
	retestable_tests	: set of retestable test cases,

```
        redundant_tests   : set of redundant test cases,
        new_tests         : sets of new test cases,

        test_coverage     : a boolean value to indicate satisfactory test coverage,
                            initialised to true,
        phase             : value to indicate test selection/test update.

begin
        foreach modification do
            requirements := Change(modification, test_suite)
            phase := test_selection
            retestable_tests := Decision_Model(costs, requirements, phase)
            implement modification
            test_coverage := Rerun(retestable_tests)
            while not test_coverage do
                    generate new_tests
                    test_coverage := Update(new_tests)
            endwhile
            phase := test_update
            redundant_tests := Decision_Model(costs, test_suite, phase)
            new_test_suite := test_suite - redundant_tests
        endfor

        return(new_test_suite)

end Selective_Revalidation.

function Decision_Model(costs, requirements, phase)

declare Modify_Model      : procedure to modify constraints,

        model             : decision model consisting of test costs and testing
                            subsets T_i,
        solution          : set of non-redundant test cases,
        constraints       : boolean value to indicate that specific constraints
                            associated with testing subsets T_i need to be amended,
                            initialised to false,

begin
        model := costs + requirements
        if constraints then
            Modify_Model(model)
        endif
        if phase = test selection then
            retestable tests := Minimum_Cover(model)
            return(retestable tests)
        elseif phase = test update then
            solution := Minimum_Cover(model)
            redundant_tests := requirements - solution
            return (redundant_tests)
        endif
end Decision_Model.
```

Figure 6.5 : A Technique for Selective Revalidation

In the case of test selection, change analysis is performed in order to determine the impact of each basic modification on the existing program code. The function `Change` relates to the change analysis technique, which was described in Section 5.5. Its purpose is to identify a set of affected testing subsets T_i, that is test requirements, with which to retest the modified code. The resulting test requirements are then used, alongside the test costs, to establish a decision model for solving the problem of test selection. After possible adjustments to the constraints and test costs (specified by users), the decision model is solved using the heuristic algorithm `Minimum_Cover` described in Section 6.4.5. It returns the set of retestable tests which are needed to ensure the consistency of the modified program. The modification is then implemented. At this stage, it may be desirable to introduce an additional step into the procedure where the impact of the modification is assessed in terms of the cost of rerunning the test cases. As a result, it may prove too costly to implement the change and alternative modifications must be investigated.

In the case of test update, the set of retestable tests is rerun and the test coverage monitored to ensure that the quality of the test suite is maintained. If this is not the case, then additional test cases may need to be generated until the given test coverage criterion has been satisfied. The test suite, together with the test costs, can now be used to establish a decision model for solving the problem of test update. Once again, adjustments can be made to the model, which may then be solved using the new heuristic method. As the solution obtained from the function `Decision_Model` forms the set of redundant test cases, the updated test suite is obtained by eliminating the redundant tests from the current test suite.

6.6 Summary

In this chapter, a systematic and efficient solution to the problems of test suite management has been developed. Through the use of operations research, the problems of test selection and test update have been formulated as binary programming or set-

covering problems. Different testing objectives and constraints have been considered as part of each problem and incorporated into the corresponding decision model in order to better reflect the characteristics of the problem. Moreover, a way has been found to specify and solve the problems of test suite management independently of any structural testing technique.

Two types of algorithms have been identified for solving the resulting decision models. While optimal algorithms tend to produce a minimal solution for a given problem, they may also exhibit an exponential run-time. In constrast, heuristic methods cannot guarantee the production of a minimal solution for all problems, but they have been found to be extremely efficient at computing solutions to problems. Subsequently, a new heuristic method has been developed which compares favourably with both existing optimal and other approximation algorithms in terms of the problem sizes being addressed, the size of solutions being generated and the worst-case run-time complexities.

Selective revalidation has also been described based on the technique developed in this thesis. Details are given, concerning procedures for test selection and test update, both of which rely upon the use of the new heuristic method. It is also shown how the technique can be integrated into the traditional maintenance cycle.

Chapter 7
RETEST - Development of a Tool Suite

7.1 Introduction

Software maintenance consists of four distinct phases whose activities are supported by a wide range of automated tools [196]. The first phase is concerned with analysing the program code to be modified in order to understand its structure; source code browsers [217], cross-referencers [209, 266], and complexity analysers [191, 178] assist with program comprehension. During the second phase, a set of modifications is specified by way of change proposals; change management systems [51, 134] provide the editing and storage facilities required for specifying such proposals. In the third phase, the consequences of the proposed changes are determined; program slicers [130, 189] and ripple effect analysers [165, 167] have been developed for this purpose. Finally, the fourth phase involves the implementation and testing of the modified program to ensure its functional consistency; numerous regression testing tools [22, 162] are available. However, none of these tools provide facilities for selective revalidation, which ensures the structural consistency of the modified program [100].

In this chapter, the development of a tool suite known as **RETEST** (**Re**gression **Te**sting Support **Tools**) [102] is described which can provide the support needed for the selective revalidation of C program modules during software maintenance. The aim of developing this prototype is not to develop a production-quality tool suite, but to demonstrate the feasibility as well as the usefulness of the technique for selective revalidation described in this thesis. It is envisaged that the tool suite will be used in conjunction with existing regression testing tools in order to complement and enhance current practices. In Section 7.2, the design of the tool suite is described, with particular emphasis being placed on discussing its structure and interface to current regression testing tools. Section 7.3 outlines the implementation of the tool suite by giving a brief

description of each of its components and their purpose with respect to selective revalidation.

7.2 Design Issues

7.2.1 Structure

The design of a tool suite emphasizes the concept of *modularity* in that each of its components is developed as a standalone tool and then assembled around a common user interface and program representation, notably in the form of the Program Dependency Graph. This approach improves the testability of the tool suite as individual components can be tested in isolation and then integrated into the tool suite. At the same time, a modular design promotes the maintainability of a tool suite as existing tools can easily be updated and new tools introduced. For debugging purposes, a diagnostic option is available from within the user interface which, when enabled, displays messages relating to the tools' current processing status. This considerably reduces the time and effort required to locate possible errors in the tool suite. To achieve portability, the tool suite is implemented under a common programming environment, such as the Unix™ operating system, and using the C programming language together with compiler development tools such as Lex and YACC. Moreover, the adaptability of the tool suite is considered so that it can be used on a variety of hardware platforms; important system parameters are therefore defined as program constants which can be adjusted if the size of programs being analysed exceeded the default values. To ensure the integrity of the tool suite, unauthorised accesses to data files or illegal operations within the user interface are kept to a minimum by restricting file permissions and including input error-handling routines.

In order to support the different tasks associated with a technique for selective revalidation, a total of five automated tools have been developed. Figure 7.1 provides a

™ Unix is a trademark of Bell Laboratories.

schematic of the tool suite, which includes tools for *program instrumentation*, *program flow analysis*, *test coverage analysis*, *change analysis*, and *test suite management*. All tools are accessed via a common user interface component which is not shown. The diagram illustrates the main flows of information through the proposed tool suite by means of bold arrows. Important inputs, and outputs, are depicted by the shaded, and striped boxes, respectively.

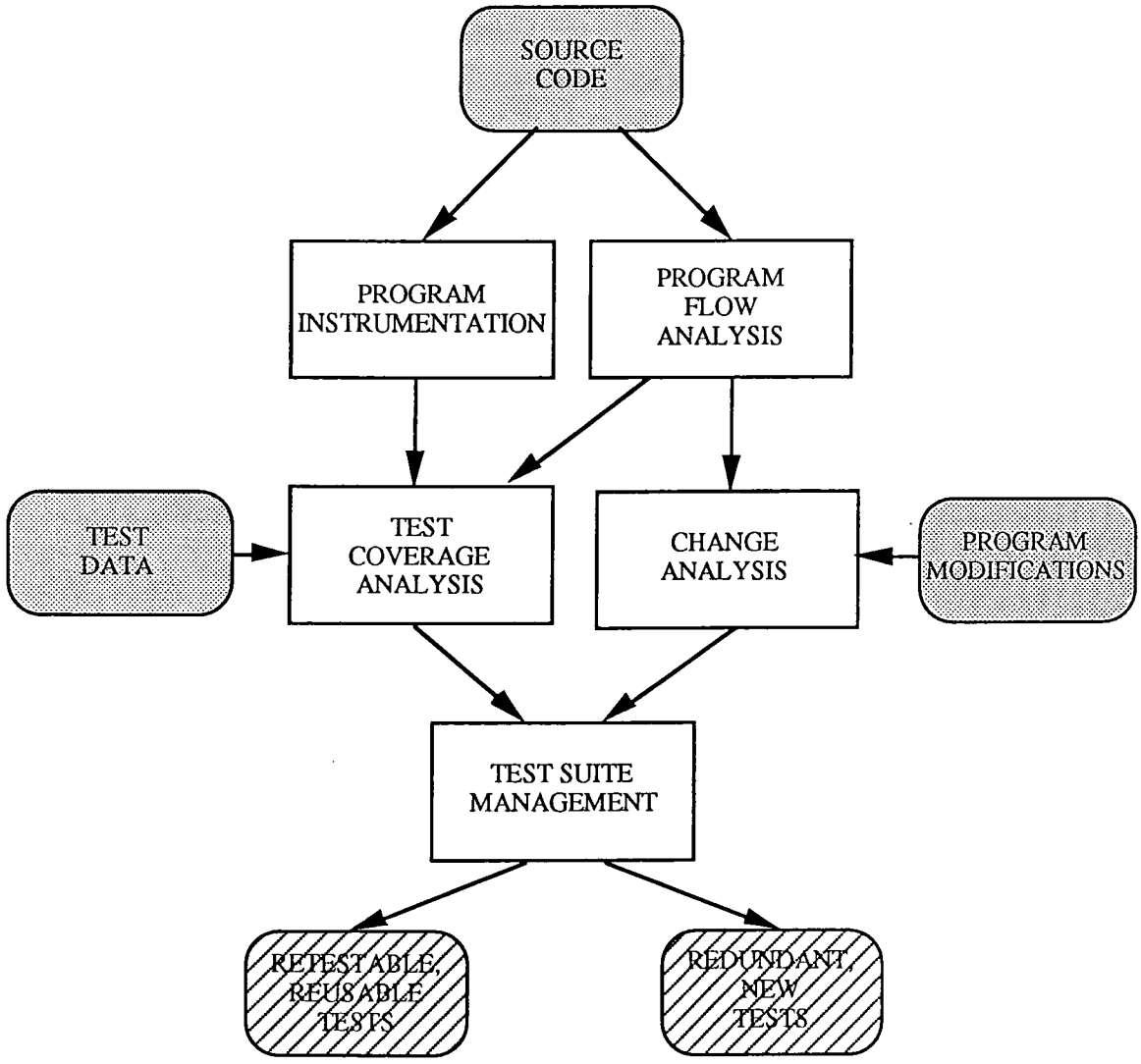


Figure 7.1 : Architecture of the Prototype Tool Suite

7.2.2 Interfaces

An important consideration in the design of the tool suite is its *interoperability*; this concerns the definition of its external interfaces. In terms of the user interface, the design envisages a menu-driven system with command-line prompts instead of a graphical user interface. Apart from considerably reducing the development effort, this approach has the advantage of allowing the tool suite to be used with character-based terminals as well as bit-mapped workstation displays.

However, particular attention must be paid to the design of the interface which allows the tool suite to interact with existing regression testing tools. The importance of this interface relates to its potential for improving the existing process of regression testing. By combining the proposed tool suite with a commercial regression testing tool, the regression testing procedure could be entirely automated. This concept is illustrated in Figure 7.2 where an existing regression testing tool and its interactions with the software under test are depicted by the shaded boxes and solid arrows, while the proposed tool suite and its interactions with both the software under test and the proposed program modifications are indicated by the striped boxes and dotted arrows.

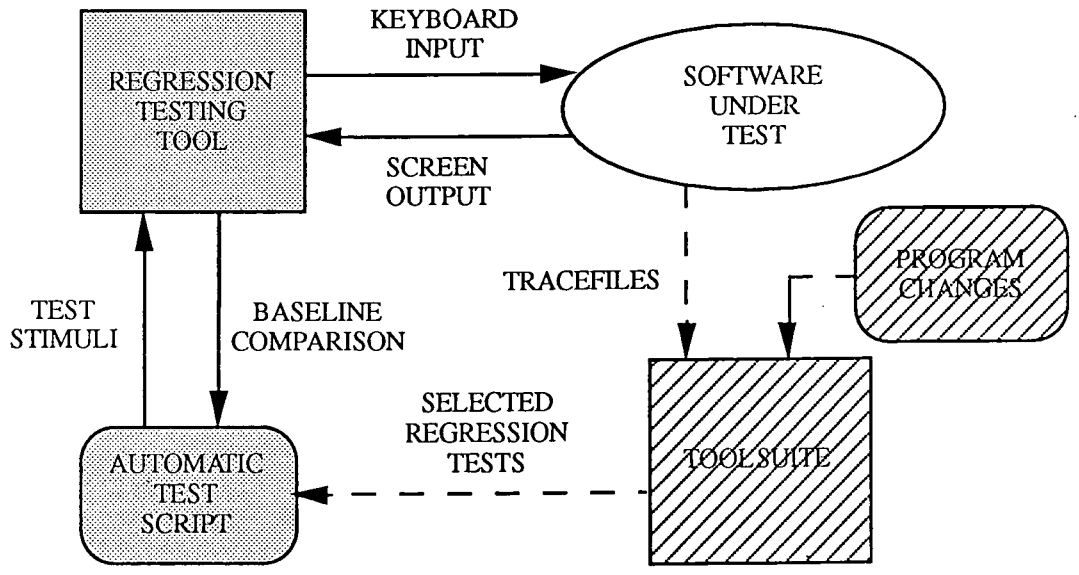


Figure 7.2 : Interfacing the Tool Suite to an Existing Regression Testing Tool

At present, regression testing tools assist users with a number of time-consuming tasks. During the initial validation of the software, a regression testing tool is used to record users' input stimuli (keystrokes) and capture the corresponding output (screenshots) from the executing application. Test cases are then formulated using these interactions, and stored as a so-called *baseline set*. In response to program modifications, the regression testing tool automatically replays those baseline test cases, which have been specified by users, and compares their current output with their previous output. Any differences in the two outputs are flagged to indicate the possible presence of errors. After any errors have been corrected, the baseline set of test cases is updated to reflect the modified functionality of the software.

To automate the current process of regression testing, users need to create an Automatic Test Script or ATS. This script is specified using a customised scripting language, which contains constructs similar to those found in most programming languages. It allows users to describe individual test cases and hierarchically organise them into *test groups*. Two advantages accrue from this type of test organisation. First, users can execute part, or all, of the test set according to their needs. Second, since the functions of most programs are organised hierarchically, the test structure can accurately model most program structures.

Figure 7.3 illustrates the current structure of an Automatic Test Script. It forms a hierarchic structure, whereby the test group `example` contains a series of test cases including `first.test` and `second.test`. Each test case specifies a list of elements including: a) the test case name (`DEFINE`); b) a brief textual comment explaining the purpose of the test (`SOURCE`); c) a series of system commands that invoke the software under test and specify the test case inputs and their outputs (`ACTIVATION`); and d) an evaluation criterion, which compares the current output of the test case with its previous output, which is stored as part of the baseline set (`EVALUATION`).

```
/* example.ats - Description of an Automatic Test Script */
DEFINE GROUP example
/* Test group defines a series of test cases */
    {
      DEFINE CASE first.test
        /* Illustrates the structure of a test case */
        {
          SOURCE
          "Comment to explain origin of this test case";
          ACTIVATION
          "System commands to execute this test case";
          EVALUATION WITH BASELINE
          "first.out" vs. "first.baseline";
        }
      DEFINE CASE second.test
        ...
    }
```

Figure 7.3 : Features of an Automatic Test Script

In the event of program changes, maintenance programmers have two possible choices for retesting their modifications. They can invoke the regression testing tool and command it to either automatically rerun a set of regression tests, which they themselves have chosen (interactive mode), or rerun the entire test suite (batch mode). Such strategies, however, can be *ad hoc* and wasteful of resources.

