Durham University

Durham E-Theses

Strong mutation testing strategies

Ishbel M.M. Duncan

How to cite:

Duncan, Ishbel M.M. (1993) Strong mutation testing strategies. Doctoral thesis, Durham University.

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a https://etheses.durham.ac.uk/id/eprint/5771/ is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
https://etheses.durham.ac.uk

Strong Mutation Testing Strategies

Ishbel M.M. Duncan

Ph.D. Thesis

University Of Durham

Computer Science Division

School of Engineering and Computer Science

1993

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

Strong Mutation Testing Strategies

Ishbel M.M. Duncan

Ph.D. Thesis

University Of Durham

Computer Science Division

School of Engineering and Computer Science

December 1993

Submitted for the Degree of Doctor of Philosophy

2 8 MAR 1994

Strong Mutation Testing Strategies

Ishbel M.M. Duncan

Ph.D. Thesis December 1993

Mutation Testing (or Mutation Analysis) is a source code testing technique which analyses code by altering code components. The output from the altered code is compared with output from the original code. If they are identical then Mutation Testing has been successful in discerning a weakness in either the test code or the test data. A mutation test therefore helps the tester to develop a program devoid of simple faults with a well developed test data set. The confidence in both program and data set is then increased.

Mutation Analysis is resource intensive. It requires program copies, with one altered component, to be created and executed. Consequently, it has been used mainly by academics analysing small programs. This thesis describes an experiment to apply Mutation Analysis to larger, multi-function test programs. Mutations, alterations to the code, are induced using a sequence derived from the code control flow graph. The detection rate of live mutants, programs whose output match the original, was plotted and compared against data generated from the standard technique of mutating in statement order. This experiment was repeated for different code components such as relational operators, conditional statement or pointer references. A test was considered efficient if the majority of live mutants was detected early in the test sequence.

The investigations demonstrated that control flow driven mutation could improve the efficiency of a test. However, the experiments also indicated that concentrations of live mutants of a few functions or statements could effect the efficiency of a test. This conclusion lead to the proposal that mutation testing should be directed towards functions or statements containing groupings of the code component that give rise to the live mutants. This effectively forms a test *focused* onto particular functions or statements.

Acknowledgement

The author would like to acknowledge the Science and Engineering Research Council (SERC, UK) for the award of a research studentship and British Telecommunications Research Laboratories (Martlesham Heath) for their financial and scientific support through a Co-operative Award in Science and Engineering (CASE studentship).

Especial thanks for guidance and patience are due to my supervisor, Dave Robson, and to Malcolm Munro for his encouragement. Thanks must also go to fellow researchers for their advice and stimulus; Dave Hinley, Jenny Newton, Dave Kinloch, Mohammed Messaoudi, Richard Turver and Nick Horner. Grateful thanks must be given to the army who supported throughout the term of the research; friends, family, childminder and cleaner without whom the load would have been too much to bear.

Final acknowledgements must go to Ian Parry, for his scientific advice and help throughout, to Eluned who kept all in perspective and to Muffy and Klio for sitting on the thesis and purring at the low times.

Ishbel Duncan

December 1993

To Ian and Eluned

Statement of Copyright

The copyright of this thesis rests with the author. No quotation from it should be published without her prior written consent and information derived from it should be acknowledged.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

'What we anticipate seldom occurs'
'what we least expect generally happens.'
B. Disraeli

The following chapter describes the purpose and contribution of the research discussed in this thesis. An outline of the thesis is also given.

1.1 Purpose of the Research

The general objective of the research undertaken and described in this thesis was to apply a particular testing technique to large, multi-function source code. The testing technique, called **Mutation Analysis**, has usually been applied to small programs. Mutation Analysis (MA) is resource intensive and was generally considered to be too expensive to apply to large test programs or systems. However, MA is also regarded as one of the most stringent source code testing techniques currently available. Consequently, an application to large, industrial systems could have enormous potential in terms of system confidence. An experiment was devised to investigate the application of MA to large programs and to discover if different application strategies could improve the efficiency of a mutation test.

1.2 Contribution of Research

A goal of the research undertaken in this thesis was to apply MA to multi-function programs and to discover any guidelines or principles that could be applied to the analysis of large scale code. As the work progressed, it was theorized that deriving a mutation sequence from the control flow of a program could improve the efficiency of a code test. Other mutation tools generate mutations on a Textual, or statement by statement, basis. Consequently, a comparison between different code traversal mechanisms was necessary to determine if any efficiency benefits were to be gained from control flow driven mutation testing.

The research sponsors had requested that code written in the C language should be tested. This provided the opportunity to develop the only tool available to perform Strong MA on C source code. Strong, Weak and Firm MA are three variations of the same basic technique. They are discussed in detail in Chapter 3. The prototype tool developed was the only system which analysed code by control flow driven mutation. It provided an opportunity to examine code for predisposition to particular faults and to fault congregation. The effect of code coverage on the test results could also be investigated. Induced faults could also be analysed for their effect on subsequent code execution. The effect of a mutation at a particular position within the code control flow graph may be viewed on the code output, or alternately, be hidden in the workings of the code.

The criteria for success for the research includes :

- A working prototype to analyse control flow driven mutation and compare this with Textual mutation.
- The development of general guidelines to enable future large scale code tests to be undertaken.
- Information regarding the effects of similar mutations in different parts of the control flow graph.
- General knowledge or points of interest regarding the analysis of C code.

1.3 Thesis Outline

The remainder of this thesis is organised as follows:

Chapter 2 summarises the field of software testing within the framework of the Software Life-Cycle. Functional, Structural and Fault Based techniques are outlined and Mutation Analysis is placed in context.

Chapter 3 details the history, theory and practice of MA . The variants named as Strong, Firm and Weak are introduced and compared. MA is also compared with data flow testing to show that it is a powerful technique. Currently available tools and developments are described, including the recent developments addressing the problems of applying MA to large programs.

Chapter 4 demonstrates the need for a new approach to MA if it is to be applicable to large scale code. The problems of using MA to analyse code with large system resource requirements are noted and a strategy is proposed to address some of these issues. In contrast to the developments in the U.S.A. of sampling mutants or using high speed processors to reduce the cost of a test, the approach described uses the structure of the test code to drive the mutation test. This approach does not reduce the cost of a test, but gathers information regarding weaknesses in the test code or test data set. It is this information that can aid the reduction of future test costs.

Chapter 5 describes the **Grail** prototype mutation system which was built to compare control flow driven mutation with Textual order mutation. The **Grail** incorporates a preprocessor, designed and built by another researcher, which determines the connectivity between the test code statements. This information is used by the main part of the tool to create and execute mutant programs in a sequence based on the test code control flow. The last section of the **Grail** mutation system plots the rate of detection of live mutants against mutants generated for Textual order mutation and for the control flow mutation sequences. A Mutation Metric is defined to compare the efficiency of each of the code traversal mechanisms.

Chapters 6 and 7 detail the results of the **Grail** experiments on five programs. These

test programs range in size from 37 to over 1800 lines of code. Two of the programs are well known in the testing literature and the remaining three are multi-function programs. These tests show the difference in efficiency between the standard, Textual method of mutating code and control flow driven mutation. The experiments demonstrate that some mutations are very fragile and die easily whilst others are more likely to die if they occur towards the start of execution. The term **Zombie** mutants is introduced to describe the mutations, normally assignment operator mutations, which can die but be re-born depending on the test environment.

Chapter 8 reviews MA and the contribution made by this thesis. An assessment of the value of the research is made and recommendations for future research are proposed.

Chapter 2

An Overview of Software Testing

'For knowledge too is itself power',
Francis Bacon

This chapter briefly describes the software life-cycle and the testing phases contained within it. As the thesis is concerned with software testing, functional, structural and fault based software testing strategies are outlined and compared showing the strengths and weaknesses of each generalised technique.