A better approach is to interface the proposed tool suite to the regression testing tool, as shown in Figure 7.2, so that users no longer need to judge which test cases need to be rerun, or have to wait for the entire test suite to be executed. Instead, the tool suite automatically selects the appropriate set of test cases based on the program changes, and the regression testing tool is used to automatically rerun them. This automation of the

regression testing process can be achieved by way of minor enhancements to the structure of the existing Automatic Test Script. Figure 7.4 illustrates these enhancements.

```
/* example.ats - Description of an Automatic Test Script */
DEFINE GROUP example
/* Test group defines a series of test cases */
     {
        DEFINE CASE first.test
           /* Illustrates the structure of a test case */
           {
             SOURCE
             "Comment to explain origin of this test case";

             ACTIVATION
             "System commands to execute this test case";

             OBJECTIVE
             "Value for test cost, time, or priority";

             TAG
             "example.ats" and "example" and "first.test";

             TRACE
             "first.trc";

             EVALUATION WITH BASELINE
             "first.out" vs. "first.baseline";
           }

        DEFINE CASE second.test

           ...
     }
```

Figure 7.4 : Enhanced Features of an Automatic Test Script

Three new elements must be added to the test script: a) a test objective (OBJECTIVE) is defined, which represents the cost factor associated with each test case; b) a test tag (TAG) is specified, which contains the names of the current test script, test group, and test case, and c) a tracefile name (TRACE) is given, which the tool suite requires for the purposes of structural testing.

7.3 Description

In the following section, the functionality of each component in the tool suite is described and its contribution towards realising a technique for selective revalidation is emphasized. Implementing an automated system, which assists in the analysis and retesting of modified software, provides a number of benefits for maintenance programmers. They would no longer need to spend a considerable amount of time in analysing the program code to determine the extent of a given modification and in selecting, what they believe, is an adequate set of regression tests. Instead, their efforts would be focussed on locating the faulty program code and deciding which change implementation is the most cost effective. Given a set of change requests, the tool suite would allow maintenance programmers to judge which test cases can be rerun using the current level of resources. Moreover, the proposed tool suite would permit a more thorough (re)testing of the program code than could be achieved manually. This improves the confidence of maintenance programmers in the correctness of their software.

7.3.1 Program Instrumentation

Program instrumentation forms the first stage of code analysis conducted by the tool suite. Its objective is to perform static analysis of C program modules and to prepare them for the purposes of dynamic testing. For program instrumentation to be successful, the program code must be syntactically correct. It is then preprocessed in order to expand any conditional compilation directives and macros that may be present. Furthermore, any program comments are removed.

As the tool suite performs structural testing of individual program modules, program instrumentation entails the insertion of *probes* at every conditional statement (branch) and sequence of statements (segment) within a program module. These probes are later executed during dynamic testing to allow the monitoring of different structural test coverage criteria. In general, the introduction of probes into the existing source code

affects the size and performance of the instrumented source code as they represent additional lines of code.

Apart from generating an instrumented version of the original source file, a corresponding *reference listing* is produced for each source file. The listing highlights, by means of comments, the probes that are inserted into the source code, and contains various program statistics for each program module. These statistics include the number of executable lines of code, a summary of the different language constructs used, Halstead's Software Science [88] and McCabe's cyclomatic number complexity metric [176].

7.3.2 Program Flow Analysis

Program flow analysis represents the second stage of code analysis conducted by the tool suite. The component performs interprocedural data-flow analysis of the source code, which when completed, allows the control dependency and data dependency for each program module to be determined and represented in a Program Dependency Graph. At the same time, the test requirements for each program module are derived; a list of statements, branches, path expressions and definition-use associations are produced.

Apart from generating the information needed for selective revalidation, program flow analysis also collates information concerning the overall structure of the program. In particular, the program call-graph is generated which lists both direct and indirect module calls and provides statistics concerning the number of user-defined modules in the program, the number of interfaces per module and, more importantly, any unreachable modules. Such details can prove to be useful to maintenance programmers when debugging and help them in understanding the program structure.

7.3.3 Test Coverage Analysis

When requested, dynamic testing of individual program modules commences using the instrumented program code and the set of test requirements derived during program flow analysis. Test coverage analysis performs three important functions: a) the selection of structural test coverage criteria, b) the compilation of the instrumented program and its execution with test data, and c) the assessment of the results to ensure the adequacy of the test data.

Before test execution, users specify their required test coverage criteria with the component providing a selection of control-flow and data-flow coverage criteria, as described in Section 2.3.1-2.3.2. In addition, users define a test coverage limit for the corresponding criteria. This value represents a threshold value at which the code has been sufficiently tested, the coverage criteria have been satisfied, and the test data is deemed adequate.

In order to perform dynamic testing of the program code, test coverage assessment provides a run-time interface for compiling the instrumented code and executing it with test data. Users can be delivered to a Unix command-shell from within the environment which allows them to perform these tasks. During test execution, the probes that were inserted during program instrumentation, are activated. They result in a set of tracefiles being generated which contain information about the branches and segments executed during each test execution. However, the generation of these tracefiles produces a run-time overhead for the instrumented program which is proportional to the number of times a call is made to the component's corresponding probe function.

Test adequacy is assessed by means of a cumulative analysis. The component judges test adequacy by referring to the existing tracefiles, the set of test requirements, the current test coverage criteria, and the test coverage limit. A test coverage profile is generated for each module, which indicates the different test requirements, test coverage

criteria, and the specified test coverage limit. It indicates which test requirements have been exercised by the existing set of test cases. If the test coverage criteria have not been satisfied to the specified limit, additional test data must be generated by users to validate the untested program code. This task is made easier through the use of the reference listing, which was produced during program instrumentation. The entire procedure is then repeated until the specified test coverage limit is reached. Once satisfactory test coverage has been achieved, a test summary report is produced by the component. It considers the testing history of the program in terms of the different test requirements, test coverage criteria, and the specified test coverage limit.

7.3.4 Change Analysis

The change analysis component relies upon information obtained from program flow analysis and the set of proposed program modifications which are specified by users. Each of these modifications needs to be presented in terms of a changed program statement, which has been either added, deleted, or modified, within a program module. Subsequently, the component determines, by means of the Program Dependency Graph, those program statements affected by each proposed modification and lists the corresponding set of test requirements.

7.3.5 Test Suite Management

Test suite management encompasses two important tasks: test selection and test update. For test selection, a decision model is formulated based on information obtained during change analysis and test coverage analysis. The model constraints are formed by combining the testing history of a program module with the set of affected test requirements resulting from each program modification. Before solving the decision model, both the test costs and requirements in the model can be adjusted, if necessary. A new decision model must be established and solved for each program modification being investigated.

After the proposed program modification has been implemented, the resulting set of retestable tests is executed using the modified program. Test execution histories are subsequently updated during test coverage analysis; if any new test requirements have resulted from the modification, then additional test data may be needed to exercise them. After the modified code has been tested, the test suite is analysed in order to determine any redundant test cases which can then be eliminated by users. A decision model, similar to that defined for test selection, is established and solved. The test update procedure is based on the current testing history of the program and does not involve the use of any change analysis information.

Three different algorithms have been implemented to solve the decision models. They include the cutting-plane method, developed by Gomory, the implicit enumeration algorithm, described by Balas, and the new heuristic algorithm developed in this thesis. In the case of test selection, the model solution presents details concerning the proposed program modification, the identity of the retestable tests, the total cost associated with these tests, and the type of algorithm used to solve the model. For test update, the model solution simply identifies the set of redundant tests.

7.4 Summary

In this chapter, the development of a tool suite known as **RETEST** is described which concentrates on the selective revalidation of program modules during maintenance. The objective of this tool suite is to support the activities associated with the technique for selective revalidation described in this thesis. Apart from providing automated support for the initial validation of the software, the tool suite includes facilities for change analysis and test suite management. As regression testing tools are currently lacking such facilities, it is envisaged that the tool suite will be used in conjunction with these tools in order to complement and enhance current practices.

Apart from outlining the design of the proposed tool suite, emphasis is placed on describing its interface to current regression testing tools. By developing such an interface, the possibility exists for the complete automation of the regression testing process. Furthermore, the implementation of the tool suite is briefly described, with the role of each component with respect to selective revalidation being examined.

Chapter 8

Evaluation of Selective Revalidation

8.1 Introduction

Few studies have been conducted to demonstrate the benefits and limitations of selective revalidation. The work by Leung-White [158] represents the *only* experimental study which has so far attempted to evaluate a technique for selective revalidation against the retest-all strategy. The study makes a number of interesting observations concerning test selection. Compared with the retest-all strategy, significant savings of upto 40% were achieved in the number of retestable tests selected during unit testing of the modified software. This figure, however, could be improved through a better choice of revalidation criterion as the current criterion did not allow an optimal set of retestable tests to be selected. Furthermore, the authors noted that the extent of a proposed modification on an existing piece of program code was not a deciding factor in predicting the number of regression tests which needed to be rerun. Their observation, however, was influenced by the fact that their selective revalidation technique did not include a change analysis technique and therefore could not assess the impact of a modification on the program code. Instead, the authors concentrated on examining the effects of changes on the program functionality.

The primary objective of this evaluation is to compare the technique for selective revalidation, developed in this thesis, with both the retest-all strategy *and* representative selective revalidation strategies described in Chapter 4. The comparison is made possible by the fact that the technique described in this thesis is independent of the underlying testing strategy and can therefore be examined alongside other selective revalidation techniques based on *structural testing* strategies. The evaluation aims to assess the usefulness of the change analysis and operations research techniques. While this evaluation restricts itself to examining techniques for test selection, it is hoped that future

work will allow a similar evaluation to be conducted for those techniques which address the problem of test update.

In this chapter, a case study is undertaken to demonstrate the benefits and limitations of the technique for selective revalidation developed in this thesis. In Section 8.2, the application of the technique is described in three distinct phases: a) initial validation, b) modification, and c) revalidation of the sample program. Each phase is discussed in terms of the code analysis techniques and procedures for test suite management developed in Chapters 5-6. Section 8.2.1 describes the initial validation of the sample program using two different structural testing strategies. In Section 8.2.2, maintenance modifications are applied to a range of program statements. Thus, the effectiveness of the selective revalidation technique, developed in this thesis, can be compared with other selective revalidation techniques in terms of change analysis. Section 8.2.3 describes the test selection procedure and demonstrates how the selective revalidation technique utilises operations research to select a set of retestable tests. Finally, Section 8.3 analyses and summarises the results of the evaluation.

Although any conclusions drawn from this case study may be insufficient to demonstrate the effectiveness of this selective revalidation technique for a broad class of programs or maintenance activities, they can nevertheless be used to highlight a number of important properties or characteristics of the technique. In future research, it is hoped that a larger sample of programs can be analysed to confirm the results and analyses presented in this thesis.

8.2 Application

8.2.1 Initial Validation

The sample program to be analysed, modified, and retested is shown in Figure 8.1. The program was chosen based on two selection criteria, namely program size and complexity. While program size was measured as lines of executable code (37), the complexity was determined in terms of McCabe's cyclomatic complexity number [175]. These criteria provided an indication of whether or not the sample program contained a sufficient number of variable interactions and distinct program paths for the case study to be non-trivial and yet manageable in terms of analysis and comprehension.

McCabe devised a measure of program complexity V(G) using graph theoretic techniques. His theory maintained that program complexity was not dependent upon program size, but on the control structure of the program. Measurement of the complexity of a program depends on transforming the program so that it is represented in terms of its control-flow graph G, and counts the number of nodes n, edges e and connected components in a graph; thus, the complexity of a program can be calculated using V(G)=e-n+2. Alternatively, the metric can be determined by adding one to the number of decision statements in the program. For the sample program, therefore, the McCabe's complexity number is six.

The McCabe complexity metric has some validity, but it suffers from a number of disadvantages. It does not take into account the data structures used in the program, the program comments, or the use of meaningful variable names. However, the metric was chosen as a selection criterion for this case study, as it is widely used and accepted in the software engineering community despite its apparent disadvantages.

```
      program example
      begin
1:      read(a,b,c,d);
2:      f = a + d;
3:      if(a > 0) then
            begin
4:            if(b > 0) then
                  begin
5:                  b = b + 1;
                  end
              else
                  begin
6:                  a = a + 1;
                  end
            end
        else
            begin
7:            if(c > 0) then
                  begin
8:                  if(d > 0) then
                        begin
9:                        a = a + 2;
                        end
                    else
                        begin
10:                       a = a + 3;
                        end
                  end
              else
                  begin
11:                 c = c + 1;
                  end
            end
12:     if(f > 0) then
            begin
13:           e = b + c;
            end
14:     write(a,b,c,d,e,f);
      end
```

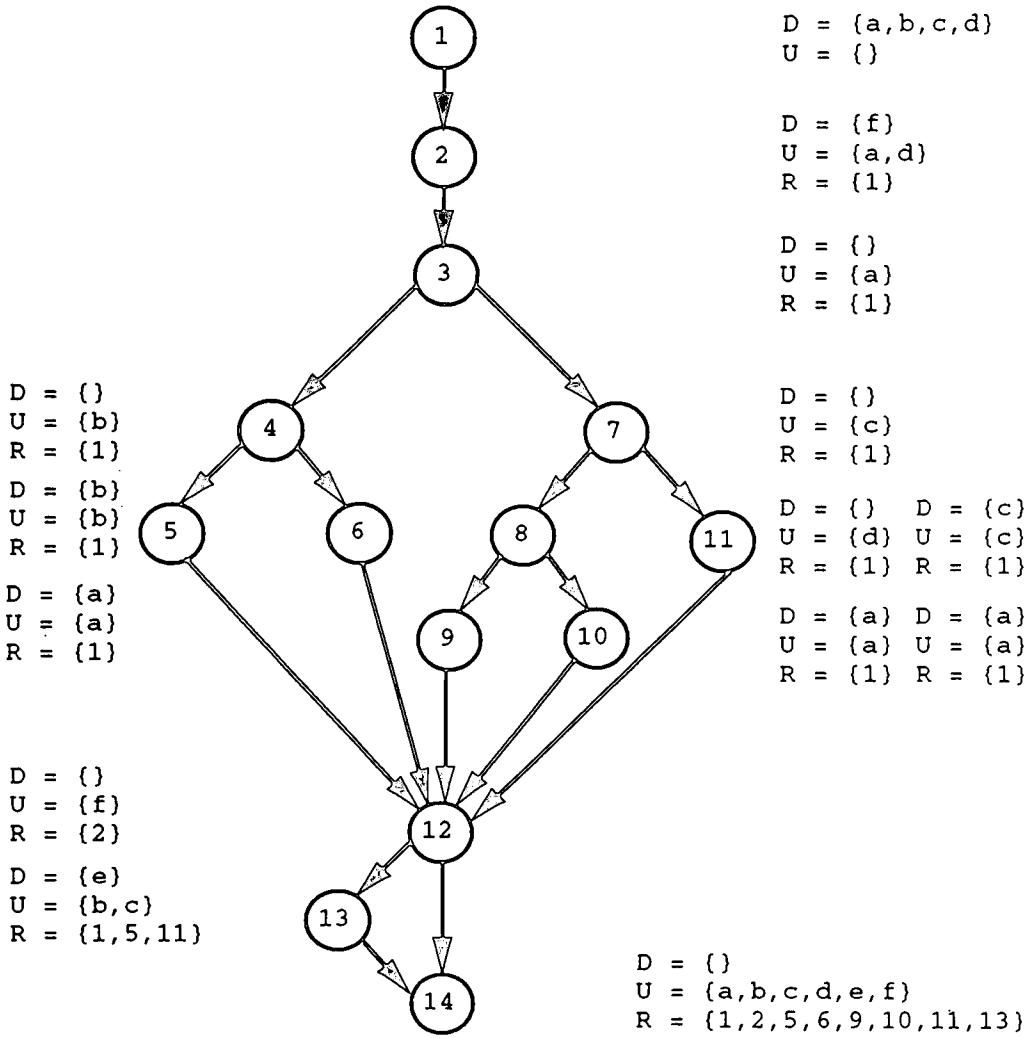


Figure 8.1: Sample Program and Graph Used in Evaluation

Instrumentation and flow analysis prepared the sample program for testing and allowed the necessary control dependency and data dependency information to be derived; the resulting data-flow graph is shown in Figure 8.1. Test requirements, in particular, path expressions and definition-use associations were determined for the sample program. These were required as the technique for selective revalidation was to be compared not only with the retest-all strategy, but also with two selective revalidation techniques, one of which was based on the path testing criterion and the other on the all-uses data-flow testing criterion. In fact, flow analysis identified a total of ten paths and thirty-two definition-use associations in the program.