2.1 Introduction

Software is employed to control large and complex industrial, commercial and life critical processes. Pressman [76] states that some 60% of project software development budgets are spent on testing, verification and validation. **Software Testing** is the process of executing a program with the intention of finding errors in the code. It is the process of exercising or evaluating a system or system component by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results [1]. The objective of testing is to show incorrectness and testing is considered to succeed when an error is detected [69]. An **error** is a conceptual mistake made by either the programmer or the designer [59] or

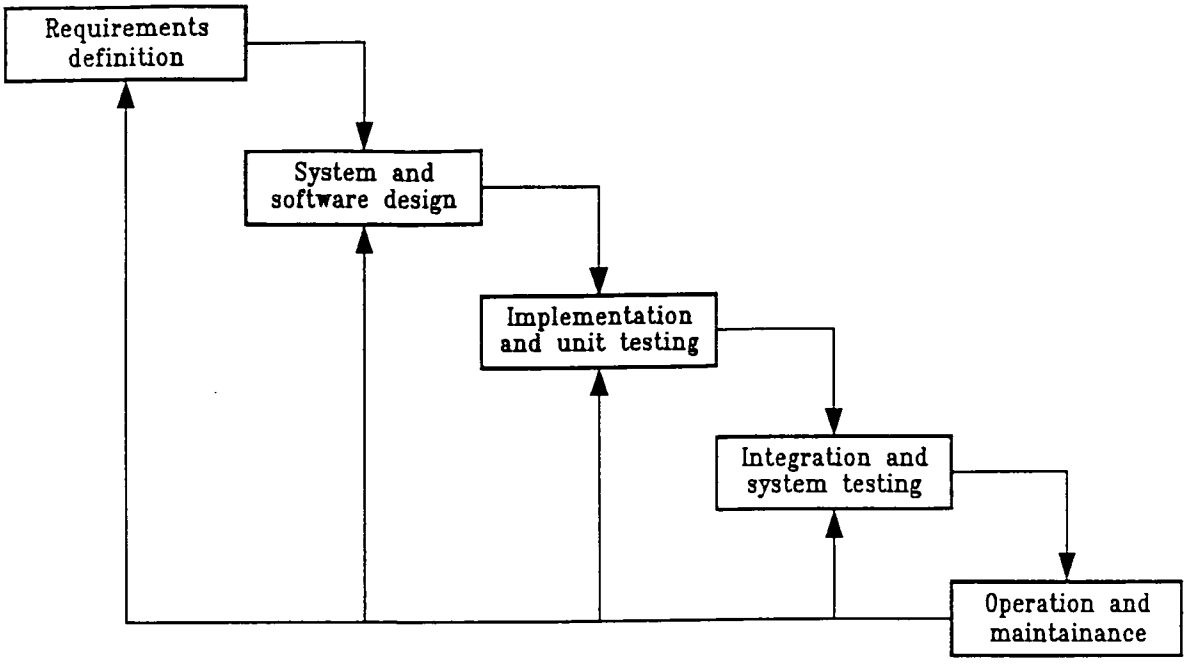
a discrepancy between a computed value and a theoretically correct value [1]. A **fault** is a specific manifestation of an error. An error may be the cause of several faults [59, 69]. A **failure** is the inability of a system or component to perform its required function within the specified limits [1]. A failure may be produced when a fault is executed or exercised. Testing should not be a distinct phase in system development but should be applicable throughout the design, development and maintenance phases [76].

The **Software Life-Cycle** is a description of the development of a software system from conception to redundancy. It models the software development as a phased set of activities which overlap and pass information to each other [82]. Once problems are identified in each phase, the information should be passed back to a previous stage for correction or adaptation, thereby describing a life-cycle which is a sequence of iterations of the development phases.

2.2 The Software Life-Cycle

The first phase of the software life-cycle is **Requirements Analysis** and **Specification**. See Figure 2.1 [82]. At this stage the **Functional** and **Non-Functional** requirements of a proposed system must be elicited from a customer. The functional requirements describe the system software in terms of its processing and input/output needs. Non-functional requirements describe the constraints of the system such as type and capacity of machine, response time, recovery and failure modes. The input to this phase is usually either a discussion between the customer and the specification writer or a document written in a non-specificational, natural language form presented to the specification specialist. The phase output is a specialised description of the system requirements, precise enough for a software designer to understand the customer's needs. A common problem that arises at this early stage of system development is a lack of precision and inconsistencies in the specification. The effect of these faults may not be evident until later in the software life-cycle. The requirements specialist should then update the specification prior to design and coding alterations.

Figure 2.1: The Software Life-Cycle



The next phase in the software life-cycle is **Systems** and **Software Design**. Taking input from the software requirements specification documentation, the designers produce a system architecture. This is an overall description of the hardware and software systems. The software designers represent and describe the units, or modules, of code ready for translation into a machine readable language. Here a **Unit** is taken to be a single function, procedure or subroutine which fulfills a particular task within the system. Common errors arising from the design phase come from inadequate understanding of and poor requirements specifications. Software design errors often evolve from problems with unit interfacing such as type and call order incompatibilities. The design phase output is a detailed description of the program units written in a formal language or pseudo code ready for translation by a programmer.

The next phases of the software life-cycle are concerned with the **Implementation** and **Integration** of software units. The design is translated into a group of programs or units. These are tested individually to ensure that each unit matches its design and requirements specification. Common coding errors found at this stage include typing mistakes, statement omission and errors in predicates such as the use of the wrong

logical operator. Passing this stage, the system is then formed by the integration of units in a pre-defined strategy such as **Top-Down** or **Bottom-Up**. The top-down approach starts with analysis of the main program. Individual functions called by the main routine are inserted one at a time, calls to others being replaced by simple, stub routines which may only consist of a print statement to simulate a correct call. Bottom-up integration merges the functions by the use of a driver routine which simulates the function calls. Design faults are commonly found faster in the top-down development approach whereas simple coding faults are discovered at a higher rate in the bottom-up approach [82]. By slowly merging functions into a system by either strategy, there is less risk of a large number of errors being missed in the integration. Errors are more likely to be localised to particular routines when the integration proceeds slowly.

The last phase of the software life-cycle is the software **Use** or **Operation** and any required **Maintenance**. Maintenance descibes any modification to the system and may be required as a result of error discovery (corrective maintenance) or because of alterations to the system requirements or environment (adaptive maintenance). As such, the system may undergo another cycle of requirements specification, design, coding and testing. The software life-cycle describes these phase iterations until the system becomes obsolete. The software development and maintenance process is not linear but is a sequence of iterations of activities [82].

2.3 Testing and the Software Life-Cycle

Each phase in the software life-cycle has a distinct end-product such as the requirements specification documentation, program unit design and program unit code. Each end-product can be checked for conformance with a previous phase and against the original requirements. Thus, errors can be detected at each phase of development. The cost of detecting and fixing an error is well documented [7] and is known to be more costly as the system develops. An error found during the operation phase is the most costly to fix. **Validation** and **Verification** should occur throughout the software life-cycle. Verification is the process of evaluating each phase end-product

to ensure consistency with the end-product of the previous phase. Validation is the process of testing software, or a specification, to ensure that it matches user requirements. Software testing is that part of validation and verification associated with evaluating and analysing program code. It is one of the two most expensive stages within the software life-cycle, the other being maintenance [23]. Software testing of a product begins after the development of the program units and continues until the product is obsolete.

This thesis is concerned with the testing of program code and as such, the following section outlines the software life-cycle phases to which code testing is applicable.

2.3.1 Unit Testing

Unit Testing, see Figure 2.1, refers to the analysis of individual units which can stand alone without requiring the processing of other units. The input generated from other units can be simulated for the purposes of testing a unit in isolation. A unit is a logical subtask of the system under development and can be a subroutine or function of any size. Marick [62] states that a unit is a single routine or a small group of closely related routines, normally less than 100 lines of code (LOC) in size. Ince [47] states that a unit can be one or more subroutines which carry out a common task. This thesis assumes that a single unit, function or subroutine can be of any size and programs, comprised of one or more units, can be several hundred or thousand LOC. Programs can be made up of modules where a module is considered to be a logically separate part of the code [1]. This thesis assumes that modules are separately compilable sections of a system and each module may contain one or more units. Unit testing also refers to module testing.

The aim of unit testing is to ensure that the unit matches its specification. It is usually a **Structural Testing** activity, i.e. some account is taken of the internal layout of the code. Components, and combinations of components, in the code must be exercised, that is, covered by the test inputs. Test data must be selected to verify specification properties and/or relations between program components. Structural testing is also known as **White Box** or **Glass Box** testing. These terms refer to

the tester having knowledge of the code layout and structure and designing tests to analyse the flow of control and data within a program.

Test Data must be developed for use at some stage prior to or during unit testing. A **Test Input** is an input value for a variable under test conditions. Test data is a particular instance of inputs required for one program execution under test conditions. A **Test Case** is documentation specifying test data and the predicted results as well as a set of execution conditions for a test item. A **Test Suite** or **Test Set** is a collection of test cases [35].

Programming faults can be detected by **Dynamic** or **Static** testing techniques. A structural test is a form of dynamic analysis. That is, the code is checked through the application of a test suite. Unit testing may also be done through static analysis. Static techniques involve the inspection of code without its execution.

2.3.2 Integration Testing

Integration Testing or **Subsystem Testing** is concerned with verifying the design and requirement functions. Units are integrated into a system using a predetermined strategy such as top-down or bottom-up. The growing system should be regularly tested. The interfaces between the installed units should be checked for compatibility, that is, the calling sequences and function parameters should be analysed for legality, order and type. There should be 100% call coverage to ensure confidence in the inter unit connections. As more units are integrated into the system it is possible to test design functions and requirements such as response time and fail safe routines [66]. The integrated units should also be tested for compatibility with the hardware environment into which the system is to be placed.

2.3.3 System and Acceptance Testing

System and **Acceptance Testing** are concerned with the execution of test cases to evaluate the whole system with respect to the user's requirements. A system test

checks for unexpected interactions between the units and modules and also evaluates the system for compliance with functional requirements. An acceptance test is the process of executing the test cases agreed with the customer as being an adequate representation of user requirements.. These are often called **Black Box** or **Functional** tests. These terms make reference to the tests being unconcerned with the internal structure of the code. They are concentrated on analysing the performance of the code with respect to the test suite.

At any stage in the software life-cycle errors may be discovered. This may lead to changes in design and/or code update resulting in a re-application of any of unit, integration, system or acceptance tests. The process of re-testing a unit during its development is called a **Revision** test in this thesis [24]. This is similar to a **Regression** test which occurs during maintenance when a system is being modified. Regression testing is the selective re-testing of a system or unit to verify that modifications have not caused unintended side-effects and that the system or unit still complies with the current specification.

2.4 Functional Testing

Functional testing is a Black Box technique and attempts to verify that the abstract functions written in the system specification are present and behave correctly. Requirements documents can be tested, or analysed, by review to discover if there are any requirements missing, redundant or capable of simplification [41]. The formal review is often used for analysing design documents although there exist helpful aids such as consistency checkers and data dictionaries to analyse design logic and data element transitions. These are essentially static techniques in which the analysis does not require execution. A functional specification may be written in a structured natural language or in a specificational language such as **Z** or **VDM**. The techniques for generating test data from such specifications are still in the early stages of research and development.

The external behaviour of code, and its functions, can be analysed to ensure that

all intended features of a software system are present. The aim is to test **Input Domains** for each category of input and **Output Domains** for each code variable. A domain is the range over which a variable is valid. Test data can be derived from the specifications, usually manually, to exercise the boundaries and midranges of the input domains. This technique can generate a large amount of data and so **Equivalence Partitioning** is often used to identify a finite set of domains with constraints. Each equivalent partition dictates a test case required for its traversal. For example, the statement 'if $a > b$ then S1 else S2' generates two partitions of the domain prior to execution of the statement. One partition incorporates the values of '$a > b$' and the other of values '$a <= b$'. **Special Values** testing can be applied to select test data on the domain boundaries and extremal values chosen to check the accuracy of the function computed. Problems arise with the large number of test cases required for partition and special values testing, but the techniques are good for detecting domain and extremal value variable faults. **Cause-Effect Graphs** translate equivalence partitions into decision table form via boolean operator descriptions of the output conditions in terms of the input variables. Test data can be generated from the decision table form, reducing the number required [69]. Hamlet [37] and Duran [26] state that partition testing is more effective than testing with randomly generated data. However, random testing is more cost effective in terms of time and man-power.

Functional testing techniques are not as well developed as structural techniques although some research has indicated that a high level of code coverage, that is the number of statements executed, can be generally high [43], but the range is variable depending on the abilities of the tester. However, functional testing techniques tend to generate large quantities of test data which is not necessarily precise enough to locate code errors and analyse problems with the specific language used in the code. Some theoretical and empirical work has been done on the viability of randomly generated test data to perform adequate functional testing. Myers [69] suggests randomly generated data gives poor statement coverage and functional checking but Duran and Ntafos [26] and Ince and Hekmatpour [48] suggested, after empirical studies, that random generation of test data was a cheap and effective way to perform functional testing. Studies of comparisons between functional and structural testing strategies generally suffer from analysis of either small programs or a small number of test subjects (testers). Most suggest functional testing is a useful preliminary to code testing,

creating a base of useful test cases with generally high code coverage [6, 68, 70]. A major advantage of functional testing is its ability to determine missing functionality or code, but the disadvantages lie in the lack of determination of the quality of the test that has been conducted.

2.5 Structural Techniques

Structural testing examines source code and analyses what is present in the code. Structural testing techniques are often dynamic, meaning that code is executed during analysis. This implies a high test cost due to compilation or interpretation, linkage, file management and execution time. Test cases are derived from analysis of the program **Control Flow**. A Control Flow graph is a representation of the flow of control between program regions such as a group of statements bounded by a single entry and exit point. Structural testing cannot expose errors of code omission but can estimate the test suite adequacy in terms of code coverage, that is, execution of components by the test suite or its fault finding ability.

2.5.1 Coverage Measures

A measure of test thoroughness is made with respect to some code coverage criterion such as **Statement**, **Branch** or **Path Coverage** . The first of these measures is a percentage formed from the number of statements executed by the test suite over the total number of executable statements in the code. The second is a similar measure on the number of logical branches executed. That is, the true and false paths of every condition should be executed at least once. The path coverage metric is the most difficult to calculate and achieve 100% coverage. Total path coverage is the execution of all independent paths, that is, unique paths, within the code. If the code contains indeterminate loops, the number of independent paths is unbounded. Loop modified path testing specifies that each loop is executed 0, 1 and more than once. Coverage can be achieved theoretically but is usually impractical for programs of more than a moderate size. A typical unit testing metric is 100% statement coverage and 85%

branch coverage [66]. Of these basic coverage metrics, statement and branch testing methods are simple to understand and implement. Untraversable code can be found by failing to find test data which traverses sections of code. However, statement coverage is not sufficient to detect faults. Basili and Selby [6] found no correlation between maximum statement coverage and the number of faults found. Branch testing is a poor test of code with few conditionals or loops in which only one iteration is required to fulfill coverage requirements. Both methods lack error detection power and other more rigorous, structural testing strategies have been developed.