The sample program was then validated using a combination of functional and structural testing techniques. A test case order was maintained whereby the imposed structural test coverage criterion was first satisfied using *functional* test cases and later supplemented with additional *structural* tests. Functional tests were derived from the natural language specification of the sample program and defined using, for example, boundary-value analysis and equivalence partitioning. With boundary-value analysis, extremal data values close to the boundary input values specified for the program were chosen. The concept of equivalence partitioning was then used to supplement this set of input values by selecting additional values from within the typical operating range of the program's input domain.

The resulting test suites for the sample program are shown in Figure 8.2, with the testing history in Figure 8.2(a) being based on path testing and the testing history in Figure 8.2(b) being based on the all-uses data-flow testing. Although the test cases generated are labelled T_1-T_{10} and T_1-T_7, respectively, they do not necessarily represent the same test inputs; the test data used to satisfy the data-flow testing criterion consisted of a subset of test cases used for path testing.

Program Statement	Test T_1	Test T_2	Test T_3	Test T_4	Test T_5	Test T_6	Test T_7	Test T_8	Test T_9	Test T_{10}
1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	0	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0	0
6	0	0	1	1	0	0	0	0	0	0
7	0	0	0	0	1	1	1	1	1	1
8	0	0	0	0	1	1	1	1	0	0
9	0	0	0	0	1	1	0	0	0	0
10	0	0	0	0	0	0	1	1	0	0
11	0	0	0	0	0	0	0	0	1	1
12	1	1	1	1	1	1	1	1	1	1
13	1	0	1	0	1	0	1	0	1	0
14	1	1	1	1	1	1	1	1	1	1

Figure 8.2(a): Testing History Based on Path Coverage

Definition-Use Association	Test T_1	Test T_2	Test T_3	Test T_4	Test T_5	Test T_6	Test T_7
(a,1,2)	1	1	1	1	1	1	1
(a,1,(3,4))	1	1	1	0	0	0	0
(a,1,(3,7))	0	0	0	1	1	1	0
(a,1,6)	0	0	1	0	0	0	0
(a,1,9)	0	0	0	1	1	0	0
(a,1,10)	0	0	0	0	0	1	0
(a,1,14)	1	1	1	1	1	1	1
(a,6,14)	0	0	1	0	0	0	0
(a,9,14)	0	0	0	1	1	0	0
(a,10,14)	0	0	0	0	0	1	0
(b,1,(4,5))	1	1	0	0	0	0	0
(b,1,(4,6))	0	0	1	0	0	0	0
(b,1,5)	1	1	0	0	0	0	0
(b,1,13)	1	0	1	1	0	0	1
(b,1,14)	1	1	1	1	1	1	1
(b,5,13)	1	0	0	0	0	0	0
(b,5,14)	1	1	0	0	0	0	0
(c,1,(7,8))	0	0	0	1	1	1	0
(c,1,(7,11))	0	0	0	0	0	0	1
(c,1,11)	0	0	0	0	0	0	1
(c,1,13)	1	0	1	1	0	0	1
(c,1,14)	1	1	1	1	1	1	1
(c,11,13)	0	0	0	0	0	0	1
(c,11,14)	0	0	0	0	0	0	1
(d,1,2)	1	1	1	1	1	1	1
(d,1,(8,9))	0	0	0	1	1	0	0
(d,1,(8,10))	0	0	0	0	0	1	0
(d,1,14)	1	1	1	1	1	1	1
(e,13,14)	1	0	1	1	0	0	1
(f,2,(12,13))	1	0	1	1	0	0	1
(f,2,(12,14))	1	1	1	1	1	1	1
(f,2,14)	1	1	1	1	1	1	1

Figure 8.2(b): Testing History Based on All-Uses Coverage

8.2.2 Modification

During the experimentation conducted by Leung-White, modifications were specified in response to a set of change requests. However, the locations of these modifications within the program code were, in effect, *ad hoc*. No systematic approach was taken by the authors to examine the performance of the associated selective revalidation technique for a wide range of maintenance scenarios. Instead, Leung-White concentrated on the analysis of a small set of *ad hoc* program changes from which they drew conclusions concerning the performance of their selective revalidation technique.

However, the evaluation described in this thesis uses a more systematic approach by first classifying the program statements in terms of their *criticality*. This criticality is determined by an inspection of the program's control structure whereby statements of *high criticality*, *medium criticality* and *low criticality* are identified according to whether they were traversed by most, some or relatively few program paths, respectively. Subsequently, modifications to statements of high, medium and low criticality are categorised into Type I, Type II and Type III modifications, respectively. For example, a Type I modification is associated with statements of high criticality such as the entry statement (1) and exit statement (14) of the sample program. Modifications to the sample program are undertaken as changes to individual statements. The change analysis technique is then applied to each variable definition and use in the modified statement. Consequently, a set of affected program statements, that is test requirements, is determined.

The application of the change analysis technique to statements of different criticality enabled its effectiveness to be evaluated for a range of possible maintenance scenarios. It also allowed the benefits and limitations of the selective revalidation technique, in selecting the set of affected test requirements, to be demonstrated. Compared with the two other selective revalidation strategies, which relied upon a manual inspection of the testing

history to select their affected statements or definition-use associations, the change analysis technique can be used to systematically determine the extent of each modification.

To achieve this, the change analysis technique relies upon both the control dependency and data dependency derived during flow analysis. This information is often displayed using its graphic representation. Figure 8.3, however, represents it in the form of matrices. While the control dependency matrix embodies both the `ControlSucc` and `ControlPred` relations associated with the Control Dependency Graph, the data dependency matrix represents the `ReachableNodes` relation, that is the transitive closure of the Data Dependency Graph. This matrix, which forms the reachability matrix of the data dependency matrix, depicts both direct and indirect interactions between each variable use and its corresponding set of definitions in the program. Direct and indirect interactions are indicated by means of the non-zero values in the matrix.

Reaching Statements \ Reached Statements

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	1	0	0	0	0	0	0	0
4	0	0	0	0	1	1	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	1	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Control Dependency

Variable Uses \ Variable Definitions

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	0	0	0
12	1	1	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	0	0	1	0	0	0	0	0	1	0	0	0
14	1	1	0	0	1	1	0	0	1	1	1	0	1	0

(b) Data Dependency

Figure 8.3 : Dependency Information for the Sample Program

All variable interactions in the reachability matrix are listed according to the program statements in which they occur. By calculating the transitive closure of the program's data dependency, the change analysis technique can determine the possible extent of a program

modification with respect to every statement in the program. The effects of the modification are then expressed in terms of the number and identity of affected program statements or definition-use associations.

Apart from the control dependency matrix, the row and column vectors of the reachability matrix, which are associated with the modified program statement, are examined. For example, a proposed modification to statement 13 in the sample program yields two reachability matrix bit-vectors, [10001000001010] and [00000000000011]. They correspond to those variable definitions, which directly or indirectly may affect the value of the computation being performed in statement 13, and those variable uses possibly affected by the changed value. Subsequently, a total of five statements (1, 5, 11, 13 and 14) and seven definition-use associations ((b,1,5), (b,1,13), (b,5,13), (c,1,11), (c,1,13), (c,11,13) and (e,13,14)) are implicated. In constrast, the two existing selective revalidation techniques either overestimate the number of affected program statements or underestimate the number of affected definition-use associations. While the technique based on path testing causes virtually all program statements to be flagged as affected, the technique based on data-flow testing identifies a total of six definition-use associations ((b,1,13), (b,5,13), (c,1,13), (c,11,13), (e,13,14) and (f,2,(12,13))). The implications of such over- and underestimates are demonstrated in this evaluation.

8.2.3 Revalidation

Revalidation of the sample program entailed the selection of a set of retestable tests based on the affected test requirements identified during the modifications. Using the two test suites created during initial validation, the retest-all strategy simply required the entire test suite to be rerun. For the two representative selective revalidation techniques, test selection criteria were applied which involved a manual inspection of the corresponding testing history. These criteria specified that all test cases, which traversed the affected test requirements, needed to be rerun. For example, the modification of statement 13 in the sample program required five test cases (T_1, T_3, T_5, T_7 and T_9) to be selected for rerun

from the testing history shown in Figure 8.2(a) and four test cases (T_1, T_3, T_4, T_7) to be selected from the testing history shown in Figure 8.2(b).

However, the technique for selective revalidation developed in this thesis can be used to further reduce the number of retestable tests. Thus, for the modification of statement 13, two decision models are defined; one of which is based on the path testing history, the other on the data-flow testing history. In both cases, an objective function **Z** is specified in terms of a set of decision variables **x** which represented the number of test cases in the respective test suite. The aim of the objective function is to select a minimal number of retestable tests, with a set of constraints being established to reflect the affected testing history[1].

Equations 8.1-8.6 depict a decision model which reflects the test cases and test requirements associated with a proposed modification of statement 13; the model is based on the path testing history illustrated in Figure 8.2(a). (Note that the test requirements in the leftmost column represent affected program statements which are used to facilitate the discussion of each decision model and do not form an actual part of the decision model.)

$$Z = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \tag{8.1}$$

$$1 \quad x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \geq 1 \tag{8.2}$$

$$5 \quad x_1 + x_2 \geq 1 \tag{8.3}$$

$$11 \quad x_9 + x_{10} \geq 1 \tag{8.4}$$

$$13 \quad x_1 + x_3 + x_5 + x_7 + x_9 \geq 1 \tag{8.5}$$

$$14 \quad x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \geq 1 \tag{8.6}$$

Similarly, Equations 8.7-8.14 represent a decision model which is based on the data-flow testing history shown in Figure 8.2(b). (Note that the test requirements in the leftmost column represent affected definition-use assocations which are used to facilitate

[1] Recall that for a modification of statement 13, the technique identified five statements (1, 5, 11, 13 and 14) and seven definition-use associations ((b,1,5), (b,1,13), (b,5,13), (c,1,11), (c,1,13), (c,11,13) and (e,13,14)).

the discussion of each decision model and do not form an actual part of the decision model.)

$$Z = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \qquad (8.7)$$

(b,1,5)	$x_1 + x_2$			≥ 1	(8.8)
(b,1,13)	$x_1 +$	$x_3 + x_4 +$	x_7	≥ 1	(8.9)
(b,5,13)	x_1			≥ 1	(8.10)
(c,1,11)			x_7	≥ 1	(8.11)
(c,1,13)	$x_1 +$	$x_3 + x_4 +$	x_7	≥ 1	(8.12)
(c,11,13)			x_7	≥ 1	(8.13)
(e,13,14)	$x_1 +$	$x_3 + x_4 +$	x_7	≥ 1	(8.14)

In order to solve the two decision models, the new heuristic method described in Section 6.4.5 is applied. In both instances, the solution consists of two test cases whereby the former decision model is satisfied by test cases T_1 (x_1=1) and T_9 (x_9=1), while the latter model is satisfied by test cases T_1 (x_1=1) and T_7 (x_7=1). These solutions represent an optimal set of test cases for traversing only the affected test requirements and thus affected program code. To compare the technique with the retest-all strategy, as well as the two other selective revalidation techniques, it was assumed that each decision model included equal test costs (c_j = 1) and constraint requirements ($b_i \geq 1$). This assumption enabled comparisons to be made solely in terms of the number of test cases and ensured that at least one test case was required to exercise the affected statement or definition-use association.

8.3 Results and Analysis

Tables 8.1-8.2 summarise the results obtained when the four different regression testing strategies were applied to the sample program and its associated testing histories. Statements were classified according to their criticality and modifications of Type I, II and III were performed. For each statement, the extent of the modification was then calculated. This was based on the testing criterion associated with the respective testing

history. While the selective revalidation technique based on path testing identified all those statements which lay on program paths traversing the modified statement, the technique based on data-flow testing determined the affected definition-use associations. In constrast, the technique developed in this thesis used change analysis to systematically select the affected test requirements from both testing histories.

A decision model was then established and solved, with the corresponding number of retestable tests being recorded. The performance of each selective revalidation technique was measured in terms of the reduction or savings achieved in the number of retestable tests. A metric was developed in which the percentage reduction achieved by the selective revalidation technique was defined as the ratio of the number of retestable tests to the total number of test cases in the respective test suite.

Modification	Type I					Type II				Type III				
Statement	1	2	3	12	14	4	7	8	13	5	6	9	10	11
Extent of Modification (Statements)	14	7	12	7	9	6	8	5	5	4	3	3	3	4
Number of Retestable Tests (My Approach)	5	2	5	2	5	2	3	2	2	1	1	1	1	1
Affected Statements (Other Approach)	14	14	14	14	14	9	11	10	14	8	8	10	9	8
Number of Retestable Tests (Other Approach)	10	10	10	10	10	4	6	4	5	2	2	2	2	2
Total Number of Test Cases (Retest-all Approach)	10	10	10	10	10	10	10	10	10	10	10	10	10	10
Retestable Tests/Total Tests in % (My Approach)	50	20	50	20	50	20	30	20	20	10	10	10	10	10
Savings in Test Cases in % (My Approach)	50	80	50	80	50	80	70	80	80	90	90	90	90	90
Retestable Tests/Total Tests in % (Other Approach)	100	100	100	100	100	40	60	40	50	20	20	20	20	20
Savings in Test Cases in % (Other Approach)	0	0	0	0	0	60	40	60	50	80	80	80	80	80

Table 8.1: Revalidation Statistics Based on the Path Testing Criterion

Modification	Type I					Type II				Type III				
Statement	1	2	3	12	14	4	7	8	13	5	6	9	10	11
Extent of Modification (Definition-Use Assocs.)	31	12	23	11	22	8	12	6	7	4	2	2	2	4
Number of Retestable Tests (My Approach)	5	2	5	2	5	2	3	2	2	1	1	1	1	1
Affected Definition-Use Assocs. (Other Approach)	21	5	2	2	12	3	3	3	6	4	3	3	3	4
Number of Retestable Tests (Other Approach)	7	7	7	7	7	3	4	3	4	5	1	2	1	1
Total Number of Test Cases (Retest-all Approach)	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Retestable Tests/Total Tests in % (My Approach)	71	28	71	28	71	28	43	28	28	14	14	14	14	14
Savings in Test Cases in % (My Approach)	29	72	29	72	29	72	57	72	72	86	86	86	86	86
Retestable Tests/Total Tests in % (Other Approach)	100	100	100	100	100	43	57	43	57	71	14	28	14	14
Savings in Test Cases in % (Other Approach)	0	0	0	0	0	57	43	57	43	29	86	72	86	86

Table 8.2: Revalidation Statistics Based on the All-Uses Data-Flow Testing Criterion

The different selective revalidation techniques were compared against the retest-all strategy in terms of the reductions or savings attained in the number of retestable tests. This revealed that the traditional selective revalidation techniques provided an average saving of approximately 40%, which confirmed the results obtained by Leung-White during their empirical study. However, the technique developed in this thesis was able to improve on this figure and attain average savings of 77% and 67% when applied with respect to a test suite consisting of a path testing and data-flow testing history, respectively. This suggested that the technique could provide an average improvement in savings of 69% over existing selective revalidation techniques. Tables 8.3-8.4 summarise the savings attained in the number of retestable tests with respect to the criticality of the modified program statements.