Test Effectiveness Ratios, **TER**s, have been defined, [91], to represent increasing degrees of code coverage. These can be applied to both static and dynamic techniques. A static technique is one in which the code is evaluated without execution [1]. TER_1 represents a test in which all statements of a program have been executed by the associated test suite. This is equivalent to 100% statement coverage. TER_2 represents a test in which all the program branches have been executed. This is equivalent to 100% branch coverage. This is sufficient for **Simple Predicates**, i.e. predicates with no logical operators. An example of this is:

if (P) then

However, requiring that all branches are executed does not imply that a **Compound Predicate** has been fully tested. A compound predicate contains one or more logical operators. An example of this is :

if (P and Q) then S1 else S2

where S1 is a statement executed when the conditional evaluates to True and S2 is executed when the conditional is False. The test inputs

Test 1 :	P = True	Q = True
Test 2 :	P = False	Q = True

exercise both branches but do not analyse the effects of Q being assigned to False.

Conditional Testing requires that the test suite exercises all combinations of each component in a predicate. That is, all components must evaluate to both True and False on different test inputs. For the compound predicate example at least one more test case must be added in which Q evaluates to False. This does not necessarily imply that every branch will be executed.

The test inputs

Test 1 :	P = True	Q = False
Test 2 :	P = False	Q = True

satisfy conditional testing but not branch testing requirements for the example statement.

A higher level of test is described by the TER_3 metric. This is a measure of the number of **Linear Code Sequence and Jumps** (LCSAJ) exercised by the test suite. An LCSAJ represents a subpath through the program code. It consists of a sequence of consecutively executed statements from a single entry point to a single exit point. An entry point may be, for example, the start of the code, the beginning of a true or false branch of a conditional statement or the body of a loop. An exit statement is a control statement changing the flow of control to another entry point or termination. An example of this would be the terminal statement of a function or the statement to which flow would be passed if a loop was bypassed. (See Figure 2.2).

Girgis [32] and Hennell et al [40] found that 85% of all seeded faults in small Fortran programs could be determined by LCSAJ coverage. The strongest code coverage metric is path coverage. Howden, [46], showed that path testing is the single best method for exposing errors. However, due to the presence of indeterminate loops the number of program paths can be astronomical and possibly infinite for even the most trivial of programs. Some paths may be infeasible due to the presence of contradictions in the predicates [40] and path testing does not ensure coverage of the requirements specifications.

A high test coverage does not necessarily imply a high rate of fault detection. Test cases must not only traverse the code but must also exercise special boundary conditions. Test coverage criteria are difficult to accomplish when analysing large programs. As the criteria become more stringent a high coverage is harder to achieve. Special test cases must be developed to improve the level of coverage. However, as the test progresses each individual test case will improve the coverage statistic by a small, and possibly marginal, amount for an increasing cost factor.

Figure 2.2: An example of LCSAJs

```
1 begin
2 read(a,b,c);
3 if a ≥ b and b ≥ c then
4     if a = b or b = c then
5         if a = b and b = c then
6             print('equilateral')
7         else
8             print('isosceles')
9     else
10        print('scalene')
11 else
12     print('illegal')
13 end.
```

LCSAJ	Statement Number		
	START	END	TARGET
1	1	3	12
2	1	4	10
3	1	5	8
4	1	6	13
5	8	9	13
6	10	11	13
7	12	13	13

The LCSAJ is comprised of a linearly connected start and end statement and a target statement to which control is passed.

2.5.2 Domain and Boundary Testing

Domain Testing is a form of path coverage and as such is mentioned here under structural testing strategies. It was mentioned earlier as being a form of functional testing wherein the functionality of the test code was used to determine test input domains. Path domains are a subset of the program input that cause execution of unique paths. The input data can be derived from the program control flow graph. Test inputs are chosen to exercise each path and also the boundaries of each domain. For example, in a program analysing the height and weight of a population, the input domain is $height \times weight$, where the inputs are real numbers greater than zero and bounded by some upper limit. The statement

if (weight $\geq$ 50.0 and height $\geq$ 1.8) then S1 else S2

would partition the path domain into two from the true and false evaluation of the predicate. A true evaluation would result in statement S1 being executed and S2 is executed when the predicate evaluates to false. Test inputs for branch, statement and domain testing could be

Test 1 :	weight = 48.0	height = 1.8
Test 2 :	weight = 50.0	height = 1.8

A boundary test would incorporate test inputs on and slightly off the boundaries of the paths. To determine data slightly off the boundary an amount, ϵ, must be added or subtracted to the value which lies on the boundary. When the boundary is determined by an integer, ϵ is 1. That is, the value 1 must be added or subtracted to the value in a predicate to form an input value which will be close to the domain boundary. When working with real numbers the procedure is more complex. The value ϵ must be the smallest number distinguishable by the base system of the program under test. For example, if the reals are single precision ϵ could be of the order of 0.000001. To test the boundary of '$weight = 50.0$' the following three input cases would be valid :

Test 1 :	weight = 50.0	height = 1.8
Test 1 :	weight = 50.0	height = 1.6
Test 1 :	weight = 50.000001	height = 1.8

Great care must be taken when working with real numbers in predicates because of the precision problems of reals. Boundary testing aids the identification of these problems and errors of path selection. However, domain and boundary analysis is only suitable for programs with a small number of input variables and with simple linear predicates.

2.5.3 Data Flow Testing

Data Flow Analysis studies the sequences of actions and variables along program paths. It can be considered and applied as both a static and as a dynamic technique.

Test data must traverse all the interactions between a variable definition and each of its uses. The program path between a variable definition and a use without an intervening definition is known as a *DU path.* Variable uses may be in predicates, *p uses*, in which the variable is referenced in the conditional expression. A computational use, *c use*, refers to all other references of variables. Clarke et al [17] state that testing *all-DU-paths* subsumes all other data flow testing criteria defined by Rapps and Weyuker [78]. The *all-DU-paths* criteria requires every definition clear subpath to be loop-free or to include a simple, one iteration, loop. The data flow testing criteria includes, in an increasing level of rigour, *all-defs, all-p-uses* and *all-uses.* Data flow testing is difficult to apply to units of more than a small size and low complexity. The cost of application is also difficult to assess; Weyuker stated that the cost of *all-defs* assessment is linear in the number of assignment statements and the cost of *all-uses* is quadratic and *all-du-paths* is exponential in the number of conditionals present in the code [87]. However, these were considered theoretical limits and empirical evidence on small Pascal programs shows the costs to be linear in the number of conditionals. Data flow testing is considered viable for incorrect uses of variables and constants as well as misspelled identifiers. As with code coverage strategies, data flow testing cannot detect missing statements.

Data Flow Anomaly Analysers detect problems with the definition and use of variables in the code under test. A **Data Flow Anomaly** can be one of three conditions :

- a variable defined then defined again on the same program path without an intervening use, a *DD* anomaly.
- a variable is defined and then undefined without an intervening use on a program path, a *DU* anomaly.
- a variable is referenced without a prior definition on a program path, a *UR* anomaly.

Data flow anomaly analysers cannot differentiate between faults in the code and those which have been deliberately introduced by a programmer. An example of this is initialisation of a numeric variable to zero prior to it being assigned a value later in the execution. This initialisation is recommended because some paths may

exist which do not re-define the variable but use it, i.e. a definition-use path, which, without the initialisation would be an undefinition-use path. Having a definition-definition path where the former is an initialisation statement is not as dangerous a construct as an undefinition-use path.

Structural testing analyses what is present in the code and cannot expose errors of code omission. Dynamic tests further require the execution of code. As the code increases in length and complexity the resources required to test it increase rapidly. However, simply increasing the code coverage does not indicate a higher confidence of fault removal. Structural coverage techniques aid the development of test cases but do not indicate the adequacy of those test cases in locating code faults.