Criticality of Statements	High	Medium	Low	Average
Average Savings (Other Approach)	0%	53%	80%	44%
Average Savings (My Approach)	62%	78%	90%	77%

Table 8.3: Savings Attained for the Path Testing History

Criticality of Statements	High	Medium	Low	Average
Average Savings (Other Approach)	0%	50%	72%	41%
Average Savings (My Approach)	46%	68%	86%	67%

Table 8.4: Savings Attained for the Data-Flow Testing History

Tables 8.3-8.4 also highlight the fact that the three techniques for selective revalidation chosen for this study provide the most significant savings whenever they are used to revalidate modified statements of low criticality. As these statements tend to be traversed by fewer program paths, and their modification implicates few data dependencies within the program, then substantial savings can be attained by the selective revalidation techniques compared with the retest-all strategy. While the existing techniques select their retestable tests based on an inspection of the test suite, the

technique described in this thesis achieves its slight improvement in savings by examining the existing variable interactions, which induce a ripple effect, and attempting to optimise the resulting set of affected test cases.

Tables 8.3-8.4 also indicate that the modification of highly critical statements results in the least amount of savings. Statements of high criticality are usually traversed by the majority of program paths and contain computations that interact with variables throughout the program code. Therefore, these statements tend to implicate the majority of statements when modified. In such circumstances, traditional selective revalidation techniques can provide little, if any, savings, with all test cases in the respective test suite needing to be rerun. However, the technique described in this thesis can attain a substantial reduction in the number of retestable tests by using change analysis to trace the ripple effects induced by the modified variables and operations research to determine an optimal set of tests.

Tables 8.5-8.6 support the notion that change analysis is especially useful in the selective revalidation of highly critical statements and that it should form an essential part of any selective revalidation technique. The two tables illustrate the extent of each modification in terms of the number of affected program statements and definition-use associations. They reveal that the technique based on path testing produces a persistent *overestimate* in the number of affected statements with respect to the technique described in this thesis. Conversely, the technique based on data-flow testing tends to *underestimate* the number of affected definition-use associations. This trend is particularly evident when considering the extent of any modifications due to highly critical statements.

Criticality of Statements	High	Medium	Low	Average
Affected Statements (Other Approach)	14	11	9	11
Affected Statements (My Approach)	10	6	3	6

Table 8.5: Extent of Modifications for the Path Testing Criterion

Criticality of Statements	High	Medium	Low	Average
Affected Definition-Use Associations (Other Approach)	8	4	3	5
Affected Definition-Use Associations (My Approach)	20	8	3	10

Table 8.6: Extent of Modifications for the All-Uses Data-Flow Testing Criterion

The technique based on data-flow testing only examines *first order* ripple effects and therefore identifies only those subpaths in the immediate vicinity of the modified statement. This, however, means that the technique underestimates the number of affected definition-use associations in the presence of any higher order ripple effects, and subsequently *overestimates* the number of retestable tests. Similarly, the technique based on path testing overestimates the number of retestable tests. In this case, the technique disregards any ripple effects and simply assumes that a modified statement is associated with statements on all program paths exercising it - the number of affected statements is overestimated. However, the change analysis technique described in this thesis is more discerning. In the presence of higher order ripple effects, it not only identifies directly, but also indirectly affected statements. Subsequently, long sequences of interrelated variable definitions and uses help the retesting effort to focus on possibly affected program paths and assist in selecting the retestable tests. As suggested above, the technique for selective revalidation performs particularly well for statements which induce a high order ripple effect.

8.5 Summary

In this chapter, a case study was conducted in order to evaluate the technique for selective revalidation developed in this thesis. Using a sample program, which was initially validated using two different structural testing strategies and then modified, comparisons were made with both the retest-all strategy and representative selective

revalidation strategies. The evaluation assessed the usefulness of change analysis and operations research techniques as part of a selective revalidation technique, and also examined the factors influencing the effectiveness of the selective revalidation technique described in this thesis.

A number of interesting observations were made as a result of this evaluation. It was found that considerable savings in the number of retestable tests were achieved not only with respect to the retest-all strategy, but also with respect to the two traditional selective revalidation techniques. In fact, an average reduction of 72% was attained in comparison with the retest-all strategy, together with an improvement of nearly the same amount over existing selective revalidation techniques.

Although the technique described in this thesis attained the largest reduction in test cases when applied to the modification of low criticality statements, its change analysis and operations research techniques proved to be most beneficial when retesting changes to highly critical statements. For each type of modification, the systematic identification of affected test requirements, as well as the efficient reduction of the corresponding set of test cases, enabled the technique to significantly reduce the number of retestable tests. The technique proved that it could maintain an appreciable reduction in the number of retestable tests for a range of maintenance modifications.

Chapter 9
Conclusions

9.1 Contributions

The work described in this thesis has addressed the subject of regression testing with particular emphasis placed on developing a technique for selective revalidation. The technique has been applied to the regression testing of programs during the maintenance phase of the software lifecycle. It is intended to help maintenance programmers analyse and retest their software in a systematic and efficient manner. This technique, together with the supporting software tools, can lead to a reduction of resources required for regression testing and can improve the confidence of maintenance programmers by ensuring that their modifications have been adequately tested. To achieve this, the technique involves the application of code analysis and operations research.

Code analysis techniques have been developed in order to *systematically* derive information about the structure of a program and assess the impact of any proposed modifications on the existing program code. In particular, these techniques have been directed at the analysis of programs written in the C programming language. Techniques for dependency analysis have been developed to examine the control dependency and data dependency of a program and depict them by way of a graphical representation known as the Program Dependency Graph. Emphasis has also been placed on addressing the problems of pointer variables and their aliases, as well as examining the dependencies which arise from them.

In response to a proposed program modification, the existing program code has to be analysed in order to determine which parts of the code could be directly or indirectly affected by the change. A change analysis technique has therefore been developed to examine the dependencies depicted in the Program Dependency Graph and to identify the

affected program statements. The technique can then be used to select those test requirements, that is test cases, which exercise the affected parts of the program.

Apart from code analysis, the technique for selective revalidation has addressed the issue of test suite classification in which test cases are categorised into reusable, retestable, redundant and new tests. It has subsequently highlighted the problems of test suite management, namely test selection and test update, and proposed solutions to these problems. A detailed description of the technique has been given to show how the concept of selective revalidation can be integrated into the traditional maintenance cycle.

Techniques adopted from operations research have been used to *efficiently* select a set of retestable tests during test selection and identify any redundant tests in the test suite during test update. By formulating the problems of test selection and test update as decision problems, it has become possible to consider a wide range of regression testing objectives and constraints. In fact, the corresponding decision models can consider objectives, such as the number, cost and priority of test cases, and constraints resulting from the use of different structural testing techniques as well as the testing requirements associated with critical parts of the program code. In addition, the decision models can accommodate the situation whereby several testing objectives need to be considered simultaneously.

Both optimal and approximation algorithms have been examined in order to find a way of solving the decision models associated with the problems of test selection and test update. Emphasis is placed on describing cutting-plane methods and implicit enumeration techniques which can provide optimal solutions to these problems. However, due to their potentially exponential run-time, such algorithms may not be able to produce a solution to larger problems in a reasonable amount of time. Therefore, a new heuristic method, which compares favourably with both optimal and approximation algorithms in terms of the size of its solutions and its worst-case run-time complexity, has been developed.

When working with large and complex programs, it is impractical to analyse and retest the software without the aid of suitable software tools. A tool suite has therefore been designed to realise the technique for selective revalidation and automate the tasks associated with code analysis and test suite management. In particular, the interface between existing regression testing tools and the tool suite is described. Together, these tools can potentially lead to the complete automation of regression testing.

A case study is presented which evaluates the benefits and limitations of the selective revalidation technique described in this thesis. In this study, the technique is compared with both the retest-all strategy and other selective revalidation strategies described in this thesis. Its results indicate that, for a range of maintenance modifications, the technique selects on average 72% fewer test cases compared with the retest-all strategy, with a similar saving in test cases being attained with respect to the existing selective revalidation techniques based on path testing and data-flow testing. The effectiveness of the technique appears to be directly dependent upon the location and extent of the proposed program modifications.

9.2 Future Directions

9.2.1 Code Analysis Techniques

The work described in this thesis concentrates on the development of code analysis techniques for individual program modules, with the resulting control dependency and data dependency information being represented by a Program Dependency Graph. These techniques, however, do not allow a program, which may consist of a collection of modules, to be examined. One way of overcoming this problem would be to construct a Program Dependency Graph by in-line substitution; this would involve replacing every call to a program module by its corresponding control dependency and data dependency information.

However, this representation would be prohibitive for all, but the most trivial, of programs. An alternative approach is to develop a System Dependency Graph, whereby a Program Dependency Graph is constructed for each module in the program and the individual Program Dependency Graphs are then linked via two types of dependency: *calling dependency* and *parameter dependency*. The calling dependency reflects the control dependency existing between two program modules and intends to capture their calling context. With respect to a collection of modules, this interaction is often illustrated by means of the program call-graph. The parameter dependency reflects the data dependency, which may arise between two program modules as a result of *parameter aliasing* where two distinct variables simultaneously accessing the same memory location. In this case, a dependency would be created between each actual parameter of the calling module and the corresponding formal parameter of the called module. Global variables would then be treated as additional module parameters.

The development of a System Dependency Graph would allow the existing change analysis technique to be extended so as to enable it to trace the impact of any proposed maintenance modifications across the module boundaries. To achieve this, change analysis would rely upon the use of the calling dependency and parameter dependency information. It would be described as an iterative process characterised by the alternate application of *intra*procedural and *inter*procedural code analysis. For example, in response to a proposed modification, the dependencies would first be analysed within the context of a program module. If it is found that any of the interface variables, such as global variables and module parameters, are affected, then the change analysis technique would trace the corresponding dependencies across the respective module boundaries and repeat the process for each implicated program module.

A significant task, which has not been addressed in this thesis, is the development of incremental code analysis techniques. At present, any changes made to the program code result in a complete reanalysis of the modified code. However, incremental code analysis would allow a program's control dependency and data dependency information, as well

as its test requirements, to be updated without the need for such an exhaustive analysis. Operations could then be defined for incrementally updating the Program Dependency Graphs and System Dependency Graph.

The proposed tool suite was designed so that new tools could easily be integrated and existing ones upgraded. With some extensions, the existing intraprocedural code analysis techniques could be used to derive data dependencies and test requirements, such as interprocedural definition-use associations, and develop a System Dependency Graph. Consequently, the existing change analysis technique could be upgraded to enable it to examine these dependencies within the framework of a System Dependency Graph. Moreover, incremental parsing and data-flow techniques and tools would need to be developed, implemented and integrated into the tool suite.

9.2.2 Test Suite Management

The work described in this thesis concentrates on solving the problems of test suite management. Emphasis is placed on the development of procedures for test selection and test update, which ensure that modifications made to individual program modules can be efficiently revalidated. The application of operations research, however, has led to the development of a technique for selective revalidation which is independent of any testing strategy and could therefore be applied not only to the testing of code modules (unit testing), but also to the testing of the design specification (integration testing) and functional specification (system testing). It is therefore possible to extend the scope of the technique by applying it to the analysis and retesting of changes made not only to the implementation, but also to the specification and design of a program.

A functional specification usually describes the functionality of a program in terms of a set of *features*. During system testing, each feature is exercised by a set of test cases and the results are recorded in a *feature-test matrix*. All test cases are generated using functional testing and the feature-test matrix represents the testing history of the program

specification. Each entry in the matrix takes either the value zero, or one, depending upon whether, or not, a test case exercises a particular feature.

The design specification relates to the decomposition of each feature into a corresponding set of data structures or design functions[1]. A **structure chart** is established in which the design functions and their interactions may be represented. Each design function is then refined, which entails specifying the function in terms of a hierarchy of subordinate design functions. As the refinement progresses, the algorithms used in each design function can be specified, in detail, by means of pseudocode or flowcharts. If, for example, the design is based on the Jackson Structured Programming methodology (JSP) [128], or Warnier-Orr method [252], test cases may be derived from the structure chart and recorded in a *condition-test matrix* to represent the testing history of the program design. Each entry in this matrix takes either the value zero, or one, depending upon whether, or not, a test case exercises a particular design condition.

For the implementation, each design function is coded as either a program module or a collection of modules. These code modules can then be tested individually (unit testing) or collectively (integration testing). In each case, a test case order prevails insofar as the test cases derived from the design specification are executed first in order to exercise some components in the program code. Additional test cases are then used to validate the remaining program code. For integration testing, the test cases are recorded in a *module-test matrix* reflecting the testing history of the implementation. Each entry in this matrix takes either the value zero, or one, depending upon whether, or not, a test case exercises a particular program module.

During maintenance, each change proposal is decomposed into a set of basic modifications with respect to the program specification, design and implementation. Changes made to the each of these program attributes may be related to the addition,

[1] This approach assumes the use of a function-oriented design methodology.

deletion or modification of features, functions and code modules. In the case of perfective maintenance, for example, changes would need to be made to the program specification, design and code. Subsequently, the technique for selective revalidation, described in this thesis, would be applied in a top-down fashion whereby a proposed change at the specification level would also be analysed and retested at the design and code levels. For each program attribute, the procedure for test selection would examine the impact of the proposed modification and, based on the corresponding *requirements-test matrix*, select a set of retestable tests. The procedure for test update would then be invoked in order to update the respective testing history and ensure that the test coverage criterion with respect to that attribute is satisfied.

The technique described in this thesis therefore represents a consistent approach to selective revalidation in which the same procedures for test selection and test update can be applied to the analysis and retesting of a modified program specification, design and implementation. Thus, a proposed modification is validated not only with respect to each program attribute at a given level, but also with respect to all attributes at lower levels in the program hierarchy.

9.2.3 Operations Research

The work described in this thesis concentrates on developing a heuristic method which can be used to solve the problems of test selection and test update. However, further empirical evidence is required in order to determine the effectiveness of the heuristic and its application to problems of test suite management. To achieve this, different test suites will need to be examined whose characteristics vary in terms of the number of decision variables (test cases) and constraints (testing subsets), associated test costs and imposed constraint requirements. The algorithm would then be applied to each test suite in order to determine its cover. The size of this cover could then be compared to those generated using different optimal algorithms.

Appendix A
Glossary of Terminology

Acceptance Testing — Formal testing conducted to determine whether or not a program satisfies its acceptance criteria, that is, enable the customer to determine whether or not to accept the system.

Adaptive Maintenance — Maintenance activities performed to make a program usable in a changed operating environment. Contrast with: corrective maintenance; perfective maintenance; preventive maintenance.

Backward Data-Flow Problem — The problem associated with determining a set of statements affecting a given statement in terms of their control-flow and data-flow interactions with that statement.

Basic Blocks — A linear sequence of program statements with a single entry and exit point. Also referred to as a segment. Constrast with: Linear Code Sequence and Jump.