2.6 Fault Based Testing

Fault based testing attempts to show the absence of certain classes of faults in code. Anomaly Analysis analyses code for uninitialised or unreferenced variables, parameter type checking etc.. The main technique, and its variants, which perform fault based testing is Mutation Analysis.

2.6.1 Mutation Analysis

Mutation Analysis (MA) is a fault based technique for determining the adequacy of a test suite in terms of its test effectiveness. A **mutant** is a copy of the original test program with one component, such as an operand or operator, altered to simulate a syntactically correct programming fault. The syntactic transformation is a **mutation**. The statement

$$\text{while (index} > 10) \text{ do}$$

could be mutated to

$$\text{while (index} \geq 10) \text{ do}$$

Thus, MA simulates simple programming errors. The test suite must be enhanced

until all non-equivalent mutants are detected by generating incorrect output. MA incorporates strategies from coverage, data flow anomaly and domain testing strategies. For example, the above statement has to be traversed by the test input

index = 10

to differentiate between the correct and the incorrect version. All statements, branches and (some) paths must be executed to differentiate incorrect mutants from the original program. By altering the constant 10, in the example, to the constants 11 and 9, boundary testing is performed. The test inputs must include cases of

index = 9, 10 and 11

to detect those mutants. By altering the definition of 'index' or replacing a use of it by another integer variable in scope, data flow anomalies can be detected.

MA provides the tester with guidelines for the development of the test suite. However, it is resource intensive requiring a large number of mutants to be created and executed on the test suite. Research [2, 10, 23, 80] indicates that the number of mutants varies with the number of code statements and variables squared. A mutation test of a large program, such as would be found in an industrial or commercial environment, would require the generation of a substantial number of mutants. A test on this scale would require management of resources. A strategy must be found to make mutation testing applicable to unit testing in a reasonable time scale and without tying up valuable resources such as time and manpower. MA is one of the most thorough of testing techniques. Empirical studies [10, 32, 33, 64] have shown it to be more stringent than other techniques. This thesis addresses the management and application of MA to large scale programs and the problems encountered.

2.7 Summary

The software life-cycle is described and the testing phases within it reviewed. Testing is required at all phases of the software life-cycle from specification through to system modification. The generalised strategies of functional, structural and fault based testing are discussed and compared. Functional testing is shown to be useful preliminary to structural and/or fault based testing. The techniques should be seen as

complimentary; functional testing requires specification coverage, structural ensures analysis of code sequences and fault based techniques give a measure of component analysis and test data adequacy in terms of fault removal confidence.

Chapter 3

Mutation Analysis

'You can observe a lot just by watching'

Yogi Berra

This chapter outlines the theory, practice, problems and current developments in Mutation Analysis. Following an introduction and background information, sections 3 and 4 detail the theory and practice and section 5 the variants of Mutation Analysis. The next sections discuss the strengths and weaknesses of the variant techniques and also compare Mutation Analysis with other testing techniques. Section 8 describes the current developments and available tools.

3.1 Introduction

Coverage criteria were outlined in Chapter 2 as being measures of test thoroughness. However, as Galvin [31] states

> *If no attempt is made to evaluate the effectiveness or thoroughness of a set of test cases then the test cases can be a misleading sense of security.*

Budd [10] defined Mutation Analysis (MA) to be

A method for evaluating the effectiveness of a set of test cases for a given assertion (program).

DeMillo [22] stated a central goal of MA as

... to determine when a software system has been adequately tested.

MA is a measure of test thoroughness and also performs testing. MA has been applied to specificational languages, Ada, Pascal, FORTRAN, COBOL and C. Research elsewhere has concentrated on its application on vector processors to reduce the time required for a full test. The general mutation strategy itself has spawned variants called Weak, Firm and Strong Mutation in an effort to make the technique more efficient. Current developments include analysis of applicable fault subsets to reduce the test resource requirements, statistical sampling of mutants, schematic descriptions of code and the application of the technique to vector and parallel processors to improve the test efficiency.

3.2 Background

DeMillo [22] states that the earliest mention of mutation-like mechanisms dates back to an unpublished manuscript written in 1970. Technical and published reports describing similar concepts appeared in 1976 and 1977 by Hanson et al [38] and Hamlet [36]. The technique was refined and named by Budd, DeMillo, Lipton and Sayward in 1978 and 1979, [3, 11, 13, 20, 58], and much research followed into the early 1980s. MA did not gain widespread acceptance by the testing community because it was considered computationally expensive and other, less stringent, forms of testing were advocated. The late 1980s saw a renaissance of interest and activity in MA aided by the advent of more powerful processors and cheap memory. Proponents argue that it is the most thorough of available techniques, encompassing control flow, data flow, domain and boundary strategies and has great value with regard to critical systems testing and risk reduction [22].

3.3 Mutation Analysis Theory

MA assumes that a program under test is almost correct. That is, an experienced programmer will write code which differs from the correct version by small, syntactically correct, faults. This is known as the **Competent Programmer Hypothesis** [2, 3, 10, 19]. Acree et al [3] postulated that

> *A competent programmer, after completing the iterative programming process and deeming that his job of designing, coding and testing is complete, has written a program that is either correct or is almost correct in that it differs from a correct program in 'simple' ways.*

Therefore a test which concentrates on small, syntactically correct, alterations may discover faults in code.

A test suite, T, must differentiate the *correct* original test program, P, from its close *incorrect* neighbours, P'. A test suite distinguishing P from all its incorrect neighbours is called **Adequate** and provides assurance that all non-equivalent neighbours are detectable. However, DeMillo [22] suggests that a finite adequate test suite may never exist and the notion is computationally intractable. He introduced the concept of **Relative Adequacy** to characterise the ability of a test suite to differentiate between the test program and its close neighbours. These are programs which differ from the *correct* code by simple, single alterations. The neighbourhood, N, of the test program is therefore restricted to programs containing a single, simple fault. As the test program, P, is close to the correct version by the Competent Programmer Hypothesis, Relative Adequacy is still a powerful measure of a test suite. Although the test program is considered to be correct, any faults discovered imply that it is revised and the updated version becomes the new test program.

A program being tested, P, is meant to compute a function F with input domain D. A finite set of test cases, T is a subset of the input domain, $T \subseteq D$, and a particular test case t is an element of T, $t \in T$. $P(t)$ is the result of executing P with t. Correctness and relative adequacy are therefore defined more formally [71] as

- P is *correct* if $P(t) = F(t)\ \forall t \in D$.
- T is an *adequate test suite* for P that computes F if $P(t) = F(t)\ \forall t \in T$ and for all programs Q such that $Q(D) \neq F(D)$, $\exists t \in T$ such that $Q(t) \neq F(t)$.
- A is a set of programs. A test suite T is *adequate relative* to A if
  $P(t) = F(t)\ \forall t \in T$
  and for all programs $Q \in A$ if $Q(D) \neq F(D)\ \exists t \in T$ such that $Q(t) \neq F(t)$.

The programs in A can be chosen to represent particular faults that the tester must choose. Thus, a testing technique using relative adequacy requires the tester to distinguish between programs seeded with, or having, specific faults.

The outlined theory also clarifies that correctness cannot be shown through testing and that adequate test sets are difficult, if not impossible to achieve. Relatively adequate test suites are attainable and, once achieved, leave the tester knowing that some faults are not present in the test code. The quality of the test suite can be measured and used with confidence to exercise code.

3.3.1 Test Suite Adequacy

A **Mutation Score** of a test suite T, for a program P, is the ratio of the number of mutants killed through the application of T, to the number of non-equivalent mutants [22].

$$MS(P,T) = \frac{DM(P,T)}{(M(P)-EM(P))}$$

Where $MS(P,T)$ is the mutation score for the test suite, T, on program P. $M(P)$ is the number of mutants generated for P, $EM(P)$ is the number of equivalent mutants of P and $DM(P,T)$ represents the number of dead mutants of P after application of T.

If the mutation score for a test suite is 1.00 then that test suite is adequate relative to the set of mutants. Offutt [71] states that

> *... a program that has been successfully tested against a (relatively) adequate test suite is either correct or contains a fault that has not been modelled by the mutants. ... A program that has been successfully distinguished from its mutants has been thoroughly tested.*

Evaluating test suite adequacy gives the tester an indication that certain faults are unlikely or not in the test program and gives a measure of the quality of the test suite. A test suite with a higher mutation score than another demonstrates a greater error detection capability. Relative Adequate test suites developed for MA have been empirically analysed and shown to be of a higher error detection quality than those developed for other strategies. This is discussed in more detail in section 3.7.

3.3.2 The Coupling Effect

Empirical research in the U.S. [72, 73], has shown that if a program with simple errors can be differentiated from the correct version by a (relatively) adequate test suite, then a larger neighbourhood of programs with complex (several) component faults can also be differentiated. This is termed the **Coupling Effect** and is the subject of some debate [61, 67, 72, 73]. DeMillo and Mathur [23], state that MA is guaranteed to reveal an error if it can be simulated by a mutation and there exist simple, single faults in the code. Other faults can be determined if they have computational ties with the mutation, that is, the mutation may not directly simulate a fault, but in attempting to devise a test case to analyse a mutation, the tester is directed towards a fault. Faults may be found which are not necessarily localised by mutations. DeMillo and Mathur also studied the errors found during the development of the text processor TEX, [23]. They concluded that complex errors may be found by determining, or searching for, simple ones. However, they acknowledged that the majority of errors cited by Knuth, [52], in the development of TEXwere unclassifiable and of those that were, only one-fifth could be described as simple errors. Most of the complex errors were related to missing code which cannot be modelled by Mutation Analysis. Also, in their experiments, DeMillo and Mathur analysed small programs and acknowledged the need for Coupling Effect trials on larger, industrial code.

Morell [67] postulated that MA was feasible only if the Coupling Effect is valid. He termed two mutants as being *coupled* if a test suite kills the individual mutants but does not kill the mutants formed of their combination. (N.B. Morell's definition of coupling is different from other empirical researchers.) Morell argued that there are relatively few cases for which two mutations couple which strengthens the case for MA being an effective fault based technique.

Lipton and Sayward [58] analysed Hoare's Find program [42], and generated over 27,000 higher order mutants. A 2-order mutant is a mutant formed from the combination of two simple mutations, a 3-order mutant is formed from the combination of three simple mutations and so on. Lipton and Sayward found that all the 2-order mutants were killed by the relatively adequate test suite developed for killing simple mutations. Applying the test suite to higher order mutants resulted in a mutation score of near unity, indicating that the Coupling Effect does hold. However, their experiment was on one small FORTRAN program of less than 30 lines of code and the higher order mutants tested were less than 5% of the total possible.

Offutt [72] experimented with three small FORTRAN programs, each of less than 30 lines of code. He demonstrated that as the number of mutant programs grows exponentially with the generation of the higher order mutations, it is important to validate the Coupling Effect for fault based strategies such as MA. Offutt's experiment showed a mutation score for 2-order mutants on a relatively adequate test suite of over 0.9996 and those that remained alive were due to variations in the test suite development. In a second experiment, he developed test suites that were not mutation relatively adequate and applied those to the 2-order mutants. The mutation scores for the 2-order mutant tests were higher than for the 1-order mutants implying that the more complex a fault is, the more likely it is to be detected. Offutt concluded that the Coupling Effect is true in a very large percentage of cases, supporting Morell's theoretical work.

Marick [61], classified 102 faults in industrial code and concluded that only 23% were simple faults and therefore could be simulated by MA. Only 8% were compound, that is made up of a combination of several simple faults, for which the Coupling Effect could be tested. He stated that mutation testing was effective on a small

proportion of faults and that the Coupling Effect was required to hold for faults of omission and complex faults if MA was to be effective in general. This shows the limitation of MA; it can detect simple faults but there is still doubt over its ability to detect compound or complex faults. More empirical and theoretical work is necessary to prove or disprove the Coupling Effect, especially on large programs. This thesis assumes a close neighbourhood of programs with single component alterations.

3.4 Mutation Analysis Practice

A test suite T is comprised of one or more test cases, t. The test suite is prepared for code analysis by manual or automatic means [71]. The code under test, P, is executed by T and the output $P(T)$ stored. P is then mutated and the syntactically correct variants, the mutants, are formed. See Figure 3.1 for examples of mutant statements.

Figure 3.1: Mutant Statements

Program P is altered to form mutants P'_N. N.B. The last two mutants of the first example assume a boolean variable R in scope and mutant P'_8 of the second assumes an integer variable sum in scope.

P	if (P and Q) then			
P'_1	if (P and P) then		P'_2	if (Q and Q) then
P'_3	if (P and not P) then		P'_4	if (not P and Q) then
P'_5	if (P or Q) then		P'_6	if not (P and Q) then
P'_7	if (P) then		P'_8	if (Q) then
P'_9	if (P and R) then		P'_{10}	if (R and Q) then

P	while (input ≥ 0) do			
P'_1	while (input ≤ 0) do		P'_2	while (input < 0) do
P'_3	while (input > 0) do		P'_4	while (input $= 0$) do
P'_5	while (input $\neq 0$) do		P'_6	while (input ≥ 1) do
P'_7	while (input $\geq$ -1) do		P'_8	while (sum ≥ 0) do

Each mutant program has one component, C', altered from the original component, C,

in the test program. The altered component is known as a **Mutation**, or a **Mutant Component**, and is effected by a **Mutation Operator** or **Mutagen**, such as the computational equivalent of *replace a variable reference with another identifier in scope*. The mutant programs, P', are executed on T and their output compared with $P(T)$. If for any $t, P(t) \neq P'(t), where\ t \in T$ then the mutant is considered **Dead** and removed from further analysis. If the mutant is not killed by any of the test cases, t, it remains **Live** and is checked for equivalence to the original program. **Equivalent** mutants often represent optimisations or de-optimisations of the test program. The recognition of equivalent mutants is done by human examination or by primitive heuristics [71]. An example of an equivalent mutant is given in Figure 3.2.

Figure 3.2: An Equivalent Mutant

```
if (a ≥ 0) then
 {
 .
 .
  if (a > 0) then
# if (a ≠ 0) then
 .
 .
```

The line marked # is equivalent to the line above.
To execute the second conditional statement, the variable a must be 0 or more. When a is positive, $a \neq 0$ is equivalent to $a > 0$.

Offutt, [71], defines three conditions to distinguish mutants from the test program.

- **Reachability**. A test input must traverse a path that includes the mutated component, a path P_R must be executed.
- **Necessity**. The mutated component must produce a state different from the test program, a path P_N must be executed.
- **Sufficiency**. The final state of the mutant must differ from the test program, a path P_S must be executed.

That is, the mutant component must be executed by at least one test case which

delivers a different outcome from the original program and that difference must be propagated throughout the execution.

Riddell et al [80], give four reasons for a mutant remaining live after the application of a test case. These are

- the mutant is equivalent.
- the mutant and the test program output the same results over the same paths for the given data, but another test case may kill the mutant.
- the mutant outputs the same result as the test program for a given test case, but a different path has been executed. This is known as **Coincidental Correctness**.
- the test data is inadequate and the altered component has not been executed.

3.4.1 Mutagens

Mutagens or **Mutation Operators** alter a component, C, in the test program to a syntactically correct component, C'. Some of the examples in Figure 3.1 showed that a variable can be mutated to another variable (or constant) of the same type, assuming that it is in scope. Logical and relational operators are mutated to members of the same operator group maintaining a syntactically correct, but altered, program. Table 3.1 gives examples of the types of mutagens that can be applied to procedural or block-structured languages. These operators simulate programmer or design errors as well as enforcing component coverage and analysis.