Bottom-Up Testing — Pertains to a testing activity which starts with the lowest-level components of a software system hierarchy and proceeds through progressively higher levels. Contrast with: top-down testing.

Branch Testing

Testing designed to execute each outcome of each decision point in a program. Contrast with: conditional testing; path testing; statement testing.

Call-Graph

A diagram, usually in the form a graph, which identifies the modules in a software system and shows which modules call one another.

Cause-Effect Graphing

Testing strategy in which test cases are developed based on combinations of input conditions (causes) and their expected outputs (effects). Causes and effects are related by means of a graph which is derived from a program's specification.

Code Analysis

The process of examining a program in order to gain some knowledge of its structure or execution behaviour.

Conditional Testing

Testing strategy in which the outcomes of the individual and overall predicate expressions at each decision point in the program are exercised by test cases. Contrast with: branch testing; path testing; statement testing.

Control-Flow Graph

A diagram, usually in the form of a graph, which depicts the control structure of a program and indicates the possible sequences in which operations may be performed during the

execution of a program.

Control-Flow Testing — Testing strategy which relies upon the control structure of a program as the basis for developing test cases.

Corrective Maintenance — Maintenance activities performed to correct faults in a program. Contrast with: adaptive maintenance; perfective maintenance; preventive maintenance.

Corrective Regression Testing — Regression testing that is applied whenever the program functionality remains unchanged, and the program modifications are restricted to the code.

Correctness — The extent to which software meets its specified requirements.

Data-Flow Graph — A diagram, usually in the form of a graph, which depicts the data structure of a program and indicates possible sequences in which variables are assigned and referenced during the execution of a program.

Data-Flow Testing — Testing strategy which relies upon the data structure of a program as the basis for developing test cases.

Def-Use Association	An ordered triple in which the first element identifies a given program variable, the second element depicts a program statement containing a definition of that variable, and the third element represents a statement or ordered pair of statements containing a corresponding use of the variable.
Directed Graph	A graph consisting of a finite set of nodes and a finite set of edges in which the direction of the edges between nodes is of importance.
Dynamic Testing	The process of evaluating a system or component based on its behavior during execution.
Entities	These relate to objects, types, values or modules that can be named or denoted in a program.
Forward Data-Flow Problem	The problem associated with determining a set of statements affected by a given statement in terms of their control-flow and data-flow interactions with that statement.
Functional Testing	Testing that ignores the internal mechanism of a system, or component, and focuses solely on the outputs generated in response to selected inputs and execution conditions; testing conducted to evaluate the compliance of a system or component with specified functional

requirements. Syn: black-box testing. Contrast with: structural testing.

Goal Programming A decision problem which encompasses the attainment of multiple objectives. See also: operations research.

Integer Programming A decision problem in which all problem parameters are expressed exclusively in terms of integer values. See also: operations research.

Integration Testing Testing in which software components are combined and tested to evaluate the interaction between them. See also: system testing; unit testing.

Linear Code Sequence and Jump A sequence of program statements in which the start point is the target line of a control-flow jump or the first line of the program text. An end point is any line which can be reached from the start point by an unbroken linear sequence of code and from which a jump is added to the start and end points.

Logical Ripple Effect The effect of a code change in one part of a system causing defects in other parts of the system and/or necessitating further changes to other parts of the system.

Module	A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor or executive routine; a logically separable part of a program.
New Tests	Test cases, both functional and structural, which are created in order to exercise modified or additional program functionality and code.
NP-completeness	A notion, which is the basis of a theory, for allowing certain classes of problems to be identified for which no polynomial-time algorithm is likely to exist.
Operations Research	A management science which defines: a) the solution of problems relative to the attainment of specified objectives or criteria, b) the identification of alternative solutions, c) the optimisation, or selection, of the best alternative for the stated criterion, and d) the provision of a system perspective in which a tendency to consider the interrelationship of components in their environment, rather than as separate entities, exists.
Path Testing	Testing designed to execute all or selected paths through a program's code. Contrast with: branch

testing; conditional testing; statement testing.

Perfective Maintenance Maintenance activities performed to enhance the functionality of a computer program. Contrast with: adaptive maintenance; corrective maintenance; preventive maintenance.

Preventive Maintenance Maintenance activities performed to improve the performance, maintainability or other attributes of a program. Contrast with: adaptive maintenance; corrective maintenance; perfective maintenance.

Program Slicing A form of program decomposition which is based on the extraction of information from the program's control structure and data structure.

Progressive Regression Testing Regression testing which is applied in the presence of an altered program specification, design and implementation.

Redundant Tests Test cases, both functional and structural, which exercise program functionality and code already exercised by other test cases.

Regression Testing Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

Retestable Tests

Test cases, both functional and structural, which need to be rerun as they exercise those parts of the program functionality and code which are affected by program modifications.

Reusable Tests

Test cases, both functional and structural, which are not affected by program changes and represent the unmodified portions of the program.

Revealing Subdomains

Testing strategy in which test cases are developed based on the intersection of partitions of the program input domain and path set.

Selective Revalidation

The analysis and retesting of those parts of a program which have been directly or indirectly affected by program modifications. See also: regression testing.

Software Development

The period of time that begins with the decision to develop a software product and ends when the software is delivered. This cycle typically includes a requirement specification phase, design phase, implementation phase, test phase and, sometimes, installation and checkout phase.

Software Lifecycle

The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life

	cycle typically includes a requirements specification phase, design phase, implementation phase, test phase and operation and maintenance phase.
Software Maintenance	The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment. See also: adaptive maintenance; corrective maintenance; perfective maintenance; preventive maintenance.
Software Testing	The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. See also: acceptance testing; dynamic testing; functional testing; integration testing; mutation testing; regression testing; structural testing; system testing; unit testing.
Statement Testing	Testing designed to execute each statement of a program. Contrast with: branch testing; conditional testing; path testing.
Static Analysis	A testing activity that occurs without a program being executed. Contrast with: dynamic testing.

Structural Testing

Testing that takes into account the internal mechanism of a system or component. Types include branch testing, conditional testing, path testing, statement testing. Syn: white-box testing. Contrast with: functional testing.

Structure Chart

A diagram that identifies modules, activities or other entities in a program and shows how larger, more general, entities break down into smaller, more specific, entities. Note: The result is not necessarily the same as the one depicted in a call graph. Contrast with: call-graph.

System Testing

Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. See also: integration testing; unit testing.

Test Coverage Criteria

The criteria that define the degree to which a given test case, or set of test cases, address all specified requirements for a given system or component.

Test Drivers

Software modules used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results.

Test Requirements

The requirements specified for a given system or component that must be satisfied by at least one

test case in a test suite.

Test Selection

This concerns the selection of test cases from the test suite in order to revalidate a modified program and ensure its functional and structural consistency. See also: retestable tests; reusable tests.

Test Stubs

A skeletal or special-purpose implementation of a software module, used to develop or test a module that calls, or is otherwise dependent upon it.

Test Suite Classification

This concerns the classification of a test suite into redundant, retestable, reusable and new tests in response to program modifications. See also: redundant tests; retestable tests; reusable tests; new tests.

Test Suite Management

This concerns the selection and maintenance of test cases in a test suite in response to program modifications. See also: test selection; test update.

Test Update

This concerns the updating of test cases in the test suite involving the identification of any redundant test cases and the possible generation of new test cases. See also: redundant tests; new tests.

Testing History

The association that is formed between each test requirement of a program and those test cases which exercise it. See also: test requirements.

Top-Down Testing

Pertains to a testing activity which starts with the highest level component of a software system hierarchy and proceeds through progressively lower levels. Contrast with: bottom-up testing.

Unit Testing

Testing of individual software units or groups of related units. See also: integration testing; system testing.

Validation

The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: verification.

Verification

The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: validation.

Appendix B
Glossary of Notation

a	An achievement vector or function.
A	Constraint matrix.
b	Requirements vector, $b_1, b_2, ..., b_m$.
c	Cost vector, $c_1, c_2, ..., c_n$.
$\mathbf{C}_j$	Denotes the *jth* column of **A**.
$\mathbf{g}_k(\eta,\rho)$	Denotes the *kth* linear function of the deviation variables η and ρ.
$\mathbf{r}_i$	Denotes the *ith* test requirement.
R	Set of test requirements, $r_1, r_2, ..., r_m$.
$\mathbf{R}_i$	Denotes the *ith* row of A
t	Set of test cases, $t_1, t_2, ..., t_n$.
$\mathbf{t}_j$	Denotes the *jth* test case.
T	Set of testing subsets, $T_1, T_2, ..., T_m$.
$\mathbf{T}_i$	Denotes the *ith* testing subset.
x	Set of decision variables, $x_1, x_2, ..., x_n$.
$\underline{\mathbf{x}}$	Incumbent solution.
u	Current partial solution.

Z	Objective function.
η	Negative deviation variable.
ρ	Positive deviation variable.

Bibliography

[1] Abbott, J. *Software Testing Techniques.* NCC Publication, 1986.

[2] Adrion, W.R., Branstad, M.A., and Cherniavsky, J.C. "Validation, Verification, and Testing of Computer Software", *ACM Computing Surveys*, vol. 14, no. 2, pp. 159-92, June, 1982.

[3] Agrawal, H. and Horgan, J.R. "Dynamic Program Slicing". In *Proceedings of ACM SIGPLAN'90 Conference on Programming Languages, Design and Implementation*, ACM Press, New York, pp. 246-56, June, 1990.

[4] Agrawal, H., DeMillo, R.A., and Spafford, E.H. "Dynamic Slicing in the Presence of Unconstrained Pointers". In *Proceedings of the Symposium on Testing, Analysis and Verification (TAV4)*, ACM Press, New York, pp. 60-73, October, 1991.

[5] Allen, F.E. "Interprocedural Data Flow Analysis". In *Proceedings of Information Processing Conference (IFIP'74)*, North-Holland Publishing Company, New York, pp. 398-402, August, 1974.

[6] Allen, F.E. and Cocke, J. "A Program Data Flow Analysis Procedure", *Communications of the ACM*, vol. 19, no. 3, pp. 137-47, March, 1976.

[7] Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. "Conversion of Control Dependence to Data Dependence". In *Proceedings of 10th Annual ACM Symposium on Principles of Programming Languages (POPL)*, ACM Press, New York, pp. 177-89, January, 1983.

[8] Ambras, J.P., Berlin, L.M., Chiarelli, M.L., Foster, A.L., O'Day, V., and Splitter, R.N. "Microscope: An Integrated Program Analysis Toolset", *Hewlett-Packard Journal*, pp. 71-82, August, 1988.

[9] Andreas, J.R. "Automated Regression Testing of Graphical User Interface Based Applications". In *Proceedings of 24th Annual Hawaii Conference on System Sciences(HICSS)*, IEEE Computer Society Press, Los Alamitos, pp. 101, January, 1991.

[10] Archie, K.C. and McLear, R.E. "Environments for Testing Software Systems", *AT&T Technical Journal*, vol. 69, no. 2, pp. 65-75, March/April, 1990.

[11] Arthur, L.J. *Software Evolution-The Software Maintenance Challenge*. John Wiley, Wiley-Interscience, 1988.

[12] Balas, E. "An Additive Algorithm for Solving Linear Programs with Zero-One Variables", *Operations Research*, vol. 13, no. 4, pp. 517-46, July/August, 1965.

[13] Balcer, M., Hasling, W., and Ostrand, T. "Automatic Generation of Test Scripts from Formal Test Specifications". In *Proceedings of ACM SIGSOFT'89 - Third Symposium on Software Testing, Verification, and Analysis (TAVS-3)*, ACM Press, New York, pp. 210-8, December, 1989.

[14] Bar-Yehuda, R. and Even, S. "A Linear-Time Approximation Algorithm for the Weighted Vertex Cover Problem", *Journal of Algorithms*, vol. 2, pp. 198-203, 1981.

[15] Barth, J.M. "A Practical Interprocedural Data Flow Analysis", *Communications of the ACM*, vol. 21, no. 9, pp. 724-36, September, 1978.

[16] Baxter, A.Q. and French, J.A. "Specifications and Testing Aided by a Variant of the Cause-Effect Graph". In *Proceedings of the 30th Annual Southeast Conference*, ed. Pancake, C.M. and Reeves, D.S., ACM Press, New York, pp. 405-8, April, 1992.

[17] Beck, L.L. and Perkins, T.E. "A Survey of Software Engineering Practice : Tools, Methods and Results", *IEEE Transactions on Software Engineering*, vol. SE-9, no. 5, pp. 541-61, September, 1983.

[18] Benedusi, P., Cimitile, A., and DeCarlini, U. "Post-Maintenance Testing Based on Path Change Analysis". In *Proceedings of Conference on Software Maintenance (CSM-88)*, IEEE Computer Society Press, Los Alamitos, pp. 352-61, October, 1988.

[19] Bergeretti, J.F. and Carré, B.A. "Information Flow and Data Flow Analysis of while Programs", *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 37-61, January, 1985.

[20] Bering, C.A. and Covey, J.H. "Software Testing - Concepts and Approach". In *Proceedings of NAECON'91*, IEEE Computer Society Press, Los Alamitos, pp. 750-6, May, 1991.

[21] Berns, G.M. "Analysis Tool Tracks Down Bugs in Fortran Code", *Computer Design*, pp. 169-74, June, 1985.

[22] Blair, R.M. and Uhrich, D. "Regression Testing Tools for Embedded Software Control Systems". In *Proceedings of Eight Pacific Northwest Software Quality Conference*, pp. 167-83, October, 1990.

[23] Brooks, F. *The Mythical Man-Month.* Addison-Wesley, 1975.

[24] Brown, J.M. and Gilg, T.J. "Sharing Testing Responsibilities in the Starbase/X11 Merge System", *Hewlett-Packard Journal*, vol. 40, no. 6, pp. 42-6, December, 1989.

[25] Brown, P. and Hoffman, D. "The Application of Module Regression Testing at TRIUMF", *Nuclear Instruments and Methods in Physics Research*, vol. A293, no. 1/2, pp. 377-81, August, 1990.

[26] Burke, M. and Cytron, R. "Interprocedural Dependence Analysis and Parallelization". In *Proceedings of ACM Sigplan'86 Symposium on Compiler Construction*, ACM Press, New York, pp. 162-75, July, 1986.

[27] Burke, M.G. and Ryder, B.G. "A Critical Analysis of Incremental Iterative Data Flow Analysis Algorithms", *IEEE Transactions on Software Engineering*, vol. SE-16, no. 7, pp. 723-8, July, 1990.

[28] Byrne, J.L. and Proll, L.G. "Initialising Geoffrion's Implicit Enumeration Algorithm for the Zero-One Linear Programming Problem", *The Computer Journal*, vol. 12, no. 4, pp. 381-4, November, 1969.

[29] Cagan, M.R. "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools", *Hewlett-Packard Journal*, vol. 41, no. 3, pp. 36-47, June, 1990.

[30] Callahan, D. "The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis". In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 47-56, June, 1988.

[31] Calliss, F.W. and Cornelius, B.J. "Dynamic Data Flow Analysis of C Programs". In *Proceedings of 21st Annual Hawaii International Conference on System Sciences (HICSS)*, IEEE Computer Society Press, Los Alamitos, pp. 518-23, January, 1988.

[32] Calliss, F.W. "*Inter-module Code Analysis Techniques for Software Maintenance*", PhD. thesis, University of Durham, 1989.

[33] Cantone, G., Cimitile, A., and DeCarlini, U. "Testability and Path Testing Strategies", *Microprocessing and Microprogramming*, vol. 21, pp. 371-82, 1987.

[34] Chang, P.S. "*Some Measures for Software Maintainability*", PhD. thesis, Northwestern University, August, 1987.

[35] Chellappa, M. "Nontraversible Paths in a Program", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 6, pp. 751-6, June, 1987.

[36] Chvátal, V. "A Greedy Heuristic for the Set-Covering Problem", *Mathematical Operations Research*, vol. 4, pp. 233-235, 1979.

[37] Cimitile, A. and DeCarlini, U. "Post-Maintenance Testing: The Effectiveness of the Structural Approach". In *2nd European Software Maintenance Workshop*, Durham, England, September, 1988.

[38] Cimitile, A., DiLucca, G.A., and Maresca, P. "Maintenance and Intermodular Dependencies in Pascal Environment". In *Proceedings of Conference on Software Maintenance(CSM-90)*, IEEE Computer Society Press, Los Alamitos, pp. 72-83, November, 1990.

[39] Clarke, L.A. "A System to Generate Test Data and Symbolically Execute Programs", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215-22, September, 1976.

[40] Clarke, L.A. and Hassel, J. "A Close Look At Domain Testing", *IEEE Transactions on Software Engineering*, vol. 8, no. 4, July, 1982.

[41] Clarke, L.A., Podgurski, A., Richardson, D.J., and Zeil, S.J. "A Comparison of Data Flow Path Selection Criteria". In *Proceedings of International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, Los Alamitos, pp. 244-51, August, 1985.

[42] Clarkson, K.L. "A Modification of the Greedy Algorithm for Vertex Cover", *Information Processing Letters*, vol. 16, no. 2, pp. 23-5, January, 1983.

[43] Collofello, J.S. and Klinkel, G.D. "An Automated Pascal Test Coverage Assessment Tool". In *Proceedings of COMPSAC'82*, IEEE Computer Society Press, Los Alamitos, pp. 626-33, November, 1982.

[44] Collofello, J.S. and Ferrara, A.F. "An Automated Pascal Multiple Condition Test Coverage Tool". In *Proceedings of COMPSAC'84*, IEEE Computer Society Press, Los Alamitos, pp. 20-3, November, 1984.

[45] Collofello, J.S. and Bortman, S. "An Analysis of the Technical Information Necessary to Perform Effective Software Maintenance". In *Proceedings of 5th Annual Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, Los Alamitos, pp. 420-4, March, 1986.

[46] Collofello, J.S. and Buck, J.J. "Software Quality Assurance for Maintenance", *IEEE Software*, vol. 4, no. 5, pp. 46-51, September, 1987.

[47] Collofello, J.S. and Cousins, L. "Towards Automatic Software Fault Location Through Decision-to-Decision Path Analysis". In *Proceedings of AFIPS Conference (NCC)*, pp. 539-44, 1987.

[48] Connell, C. "DEC's VAXset Tools", *DEC Professional*, pp. 72-7, March, 1987.

[49] Cooper, K.D. and Kennedy, K. "Efficient Computation of Flow Insensitive Interprocedural Summary Information". In *Proceedings of ACM SIGPLAN'84 Symposium on Compiler Construction (SIGPLAN Notices)*, ACM Press, New York, pp. 247-58, June, 1984.

[50] Cooper, K.D., Kennedy, K., and Torczon, L. "Interprocedural Optimisation: Eliminating Unnecessary Recompilation". In *Proceedings of ACM Sigplan'86 Symposium on Compiler Construction*, ACM Press, New York, pp. 58-67, July, 1986.

[51] Cooper, S.D. and Munro, M. "Software Change Information for Maintenance Management". In *Proceedings of Conference on Software Maintenance(CSM-89)*, IEEE Computer Society Press, Los Alamitos, pp. 279-87, October, 1989.

[52] Coward, P.D. "A Review of Software Testing", *Information and Software Technology*, vol. 30, no. 3, pp. 189-98, April, 1988.

[53] Coward, P.D. "Determining Path Feasibility For Commercial Programs", *ACM Sigplan Notices*, vol. 23, no. 3, pp. 93-101, March, 1988.

[54] Daly, E.B. "Management of Software Development", *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 229-42, May, 1977.

[55] Deb, R.K. "On Generation of Test Data and Minimal Cover of Directed Graphs". In *Proceedings of IFIP Congress'77*, pp. 13-6, 1977.

[56] DeMillo, R.A. "Test Adequacy and Program Mutation", Technical Report SERC-TR-37-P, Software Engineering Research Center(SERC), Purdue University, February, 1989.

[57] Denning, P.J. "What is Software Quality?", *Communications of the ACM*, vol. 35, no. 1, pp. 13-5, January, 1992.

[58] Dijkstra, E.W. "Notes on Structured Programming". In *Structured Programming*, Academic Press, ed. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., pp. 6, 1972.

[59] Ebert, J. "A Sensitive Transitive Closure Algorithm", *Information Processing Letters*, vol. 12, no. 5, pp. 255-8, October, 1981.

[60] Elmendorf, W.R. "Functional Analysis using Cause-Effect Graphs". In *Proceedings of SHARE XLIII*, SHARE, New York, pp. 567-77, 1974.

[61] Etcheberry, J. "The Set-Covering Problem: A New Implicit Enumeration Algorithm", *Operations Research*, vol. 25, no. 5, pp. 760-72, September/October, 1977.

[62] Ferrante, J. "A Program Form Based on Data Dependency in Predicate Regions". In *Proceedings of 10th Annual ACM Symposium on Principles of Programming Languages (POPL)*, ACM Press, New York, pp. 217-31, January, 1983.

[63] Ferrante, J., Ottenstein, K.J., and Warren, J.D. "The Program Dependence Graph and Its Use in Optimisation", *ACM Transactions on Programming Languages and Systems (POPLAS)*, vol. 9, no. 3, pp. 319-49, July, 1987.

[64] Fiala, F. "Computational Experience with a Modification of an Algorithm by Hammer and Rudeanu for 0-1 Linear Programming". In *Proceedings of ACM National Conference*, ACM Press, New York, pp. 482-8, 1971.

[65] Fischer, K.F. "A Test Case Selection Method for the Validation of Software Maintenance Modifications". In *Proceedings of COMPSAC'77*, IEEE Computer Society Press, Los Alamitos, pp. 421-6, November, 1977.

[66] Fischer, K.F. "*A Graph Theoretic Approach to the Validation of Software Maintenance Modifications*", PhD. thesis, University of California, Los Angeles, 1980.

[67] Fischer, K.F., Raji, F., and Chruscicki, A. "A Methodology for Re-Testing Modified Software". In *Proceedings of National Telecommunications Conference*, IEEE Computer Society Press, Los Alamitos, pp. B6.3.1-6, November, 1981.

[68] Fischer, K.F., Raji, F., and Onaszko, D. "Software Retest Techniques", Technical Report RADC-TR-82-275, Rome Air Development Center, October, 1982.

[69] Fleischmann, B. "Computational Experience with the Algorithm of Balas", *Operations Research*, vol. 15, no. 1, pp. 153-5, 1967.

[70] Frankl, P.G., Weiss, S.N., and Weyuker, E.J. "ASSET: A System to Select and Evaluate Tests". In *Proceedings of International Conference on Software Tools*, IEEE Computer Society Press, Los Alamitos, pp. 72-9, April, 1985.

[71] Frankl, P.G. and Weyuker, E.J. "A Data Flow Testing Tool". In *Proceedings of SOFTFAIRII - 2nd Conference on Software Development, Tools, Techniques and Alternatives*, IEEE Computer Society Press, Los Alamitos, pp. 46-53, 1985.

[72] Frankl, P.G. and Weyuker, E.J. "An Applicable Family of Data Flow Testing Criteria", *IEEE Transactions on Software Engineering*, vol. SE-14, no. 10, pp. 1483-98, October, 1988.

[73] Fuget, C.D. and Scott, B.J. "Tools for Automating Software Test Package Execution", *Hewlett-Packard Journal*, vol. 37, no. 3, pp. 24-8, March, 1986.

[74] Furukawa, Z., Nogi, K., and Tokunaga, K. "AGENT: An Advanced Test-Case Generation System for Functional Testing". In *Proceedings of AFIPS Conference (NCC)*, pp. 525-35, 1985.

[75] Gallagher, K.B. and Lyle, J.R. "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering*, vol. SE-17, no. 8, pp. 751-61, August, 1991.

[76] Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[77] Garfinkel, R.S. and Nemhauser, G.L. *Integer Programming*. John Wiley, Interscience, 1972.

[78] Geoffrion, A.M. "Integer Programming by Implicit Enumeration and Balas' Method", *SIAM Review*, vol. 9, no. 2, pp. 178-90, April, 1967.

[79] Geoffrion, A.M. "An Improved Implicit Enumeration Approach for Integer Programming", *Operations Research*, vol. 17, pp. 437-54, 1969.

[80] Glover, F. "A Multiphase-Dual Algorithm for the Zero-One Integer Programming Problem", *Operations Research*, vol. 13, no. 4, pp. 879-919, November/December, 1965.

[81] Gomory, R. "All-Integer Integer Programming", Technical Report RC-189, IBM Research Center, 1960.

[82] Gondram, M. and Minoux, M. *Graphs and Algorithms*. John Wiley, 1984.

[83] Goodenough, J.B. and Gerhart, S.L. "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 156-73, June, 1975.

[84] Graham, D.R. "Software Testing Tools: A New Classification Scheme", *Journal of Software Testing, Verification and Reliability*, vol. 1, no. 3, pp. 17-34, October/December, 1991.

[85] Gue, R.L., Liggett, J.C., and Cain, K.C. "Analysis of Algorithms for the Zero-One Programming Problems", *Communications of the ACM*, vol. 11, no. 12, pp. 837-44, December, 1968.

[86] Henry, S.H. and Wake, S. "Predicting Maintainability with Software Quality Metrics", *Journal of Software Maintenance : Research and Practice*, vol. 3, no. 3, pp. 129-43, September, 1991.

[87] Haley, A. and Zweben, S. "Development and Application of a White Box Approach to Integration Testing", *Journal of Systems and Software*, vol. 4, no. 4, pp. 309-15, November, 1984.

[88] Halstead, M.H. *Elements of Software Science*. North Holland, Amsterdam, 1977.

[89] Hamlet, R.G. "Testing Programs with the Aid of a Compiler", *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 279-90, July, 1977.

[90] Harrold, M.J. and Soffa, M.L. "An Incremental Approach to Unit Testing During Maintenance". In *Proceedings of Conference on Software Maintenance (CSM-88)*, IEEE Computer Society Press, Los Alamitos, pp. 362-67, October, 1988.

[91] Harrold, M.J. and Soffa, M.L. "Interprocedural Data Flow Testing". In *Proceedings of ACM SIGSOFT'89 - Third Symposium on Software Testing, Verification, and Analysis (TAVS-3)*, ACM Press, New York, pp. 158-67, December, 1989.

[92] Harrold, M.J. and Soffa, M.L. "An Incremental Data Flow Testing Tool". In *Proceedings of 6th International Conference on Testing Computer Software*, Washington D.C., June, 1989.

[93] Harrold, M.J. and Soffa, M.L. "Efficient Computation of Interprocedural Data Dependencies". In *Proceedings of International Conference on Computer Languages*, IEEE Computer Society Press, Los Alamitos, pp. 297-306, March, 1990.

[94] Harrold, M.J., Gupta, R., and Soffa, M.L. "TBM: A Testbed Management Tool". In *Proceedings of 7th International Conference on Testing Computer Software*, San Francisco, pp. 47-55, June, 1990.

[95] Harrold, M.J., Gupta, R., and Soffa, M.L. "A Methodology for Controlling the Size of a Test Suite". In *Proceedings of Conference on Software Maintenance (CSM-90)*, IEEE Computer Society Press, Los Alamitos, pp. 302-10, November, 1990.

[96] Harrold, M.J. and Soffa, M.L. "Selecting and using Data for Integration Testing", *IEEE Software*, vol. 8, no. 2, pp. 58-65, March, 1991.

[97] Harrold, M.J. and Malloy, B. "A Unified Interprocedural Program Representation for a Maintenance Environment". In *Proceedings of Conference on Software Maintenance (CSM-91)*, IEEE Computer Society Press, Los Alamitos, pp. 138-47, October, 1991.

[98] Hartmann, J. and Robson, D.J. "Approaches to Regression Testing". In *Proceedings of Conference on Software Maintenance (CSM-88)*, IEEE Computer Society Press, Los Alamitos, pp. 368-72, October, 1988.

[99] Hartmann, J. and Robson, D.J. "Revalidation During the Software Maintenance Phase". In *Proceedings of Conference on Software Maintenance(CSM-89)*, IEEE Computer Society, Los Alamitos, pp. 70-80, October, 1989.

[100] Hartmann, J. "Development of Testing Methodologies and Tools for Use in Software Maintenance". In *Proceedings of Third International Workshop on Computer-Aided Software Engineering (CASE'89)*, pp. 33-4, July, 1989.

[101] Hartmann, J. and Robson, D.J. "Techniques for Selective Revalidation", *IEEE Software*, vol. 7, no. 1, pp. 31-6, January, 1990.

[102] Hartmann, J. and Robson, D.J. "RETEST - Development of a Selective Revalidation Prototype Environment for Use in Software Maintenance". In *Proceedings of 23rd Annual Hawaii International Conference on System Sciences (HICSS-23)*, IEEE Computer Society Press, Los Alamitos, pp. 93-101, January, 1990.

[103] Hartmann, J. and Robson, D.J. "A Systematic Approach to Regression Testing". In *Proceedings of Ninth Annual Pacific Northwest Software Quality Conference*, pp. 309-23, October, 1991.

[104] Hedley, D. and Hennell, M.A. "The Causes and Effects of Infeasible Paths in Computer Programs". In *Proceedings of International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, Los Alamitos, pp. 259-66, August, 1985.

[105] Hennell, M.A. and Prudom, J.A. "A Static Analysis of the NAG Library", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 4, pp. 329-33, 1980.

[106] Hennell, M.A., Hedley, D., and Riddell, I.J. "Assessing a Class of Software Tools". In *Proceedings of International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, Los Alamitos, pp. 266-77, March, 1984.

[107] Herington, D.E., Nichols, P.A., and Lipp, R.D. "Software Verification using Branch Analysis", *Hewlett-Packard Journal*, vol. 38, no. 6, pp. 13-23, June, 1987.

[108] Herman, P.M. "A Data Flow Analysis Approach to Program Testing", *The Australian Computer Journal*, vol. 8, no. 3, pp. 92-6, November, 1976.

[109] Hetzel, W.C. *The Complete Guide to Software Testing*. QED Information Sciences, Inc., 2nd Edition, 1988.

[110] Hillier, F.S. and Lieberman, G.J. *Introduction to Operations Research*. Holden-Day, 4th Edition, 1986.

[111] Hochbaum, D.S. "Approximation Algorithms for the Set Covering and Vertex Cover Problems", *SIAM Journal of Computing*, vol. 11, no. 3, pp. 555-6, August, 1982.

[112] Holzman, A.G. "Linear Programming". In *Mathematical Programming for Operations Researchers and Computer Scientists*, Marcel Dekker, ed. Holzman, A.G., chapt. 1, pp. 1-40, 1981.

[113] Horgan, J.R. and London, S. "Data Flow Coverage and the C Language". In *Proceedings of the Symposium on Testing, Analysis and Verification (TAV4)*, ACM Press, New York, pp. 87-97, October, 1991.

[114] Horgan, J.R. and London, S. "A Data Flow Coverage Testing Tool for C". In *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, IEEE Computer Society, Los Alamitos, pp. 2-10, May, 1992.

[115] Horwitz, S., Reps, T., and Binkley, D. "Interprocedural Slicing Using Dependence Graphs", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26-60, January, 1990.

[116] Howden, W.E. "Methodology for the Generation of Program Test Data", *IEEE Transactions on Computers*, vol. C-24, no. 5, pp. 554-9, May, 1975.

[117] Howden, W.E. "Reliability of the Path Testing Strategy", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 208-15, September, 1976.