A mutation of $<$ to $\leq$ simulates the common fault of 'off-by-one' in which an error has been made in determining the path boundaries of the code. Problems of coincidental correctness in the test suite can also come to light with the application of mutagens. The result of a computation on a test input is coincidentally correct if the result is correct although the computation or path is incorrect. See Figure 3.3 for an example. Altering a variable identifier to another of the same type allows the detection of data

Table 3.1: Some Common Mutagens

Component Type	Mutagens
Relational Operators	Change to another relational operator
Arithmetic Operator	Change to another arithmetic operator
Logical Operator	Change to another logical operator
Constants	Change by ε, where ε is the smallest variation in the type detectable by the base type of the computer (1 for integers)
Variables	Change to another of the same type in scope Alter by ε to check boundary conditions
Conditionals	Negate and alter predicate components
Statements	Remove or move position
Pointer Variables	Increment or decrement the pointer value and change the variable to another of same type

flow anomalies. In languages such as C and Pascal which have dynamic variables, mutagens can simulate 'off-by-one' errors by moving pointer variables up or down the object list structures. The head and tail of the storage list may be altered to check special case manipulations and the temporary pointer variables as well as constant pointer identifiers replaced by one another to examine problems of identifier confusion, generation, management and manipulation of the list structures. Predicate faults can be discovered by altering the components of the predicate; replacing them with tokens of the same type, negating the components, altering the precedence of the predicate components or changing the logical operators.

Figure 3.3: An example of Coincidental Correctness

$$
\begin{array}{l}
\vdots \\
\quad c = a \uparrow 2 \\
\#\ c = a * 2 \\
\vdots
\end{array}
$$

The line marked # is a mutation of the previous line. If the only test case which traversed these statements included $a = 2$ then the output from these statements would be correct, although the second is an incorrect statement.

Another type of mutagen, as yet unimplemented but noted by researchers [24, 83], is

the environmental or portability fault simulator. Spafford [83] states that environmental bugs are limitations of precision or capacity such as memory or numeric storage limitations or system errors caused by a change in compiler. Mutagens should force the development of test cases which test the limits of memory allocation routines or numeric storage. Spafford suggests two new mutagens called IOFLOW and IUFLOW which return the value of the numeric to which they are applied unless overflow or underflow has occurred. When this occurs the mutagens abort the process. These mutagens then force the tester to construct test cases which use numerics close to the end of the numeric ranges. Any alterations to the system or compiler should then be obvious. Mutagens analysing environmental problems have not yet been designed for all languages and systems as they are unique to each language, compiler and system. However, the issues raised in testing for such problems shows the power of Mutation Analysis in its ability to adapt to changing testing requirements.

3.4.2 Cost of Mutation Analysis

Marick [60] states that costs are determined by the test input development time and effort, calculation of output correctness, execution time and resources and finally, the cost of re-application of the testing technique during maintenance. The cost of MA is considered to be the number of mutants generated and is therefore higher than most other testing techniques. This should be offset against the higher level of automation of mutation tools and the amount of information generated. Each mutant requires, at worst, compilation and execution on all the test cases. If the mutant is interpreted and the mutation applied to low level code, the altered statement must be inserted into the low level code stream before application of the test suite. Budd [10], considered the number of mutants to vary with the number of data references and the number of data objects. Acree [2] stated that the cost of MA is quadratic in the number of lines of code, a rule ratified by Riddell et al, [80] and DeMillo and Mathur, [23]. Offutt, Rothermel and Zapf [75] studied 28 small FORTRAN77 programs of less than 165 lines of code (LOC) and considered the most accurate predictor of costs to be the number of variables multipled by the number of variable references. However, they also confirmed previous estimates based on quadratics in lines of code or variables. It

should be noted that these cost estimates relate to small FORTRAN66, FORTRAN77 and Pascal code. Research has yet to confirm that the cost rules apply to larger units and to other languages.

3.5 Mutation Analysis Strengths

MA is considered to be resource intensive. Less intensive forms were developed [45, 88] and the original technique was renamed **Strong** Mutation Analysis. The other forms are now named as **Weak** and **Firm** Mutation Analysis and are described in this section.

3.5.1 Weak Mutation Analysis

Howden, [45], introduced the concept of Weak Mutation Analysis in his 1982 paper. The idea was derived from earlier work [29], and was designed to overcome the costs of testing with Strong MA.

Assume that a component, C, in program, P, is mutated to form component C' in program P'. In weak mutation testing, it is required that a test, t, must be constructed so that C and C' are executed when t is applied to P and P'. A test case t must also have the property that C computes a different value from C'. Howden [45], states that

> *On at least one such execution of C, C computes a different value from C'.*

Although the outcome of executing C' may differ from the outcome of executing C, it is still possible for the mutant P' to be live. That is, the outcome of the mutated component may differ from the original component, but the program output is unaffected by the change. Weak MA does not guarantee the exposure of all errors in the class of errors defined by the mutagens.

Howden,[45], defined components as elementary computational structures such as variable references and assignments, arithmetic relations and expressions and boolean expressions. Offutt and Lee [74] state that Howden's components are not defined precisely enough for empirical research on the value of Weak mutation testing in comparison to Strong mutation testing. They use a definition of component corresponding to Woodward and Halewood [88], which refers to a component as the location where the states of the original and mutant program are compared. That is, a component is the program state at some point after the mutated token has been executed.

It is possible to simulate, or enable, several Weak mutation component alterations in any one program execution. Weak MA is primarily concerned with determining that each component evaluates to a different value at least once in the test suite. Table 3.2 summarises the conditions for the Weak mutagens, *wrong variable*, *off-by-one*, *wrong relational operator* and *arithmetic expression* [34].

Table 3.2: Adequacy measures for Weak Mutation

Component	**Weakly Adequate Test Data**
Variable Definition	New Value
Variable Reference	Unique Value
Relational Expression (LHS op RHS)	Values where LHS-RHS $= -\varepsilon, 0, \varepsilon$
Arithmetic Expression	Non zero value

N.B. ε is the smallest number distinguishable on the base system

Weak mutations are not always enabled but can be determined a priori to improve the test suite. Weak MA is a mechanism for improving the quality of the test suite without necessarily executing the mutant programs on the data. However, a Weak mutation adequate test suite is not Strong mutation adequate. In Offutt's terminology [71], P_N has been executed, meaning that a test case has traversed the component C' and the outcome is different to the same test case traversing the original C. Execution of P_S, a path propagating the mutation's effect to the end of the execution, is not necessarily achieved.

Weak mutation is a refinement of branch testing in which branches and other com-

ponents must be executed at least twice on error related data. Budd [14] states that Weak MA incorporates branch and predicate boundary testing by satisfying that

- A predicate is exercised on data which results in both true and false paths being executed.
- For relational predicates, test data should evaluate the expression to a negative, a zero and a positive value. (LHS - RHS $= -\varepsilon, 0, \varepsilon$)
- Test data should be chosen close to predicate boundaries (off-by-one).

Marick [61] stated the **Weak Mutation Hypothesis** as, referring to Offutt's three conditions of detection, that reachability and necessity must imply sufficiency. That is, if a mutant survives weak mutation it may also survive strong mutation. Offutt [72], had declared the Weak Mutation Hypothesis to hold true for 61% of cases taken from a small study of FORTRAN programs of less than 30 lines of code. Marick analysed 5 routines of between 9 and 206 lines of code and injected faults manually. He concluded that the Weak Mutation Hypothesis was likely to hold in more than 70% of cases and that Weak Mutation testing coupled with branch testing was highly effective at locating simple faults. However, the routines he tested were non arithmetic and the faults were manually introduced by himself as the tester.