[118] Howden, W.E. "Symbolic Testing And The DISSECT Symbolic Evaluator", *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, April, 1977.

[119] Howden, W.E. "Functional Program Testing", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 162-9, March, 1980.

[120] Howden, W.E. *Functional Program Testing and Analysis*. McGraw-Hill, Computer Science, 1987.

[121] Howley, P.P. "An Assessment of Software Testing Techniques for Maintenance". In *Proceedings of Software Maintenance Workshop*, IEEE Computer Society Press, Los Alamitos, pp. 261-6, December, 1983.

[122] Huang, J.C. "An Approach to Program Testing", *ACM Computing Surveys*, vol. 7, no. 3, pp. 113-28, September, 1975.

[123] Huang, J.C. "Detection of Data Flow Anomaly Through Program Instrumentation", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 226-36, May, 1979.

[124] Ignizio, J.P. *Linear Programming in Single- and Multiple-Objective Systems*. Prentice-Hall Inc., 1982.

[125] Ince, D. and Hekmatpour, S. "The Evaluation of Some Black-Box Testing Methods", Technical Report 84/7, Open University, December, 1984.

[126] Ince, D.C. and Hekmatpour, S. "An Empirical Evaluation of Random Testing", *The Computer Journal*, vol. 29, no. 4, pp. 380, August, 1986.

[127] Jachner, J. and Agrawal, V.K. "Data Flow Anomaly Detection", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 432-7, July, 1984.

[128] Jackson, M.A. *Principles of Program Design.* Academic Press, London, 1975.

[129] Jensen, K. and Wirth, N. *PASCAL: User Manual and Report.* Springer-Verlag, 3rd Edition, 1985.

[130] Jiang, J., Zhou, X., and Robson, D.J. "Program Slicing for C - The Problems In Implementation". In *Proceedings of Conference on Software Maintenance (CSM-91)*, IEEE Computer Society Press, Los Alamitos, pp. 182-90, October, 1991.

[131] Johmann, K., Liu, S.S., and Yau, S. "Dataflow Equations for Context-Dependent Flow-Sensitive Interprocedural Analysis", Technical Report SERC-TR-45-F, Software Engineering Research Center(SERC), University of Florida, January, 1991.

[132] Johnson, D.S. "Approximation Algorithms for Combinatorial Problems", *Journal of Computer and System Sciences*, vol. 9, pp. 256-78, 1974.

[133] Johnson, M.A. "Automated Testing of User Interfaces". In *Proceedings of the 5th Pacific Northwest Software Quality Conference*, pp. 285-93, 1987.

[134] Kaiser, G.E., Perry, D.E., and Schell, W.M. "INFUSE: Fusing Integration Test Management with Change Management". In *Proceedings of COMPSAC'89*, IEEE Computer Society, Los Alamitos, pp. 552-8, September, 1989.

[135] Kao, H. and Chen, T.Y. "Data Flow Analysis For COBOL", *ACM Sigplan Notices*, vol. 19, no. 7, pp. 18-21, July, 1984.

[136] Kennedy, K. "A Survey of Data Flow Analysis Techniques". In *Program Flow Analysis: Theory and Applications*, Prentice-Hall Inc., ed. Muchnick, S.S. and Jones, N.D., chapt. 1, pp. 5-54, 1981.

[137] Kernigan, B.W. and Ritchie, D. *The C Programming Language.* Prentice-Hall, 1st Edition, 1978.

[138] King, J.C. "Symbolic Execution and Program Testing", *Communications of the ACM*, vol. 19, no. 7, pp. 385-94, July, 1976.

[139] King, K.N. and Offutt, A.J. "A Fortran Language System for Mutation-based Software Testing", *Software - Practice and Experience*, vol. 21, no. 7, pp. 685-718, July, 1991.

[140] Kojo, T., Kobayashi, H., and Kitamura, A. "A Software Testing System for Digital Switching Systems". In *Proceedings of International Communications Conference*, IEEE Computer Society Press, Los Alamitos, pp. 1016-9, May, 1984.

[141] Korel, B. and Laski, J. "A Tool for Data Flow Oriented Program Testing". In *Proceedings of SOFTFAIRII - 2nd Conference on Software Development Tools, Techniques and Alternatives*, IEEE Computer Society Press, Los Alamitos, pp. 34-7, December, 1985.

[142] Korel, B. and Laski, J. "Dynamic Program Slicing", *Information Processing Letters*, vol. 29, no. 3, pp. 155-63, October, 1988.

[143] Korel, B. and Laski, J. "Dynamic Slicing of Computer Programs", *Journal of Systems and Software*, vol. 13, no. 3, pp. 187-95, November, 1990.

[144] Korel, B. "A Dynamic Approach to Test Data Generation". In *Proceedings of Conference on Software Maintenance(CSM-90)*, IEEE Computer Society Press, Los Alamitos, pp. 311-7, November, 1990.

[145] Krause, K.W. and Smith, R.W. "Optimal Software Test Planning Through Automated Network Analysis". In *Proceedings of Symposium on Computer Software Reliability*, IEEE Computer Society Press, Los Alamitos, pp. 18-22, April, 1973.

[146] Landi, W. and Ryder, B.G. "Pointer-induced Aliasing: A Problem Classification". In *Proceedings of 18th Annual ACM Symposium on Principles of Programming Languages (POPL)*, ACM Press, New York, pp. 93-103, January, 1991.

[147] Laski, J. "On Data Flow Guided Program Testing", *ACM Sigplan Notices*, vol. 17, no. 9, pp. 62-71, September, 1982.

[148] Laski, J.W. and Korel, B. "A Data Flow Oriented Program Testing Strategy", *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 347-54, May, 1983.

[149] Laski, J. "Data Flow Testing in STAD", *Journal of Systems and Software* , vol. 12, no. 1, pp. 3-14, April, 1990.

[150] Leach, D.M., Paige, M.R., and Satko, J.E. "AUTOTESTER: A Testing Methodology for Interactive User Environments". In *Proceedings of 2nd Annual Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, Los Alamitos, pp. 143-7, March, 1983.

[151] Lee, J.A.N. and He, X. "A Methodology for Test Selection", *Journal of Systems and Software*, vol. 13, no. 3, pp. 177-85, November, 1990.

[152] Lee, J., Wang, K.H., and Chou, C.R. "An Implementation of Software Tools for Replay and Partial Replay of Concurrent-C Programs". In *Proceedings of COMPSAC'90*, IEEE Computer Society Press, Los Alamitos, pp. 106-11, October, 1990.

[153] Lemke, C.E., Salkin, H.M., and Spielberg, K. "Set Covering by Single-Branch Enumeration with Linear-Programming Subproblems", *Operations Research*, vol. 19, no. 4, pp. 998-1022, July/August, 1971.

[154] Leung, H.K.N. and White, L. "A Study of Regression Testing", Technical Report TR 88-15, University of Alberta, September, 1988.

[155] Leung, H.K.N. and White, L.J. "Insights into Regression Testing". In *Proceedings of Conference on Software Maintenance(CSM-89)*, IEEE Computer Society Press, Los Alamitos, pp. 60-9, October, 1989.

[156] Leung, H.K.N. and White, L.J. "A Study of Regression Testing". In *Proceedings of 6th International Conference on Testing Computer Software*, Washington D.C., June, 1989.

[157] Leung, H.K.N. and White, L.J. "Regression Testing at the Integration Testing Level". In *Proceedings of 7th International Conference on Testing Computer Software*, pp. 225-35, June, 1990.

[158] Leung, H.K.N. and White, L.J. "A Study of Integration Testing and Software Regression at the Integration Level". In *Proceedings of Conference on Software Maintenance (CSM-90)*, IEEE Computer Society Press, Los Alamitos, pp. 290-301, November, 1990.

[159] Leung, H.K.N. and White, L. "A Cost Model to Compare Regression Testing Strategies". In *Proceedings of Conference on Software Maintenance (CSM-91)*, IEEE Computer Society Press, Los Alamitos, pp. 201-8, October, 1991.

[160] Leung, H.K.N. "*A Framework for Regression Testing*", PhD. thesis, University of Alberta, May, 1992.

[161] Lewis, J. and Henry, S. "A Methodology for Integrating Maintainability using Software Metrics". In *Proceedings of Conference on Software Maintenance(CSM-89)*, IEEE Computer Society Press, Los Alamitos, pp. 32-9, October, 1989.

[162] Lewis, R., Beck, D.W., and Hartmann, J. "Assay - A Tool to Support Regression Testing". In *Proceedings of 2nd European Software Engineering Conference (ESEC'89)*, pp. 487-96, September, 1989.

[163] Lientz, B.P., Swanson, E.B., and Tompkins, G.E. "Characteristics of Application Software Maintenance", *Communications of the ACM*, vol. 21, no. 6, pp. 466-71, June, 1978.

[164] Lin, J.C. and Chung, C.G. "Zero-One Integer Programming Model in Path Selection Problem of Structural Testing". In *Proceedings of COMPSAC'89*, IEEE Computer Society, Los Alamitos, pp. 618-27, September, 1989.

[165] Liu, S.S. and Ogando, R. "An Emacs-Based Logical Ripple Effect Analyzer Prototype User's Manual", Technical Report TR-32-F, Software Engineering Research Center(SERC), University of Florida, September, 1989.

[166] Liu, L. and Robson, D.J. "SEMST - A Support Environment for the Management of Software Testing". In *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, IEEE Computer Society, Los Alamitos, pp. 11-20, May, 1992.

[167] Livadas, P.E. "The C-Ghinsu Tool", Technical Report SERC-TR-49-F, Software Engineering Research Center(SERC), University of Florida, March, 1991.

[168] Lu, Q. and Qian, J. "Incremental Flow Analysis and its Applications in Incremental Programming Environment". In *Proceedings of COMPSAC'87*, IEEE Computer Society Press, Los Alamitos, pp. 89-95, October, 1987.

[169] Luss, H. "Optimization: Methodology, Algorithms, and Applications", *AT&T Technical Journal*, vol. 68, no. 3, pp. 3-6, May/June, 1989.

[170] Lutz, M. "Testing Tools", *IEEE Software*, vol. 7, no. 3, pp. 53-7, May, 1990.

[171] Lyle, J.R. and Gallagher, K.B. "Using Program Decomposition to Guide Modifications". In *Proceedings of Conference on Software Maintenance(CSM-88)*, IEEE Computer Society Press, Los Alamitos, pp. 265-9, October, 1988.

[172] Müller-Merbach, H. "Modelling Techniques and Heuristics for Combinatorial Problems". In *Combinatorial Programming: Methods and Applications*, Reidel Publishing, ed. Roy, B., pp. 3-27, 1974.

[173] Mahendrarajah, A. and Fiala, F. "A Comparison of Three Algorithms for Linear Zero-One Programs", *ACM Transactions on Mathematical Software (TOMS)*, vol. 2, no. 4, pp. 331-4, December, 1976.

[174] Manber, U. *Introduction to Algorithms - A Creative Approach*. Addison-Wesley, 1989.

[175] McCabe, T.J. *Tutorial: Structured Testing*. IEEE Computer Society Press, Los Alamitos, October , 1982.

[176] McCabe, T.J. "Structured Testing: A Testing Methodology using the McCabe Complexity Metric". In *Tutorial: Structured Testing*, IEEE Press, chapt. Section II, pp. 19-47, Los Alamitos, 1983.

[177] McCabe, T.J. and Schulmeyer, G.G. "System Testing Aided by Structured Analysis: A Practical Experience", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 9, pp. 917-21, September, 1985.

[178] Meekel, J. and Viala, M. "LOGISCOPE: A Tool for Maintenance". In *Proceedings of Conference on Software Maintenance(CSM-88)*, IEEE Computer Society Press, Los Alamitos, pp. 328-34, October, 1988.

[179] Melhorn, K. *Graph Algorithms and NP-Completeness: Data Structures and Algorithms*. Springer Verlag, 2, 1984.

[180] Meredith, D.C. "Regression Testing As Preventive Medicine", *System Builder*, pp. 42-7, October/November, 1989.

[181] Miller, E.F., Bardens, J.A., Benson, J.P., Melton, R.A., Urban, R.J., and Wisehart, W.R. "Structurally Based Automatic Program Testing". In *Proceedings of Eascon'74*, IEEE Computer Society Press, Los Alamitos, pp. 134-9, 1974.

[182] Miller, E. "Advanced Methods in Automated Software Test". In *Proceedings of Conference on Software Maintenance(CSM-90)*, IEEE Computer Society Press, Los Alamitos, pp. 111, November, 1990.

[183] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[184] Minoux, M. *Mathematical Programming - Theory and Algorithms*. John Wiley, chapt. 7, pp. 245-59, 1986.

[185] Mullin, F.J. "Considerations for a Successful Software Test Program", TRW-SS-77-01, TRW Systems Engineering and Integration Division, Redondo Beach, CA, January, 1977.

[186] Munro, I. "Efficient Determination of the Transitive Closure of a Directed Graph", *Information Processing Letters*, vol. 1, pp. 56-8, 1971.

[187] Myers, G.J. *Software Reliability: Principles and Practices*. John Wiley, New York, 1976.

[188] Myers, G.J. *The Art of Software Testing*. John Wiley, New York, Interscience, 1979.

[189] Nanja, S. and Samadzadeh, M. "A Slicing/Dicing-Based Debugger for C". In *Proceedings of Eight Pacific Northwest Software Quality Conference*, pp. 204-12, October, 1990.

[190] Narula, S.C. and Kindorf, J.R. "Linear 0-1 Programming: A Comparison of Implicit Enumeration Algorithms", *Computing and Operations Research*, vol. 6, pp. 47-51, 1979.

[191] Nejmeh, B.A. "NPATH: A Measure of Execution Path Complexity and Its Application", *Communications of the ACM*, vol. 31, no. 2, pp. 188-200, February, 1988.

[192] Ntafos, S.C. and Hakimi, S.L. "On Path Cover Problems in Digraphs and Applications to Program Testing", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 520-9, September, 1979.

[193] Ntafos, S.C. "On Required Element Testing", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 795-803, November, 1984.

[194] Ntafos, S.C. "A Comparison of Some Structural Testing Strategies", *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 868-74, June, 1988.

[195] Offut, A.J. "Using Mutation Analysis to Test Software". In *Proceedings of 7th International Conference on Testing Computer Software*, San Francisco, pp. 75-7, June, 1990.

[196] Oman, P. "Maintenance Tools", *IEEE Software*, vol. 7, no. 3, pp. 59-64, May, 1990.

[197] Omar, A.A. and Mohammed, F.A. "Structural Testing of Programs : A Survey", *ACM Sigsoft Software Engineering Notes*, vol. 14, no. 2, pp. 62-70, April, 1989.

[198] Omar, A.A. and Mohammed, F.A. "A Survey of Software Functional Testing Methods", *ACM Sigsoft Software Engineering Notes*, vol. 16, no. 2, pp. 75-82, April, 1991.

[199] Osterweil, L.J., Fosdick, L.D., and Taylor, R.N. "Error and Anomaly Diagnosis Through Data Flow Analysis". In *Computer Program Testing*, Elsevier North-Holland, ed. Chandrasekaran, B. and Radicchi, S., pp. 35-63, 1981.

[200] Ostrand, T. and Balcer, M.J. "The Category-Partition Method for Specifying and Generating Functional Tests", *Communications of the ACM, Special Section on Software Testing*, vol. 31, no. 6, pp. 676-86, June, 1988.

[201] Ostrand, T.J. and Weyuker, E.J. "Using Data Flow Analysis for Regression Testing". In *Proceedings of 6th Annual Pacific NorthWest Quality Assurance Conference*, pp. 233-48, September, 1988.

[202] Ostrand, T. and Weyuker, E.J. "Data Flow-Based Test Adequacy Analysis for Languages with Pointers". In *Proceedings of the Symposium on Testing, Analysis and Verification (TAV4)*, ACM Press, New York, pp. 74-86, October, 1991.

[203] Ottenstein, K.J. and Ottenstein, L.M. "The Program Dependence Graph in a Software Development Environment". In *Proceedings of ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments*, ACM Press, New York, pp. 177-84, April, 1984.

[204] Pande, H., Ryder, B.G., and Landi, W. "Interprocedural Def-Use Associations in C Programs". In *Proceedings of the Symposium on Testing, Analysis and Verification (TAV4)*, ACM Press, New York, pp. 139-53, October, 1991.

[205] Panzl, D.J. "Automatic Software Test Drivers", *IEEE Computer*, vol. 11, no. 4, pp. 44-50, April, 1978.

[206] Panzl, D.J. "A Language for Specifying Software Tests". In *Proceedings of AFIPS Conference (NCC)*, pp. 609-19, 1978.

[207] Peleato, J.M., Mills, F.W., and Cheng, S.K. "Automation of Regression Testing for Packet Switches". In *Proceedings of Globecom'87*, IEEE Computer Society Press, Los Alamitos, pp. 1638-41, 1987.

[208] Petersen, C.C. "Computational Experience with Variants of the Balas Algorithm Applied to the Selection of R&D Projects", *Management Science*, vol. 13, no. 9, pp. 736-50, May, 1967.

[209] Platoff, M., Wagner, M., and Camaratta, J. "An Integrated Program Representation and Toolkit for the Maintenance of C Programs". In *Proceedings of Conference on Software Maintenance (CSM-91)*, IEEE Computer Society Press, Los Alamitos, pp. 129-37, October, 1991.

[210] Podgurski, A. and Clarke, L.A. "The Implications of Program Dependences for Software Testing, Debugging and Maintenance". In *Proceedings of ACM SIGSOFT'89 - Third Symposium on Software Testing, Verification, and Analysis (TAVS-3)*, ACM Press, New York, pp. 168-78, December, 1989.

[211] Popkin, G.S. and Shooman, M.L. "On the Number of Tests Necessary to Verify a Computer Program", Technical Report RADC TR-78-229, Rome Air Development Center, November, 1978.

[212] Popkin, G.S. "*A Binary Programming Solution to a Problem in Computer Program Testing*", PhD. thesis, Polytechnic Institute of New York, Brooklyn, NY, January, 1987.

[213] Powell, P.B. "Software Validation, Verification and Testing Technique and Tool Reference Guide", Technical Report NIST-SPEC-PUB 500-93, NIST, September, 1982.

[214] Pressman, R.S. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 1st Edition, 1987.

[215] Purdom, P. "A Transitive Closure Algorithm", *BIT*, vol. 10, pp. 76-94, 1970.

[216] Radatz, J.W. "Analysis of IV&V Data", Technical Report RADC-TR-81-145, Rome Air Development Center, June, 1981.

[217] Rajlich, V. "VIFOR: A Tool for Software Maintenance", *Software Practice and Experience*, vol. 20, no. 1, pp. 67-77, January, 1990.

[218] Rapps, S. and Weyuker, E.J. "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367-75, April, 1985.

[219] Richardson, D.J. and Clarke, L.A. "A Partition Analysis Method to Increase Program Reliability". In *Proceedings of International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, Los Alamitos, pp. 244-53, March, 1981.

[220] Richardson, D.J. and Clarke, L.A. "Partition Analysis: A Method Combining Testing and Verification", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1477-90, December, 1985.

[221] Riechert, G.E. and Smith, S.S. "Automatic Regression Test System". In *Proceedings of International Switching Symposium*, IEEE Computer Society Press, Los Alamitos, pp. B9.1.1-5, March, 1987.

[222] Roe, R.P. and Rowland, J.H. "Some Theory Concerning Certification of Mathematical Subroutines by Black-Box Testing", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 6, pp. 677-82, June, 1987.

[223] Roper, M. and Smith, P. "A Structural Testing Method for JSP Designed Programs", *Journal of Software - Practice and Experience*, vol. 17, no. 2, pp. 135-57, February, 1987.

[224] Ryder, B.G. "Incremental Data Flow Analysis". In *Proceedings of 10th Annual ACM Symposium on Principles of Programming Languages (POPL)*, ACM Press, New York, pp. 167-76, January, 1983.

[225] Ryder, B.G. and Carroll, M.D. "An Incremental Algorithm for Software Analysis". In *ACM Sigplan/Sigsoft Symposium on Practical Software Environments*, pp. 171-9, December, 1986.

[226] Ryder, B.G., Landi, W., and Pande, H.D. "Profiling an Incremental Data Flow Analysis Algorithm", *IEEE Transactions on Software Engineering*, vol. SE-16, no. 3, pp. 129-40, March, 1990.

[227] Salkin, H.M. and Koncal, R.D. "Set Covering by an All Integer Algorithm: Computational Experience", *Journal of the ACM*, vol. 20, no. 2, pp. 189-93, April, 1973.

[228] Scherr, A.L. "Developing and Testing a Large Programming System". In *Program Test Methods*, Prentice-Hall, ed. Hetzel, W.C., chapt. 14, pp. 168-80, 1973.

[229] Schneidewind, N.F. "The State of Software Maintenance", *IEEE Transactions of Software Engineering*, vol. SE-13, no. 3, pp. 303-10, March, 1987.

[230] Shumskas, A.F. "Certification of Production-Representative/Production Software-Intensive Systems for Dedicated Test and Evaluation", *IEEE Aerospace and Electronic Systems Magazine*, vol. 6, no. 9, pp. 9-14, September, 1991.

[231] Solis, D.M. "AutoParts - A Tool to Aid in Equivalence Partition Testing". In *Proceedings of SOFTFAIR II - 2nd Conference on Software Development Tools, Techniques and Alternatives*, IEEE Computer Society Press, Los Alamitos, pp. 122-5, December, 1985.

[232] Stuebing, H.G. "A Modern Facility for Software Production and Maintenance". In *Proceedings of COMPSAC'80*, IEEE Computer Society Press, Los Alamitos, pp. 407-18, 1980.

[233] Su, J. "Testing Motif". In *Proceedings of the 7th Pacific Northwest Software Quality Conference*, pp. 361-79, 1989.

[234] Su, J. and Ritter, P.R. "Experience in Testing the Motif Interface", *IEEE Software*, vol. 8, no. 2, pp. 26-33, March, 1991.

[235] Syslo, M.M., Deo, N., and Kowalik, J.S. *Discrete Optimization Algorithms Using Pascal Programming*. Prentice-Hall, chapt. 1, pp. 1-116, 1983.

[236] Taha, H.A. "Integer Programming". In *Mathematical Programming for Operations Researchers and Computer Scientists*, Marcel Dekker, ed. Holzman, A.G., chapt. 2, pp. 41-69, 1981.

[237] Taha, H.A. *Operations Research - An Introduction*. Macmillan Publishing, 3rd Edition, 1982.

[238] Taha, A.B., Thebaut, S.M., and Liu, S.S. "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis". In *Proceedings of COMPSAC'89*, IEEE Computer Society, Los Alamitos, pp. 527-34, September, 1989.

[239] Tai, K.C. "On Testing Concurrent Programs". In *Proceedings of COMPSAC'85*, IEEE Computer Society Press, Los Alamitos, pp. 310-7, 1985.

[240] Tai, K.C. and Su, H.K. "Test Generation for Boolean Expressions". In *Proceedings of COMPSAC'87*, IEEE Computer Society Press, Los Alamitos, pp. 278-83, 1987.

[241] Tai, K.C. "What to Do Beyond Branch Testing?". In *Proceedings of 6th International Conference on Testing Computer Software*, Washington D.C., June, 1989.

[242] Tai, K.C. "Condition Testing for Software Quality Assurance". In *Proceedings of COMPASS'89*, IEEE Computer Society Press, Los Alamitos, pp. 31-5, June, 1989.

[243] van Tassel, D. *Program Style, Design, Efficiency, Debugging and Testing.* Prentice-Hall, 1974.

[244] Taylor, R.N. and Osterweil, L.J. "Anomaly Detection in Concurrent Software by Static Data Flow Analysis", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 265-77, May, 1980.

[245] Taylor, R.N. and Kelly, C.D. "Structural Testing of Concurrent Programs". In *Proceedings of Workshop on Software Testing*, IEEE Computer Society Press, Los Alamitos, pp. 164-9, July, 1986.

[246] Trauth, C.A. and Woolsey, R.E. "Integer Linear Programming: A Study in Computational Efficiency", *Management Science*, vol. 15, no. 9, pp. 481-93, May, 1969.

[247] Tuttle, M.R. and Low, D. "Videoscope: A Nonintrusive Test Tool for Personal Computers", *Hewlett-Packard Journal*, vol. 30, no. 4, pp. 58-64, June, 1989.

[248] Viravan, C. and Dunsmore, H.E. "Where Design Testing Fits In", *IEEE Software*, vol. 7, no. 3, pp. 105-6, May, 1990.

[249] Wallace, D.R. "The Validation, Verification and Testing of Software: An Enhancement to Software Maintainability". In *Proceedings of Conference on Software Maintenance (CSM'85)*, IEEE Computer Society Press, Los Alamitos, pp. 69-78, November, 1985.

[250] Wallace, D.R. and Fujii, R.U. "Software Verification and Validation: An Overview", *IEEE Software*, vol. 6, no. 3, pp. 10-7, May, 1989.

[251] Wang, H.S., Hsu, S.R., and Lin, J.C. "A Generalized Optimal Path-Selection Model for Structural Program Testing", *Journal of Systems and Software*, vol. 10, no. 1, pp. 55-63, July, 1989.

[252] Warnier, J.D. *Logical Construction of Programs.* Van Nostrand Reinhold, New York, 1977.

[253] Warren, H.S. "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations", *Communications of the ACM*, vol. 18, no. 4, pp. 218-20, April, 1975.

[254] Warshall, S. "A Theorem on Boolean Matrices", *Journal of the ACM*, vol. 9, no. 1, pp. 11-2, 1962.

[255] Weiser, M. "Programmers Use Slices When Debugging", *Communications of the ACM*, vol. 25, no. 7, pp. 446-52, July, 1982.

[256] Weiser, M. "Program Slicing", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-7, July, 1984.

[257] Weiser, M.D., Gannon, J.D., and McMullin, P.R. "Comparison of Structural Test Coverage Metrics", *IEEE Software*, pp. 80-5, March, 1985.

[258] Weyuker, E. and Ostrand, T. "Theories of Program Testing and the Application of Revealing Subdomains", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 236-46, May, 1980.

[259] Weyuker, E.J. "An Empirical Study of the Complexity of Data Flow Testing". In *Proceedings of Second Workshop on Software Testing, Verification, and Analysis (TAVS-2)*, IEEE Computer Society Press, Los Alamitos, pp. 188-95, July, 1988.

[260] Weyuker, E.J. "The Cost of Data Flow Testing: An Empirical Study", *IEEE Transactions on Software Engineering*, vol. SE-16, no. 3, pp. 121-8, March, 1990.

[261] Weyuker, E.J. and Jeng, B. "Experiences with Data Flow Testing". In *Proceedings of 7th International Conference on Testing Computer Software*, San Francisco, pp. 219-24, June, 1990.

[262] White, L.J. and Cohen, E.I. "A Domain Strategy for Computer Program Testing", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 247-57, May, 1980.

[263] White, L.J. and Sahay, P.N. "A Computer System for Generating Test Data using the Domain Strategy". In *Proceedings of SOFTFAIRII - 2nd Conference on Software Development Tools, Techniques and Alternatives*, IEEE Computer Society Press, Los Alamitos, pp. 38-45, December, 1985.

[264] White, L.J. and Sahay, P.N. "Experiments Determining Best Paths for Computer Program Predicates". In *Proceedings of International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, Los Alamitos, pp. 238-43, August, 1985.

[265] White, L.J. and Wiszniewski, B. "Path Testing of Computer Programs with Loops using a Tool for Simple Loop Patterns", *Software - Practice and Experience*, vol. 21, no. 10, pp. 1075-102, October, 1991.

[266] Wilde, N., Huitt, R., and Huitt, S. "Dependency Analysis Tools: Reusable Components for Software Maintenance". In *Proceedings of Conference on Software Maintenance(CSM-89)*, IEEE Computer Society Press, Los Alamitos, pp. 126-31, October, 1989.

[267] Woodfield, S.N., Gibbs, N.E., and Collofello, J.S. "Improved Software Reliability Through the Use of Functional and Structural Testing". In *Proceedings of 2nd Annual Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, Los Alamitos, pp. 154-7, March, 1983.

[268] Woodward, M.R., Hedley, D., and Hennell, M.A. "Experience with Path Analysis and Testing of Programs", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 278-86, May, 1980.

[269] Woodward, M.R. "Mutation Testing - An Evolving Technique". In *Colloqium on Software Testing for Critical Systems*, BCS Press, pp. 3.1-3.6, June, 1990.

[270] Yau, S.S., Collofello, J.S., and MacGregor, T. "Ripple Effect Analysis of Software Maintenance". In *Proceedings of COMPSAC'78*, IEEE Computer Society Press, Los Alamitos, pp. 60-5, November, 1978.

[271] Yau, S.S., Grabow, P.C., and Weems, B.P. "A Binary Relation Representation for Program Models". In *Proceedings of COMPSAC'82*, IEEE Computer Society Press, Los Alamitos, pp. 188-95, November, 1982.

[272] Yau, S.S., Nicholl, R.A., and Tsai, J.J.P. "An Evolution Model for Software Maintenance". In *Proceedings of COMPSAC'86*, IEEE Computer Society Press, Los Alamitos, pp. 440-6, 1986.

[273] Yau, S.S. and Kishimoto, Z. "A Method for Revalidating Modified Programs in the Maintenance Phase". In *Proceedings of COMPSAC'87*, IEEE Computer Society Press, Los Alamitos, pp. 272-7, October, 1987.

[274] Yau, S.S., Nicholl, R.A., Tsai, J.J.P., and Liu, S.S. "An Integrated Life-Cycle Model for Software Maintenance", *IEEE Transactions on Software Engineering*, vol. SE-14, no. 8, pp. 1128-44, August, 1988.

[275] Yau, S.S. and Chang, P.S. "A Metric for Modifiability for Software Maintenance". In *Proceedings of Conference on Software Maintenance(CSM-88)*, IEEE Computer Society Press, Los Alamitos, pp. 374-81, October, 1988.

[276] Yellin, D. "A Dynamic Transitive Closure Algorithm", Technical Report RC-13535-61958, IBM Research, June, 1988.

[277] Zadeck, F.K. "Incremental Data Flow Analysis in a Structured Program Editor". In *Proceedings of ACM Sigplan'84 Symposium on Compiler Construction*, ACM Press, New York, pp. 132-42, June, 1984.

[278] Zang, X. and Thompson, D. "Public-Domain Tool Makes Testing More Meaningful", *IEEE Software*, vol. 8, no. 4, pp. 109-10, July, 1991.

[279] Ziegler, J., Grasso, J.M., and Burgermeister, L.G. "An Ada-Based Real-time Closed-Loop Integration and Regression Test Tool". In *Proceedings of Conference on Software Maintenance(CSM-89)*, IEEE Computer Society Press, Los Alamitos, pp. 81-90, October, 1989.

[280] *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, 1983.

[281] *Standard FORTRAN Programming Manual*, 2nd Edition, 1972.

[282] *"IEEE Standard Glossary of Software Engineering Terminology"*. IEEE Computer Society Press, Los Alamitos, 1983.

[283] *"1986 Results of Survey on Software Testing"*, Quality Assurance Institute Report 1987.