3.5.2 Firm Mutation Analysis

Woodward and Halewood, [88], introduced the notion of Firm Mutation Analysis as a way of performing mutation testing on program fragments. The theory behind the strategy was first mentioned in [80]. Firm MA covers the range of effect of a mutation over a slice of the program execution at least as long as the execution of a single statement.

Different mutation results can be generated from a change in a statement which may be executed more than once in any execution. Altering any one of the stages in the execution at which

- the mutation is applied (t_{change})
- the stage at which the change is reversed (t_{undo}) and the outcomes compared
- the actual components are compared

can affect the outcome of a mutation experiment. More than one firm mutation can be analysed in any one execution. This can be achieved by reversing the effect of t_{change} at t_{undo}. Execution continues from t_{undo} with the program states equivalent to the states in the original test program. Here a component is considered to be a program state. Firm MA corresponds to a mutation in a program slice which persists for more than one execution such as would occur when the component resides in a program loop. It should be noted that t_{undo} may never be reached because of the mutation effect. This is also a problem with Strong MA where t_{undo} corresponds to the end of execution. An endless loop caused by the mutation may result in t_{undo} not being executed.

Output Comparison

The actual components or values output can affect the outcome of a mutation test [80]. Output can be

- actual physical output (characters or binary).
- final values of data objects.
- a trace of data definitions and/or references.
- a trace of the control flow.

A character by character comparison is expensive and perhaps too rigorous for the test. An extra carriage return in the output may not be considered an important enough reason to kill a mutant. Riddell et al [80], declare output files to be **Strongly Equal** if they match. If the non blank characters are identical then the files are considered **Weakly Equal**. An extreme situation is a test program which has no output

generating 100% live mutants. A program written by a novice, liberally strewn with print statements of variable values, is more likely to generate fewer live mutants than one with a terminating printout of results. Depending on where the mutations occur, the relative position of an output statement and the actual variables or states output, a mutant can be live or dead. See Figure 3.4 for an example. More comprehensive examples are given in [88].

Figure 3.4: Outcome dependency of a code segment

```
.
.
  a = x ;
  b = y ;
# b = -y;
.
.
  surf-area = Pi ↑ 2 * (b ↑ 2 − a ↑ 2) ;
.
.
  print(surf-area);
  print(surf-area, a, b);
```

The line marked # is the mutant statement of the line above. The print statements determine whether the mutant program is live or dead. A mutant with the first print statement is always live. If the variable b is not reassigned prior to printing, a mutant with the second print statement will be dead.

Using Firm MA, mutation testing can become part of the development process of a unit. Selective mutation on logical sequences such as loops or functions can aid the detection of problematic constructs and development of test cases. Particular components or program states can be chosen for comparison with the original execution to give a clearer understanding of the effect of a mutation. Firm MA is less expensive than Strong MA as partial executions can be performed with the use of an interpreter. It also detects mutations that develop a different outcome on a statement level and is therefore stricter than Weak MA in determining components that have an effect on following code. Firm MA also provides control over component output for result comparison. The disadvantages of Firm MA are that it is difficult to assess the test

suite in terms of Weak or Strong Adequacy and there is, as yet, no systematic basis on which to select code regions for Firm mutation testing.

3.6 Mutation Variant Comparisons

Measuring the test suite adequacy is a mechanism for comparing testing techniques. Several studies have compared the forms of mutation testing with each other as well as other common testing strategies.

Horgan and Mathur, [44], compared Weak with Strong MA. They stated that any test suite which is strong mutation adequate is also weak mutation adequate and the converse can be nearly true. They considered a state of a program to be values of variables at a specific point in the code execution. Their probabilistic analysis showed that weakly adequate test suites have a high probability of being strongly adequate. Horgan and Mathur are currently building a tool to investigate their hypothesis.

Offutt and Lee, [74], define four variants of weak mutation analysis to determine whether weak mutation is viable and which compare point, t_{undo}, is optimum.

- **EX-WEAK** is expression weak mutation which compares the program states after the first execution of the innermost expression that surrounds the mutant. For statement mutations such as statement deletion or replacement, the comparison was done at the state immediately following the mutation execution. For conditional expression operators an EX-WEAK mutation would compare the states immediately following the popping of the conditional from the run-time stack.

- **ST-WEAK** is statement weak mutation and compares the states after the first execution of the mutated statement. For statement mutations, EX-WEAK and ST-WEAK are identical.

- **BB-WEAK/1** refers to basic block weak with one execution. A basic block here is a maximal sequence of instructions with one entry and one exit point.

The mutation state is compared after the basic block containing the mutant statement has been executed. A mutation within a loop is therefore analysed at the end of the first iteration of the loop.

- **BB-WEAK/N** compares the program states after each execution of the basic block.

Offutt and Lee found that many mutations within loops could not be killed after the first iteration. That is, mutations were live on a BB-WEAK/1 test but dead on a BB-WEAK/N test. They report that if a mutant is killed under Strong MA it can be killed under BB-WEAK/N. Equivalent mutants under BB-WEAK/N are also Strong mutation equivalent but the converse is not true. Equivalent mutants for BB-WEAK/1, ST-WEAK and EX-WEAK are different from BB-WEAK/N and Strong.

Comparing test data on Weak and Strong mutation systems, they found that the Weak mutation score was always greater than the Strong. This means that the requirement for Weak mutation is weaker than the requirement for Strong mutation testing. Offutt and Lee generated 100% adequate test suites for each Weak mutation variant and computed the Strong mutation score. They discovered that the Strong mutation score for the BB-WEAK/N test suites were less than those generated for ST-WEAK and BB-WEAK/1. They indicated that this may be an attribute of the low complexity of the test programs which were all less than 29 lines of code.

In their second experiment, Offutt and Lee generated test suites which were less than mutation adequate and computed the Weak mutation scores. They discovered that BB-WEAK/N was not significantly more expensive than BB-WEAK/1 and that Weak Mutation was generally more powerful if applied to small components at ST-WEAK or BB-WEAK/1 levels. They concluded that ST-WEAK and BB-WEAK/1 were more powerful a measure than BB-WEAK/N and that it was difficult, if not impossible, to relate Weak and Strong mutation scores because the scores differed greatly across the suite of small programs. However, they recognised Weak mutation as a cost effective alternative to Strong mutation testing for non critical testing but acknowledged that it was not proven if Weak mutation could be as effective as Strong mutation when applied to large scale code. Offutt and Lee stated that a cost effective

mechanism would be to generate 100% ST-WEAK coverage for components followed by a full mutation on those mutants which were equivalent under ST-WEAK. This principle has been followed by Weiss and Fleyshgakker [86] who determine whether a mutant is live under Weak mutation before proceeding to full mutation. This is discussed in more detail in section 3.8.2.

3.7 Technique Adequacy Comparisons

Criticisms of Mutation Analysis centre on the expense in time and computational resources required in applying the technique. Some research has been done in comparing MA to other structural techniques to assess its worthiness.

Mathur [64] compared the test data adequacy criteria for data flow and strong mutation testing. He stated that a test suite, T, **Weakly Satisfies** the ALL-DU paths criteria for a data flow test, if T causes the execution of each feasible path from a variable definition to a variable use. He defined the data flow adequacy as

$$T^d = \frac{NumberOfDUPathsCovered}{NumberOfFeasibleDUPaths}$$

To compare the data flow and mutation scores of a test suite, Mathur scored each technique on an adequate test suite of the other technique. That is, he developed a test suite which was data flow adequate and then used the test suite to analyse a program under a mutation test and vice versa. Mathur used 18 small FORTRAN and Pascal programs for the experiment, each containing between 2 and 28 decisions. The FORTRAN programs were tested using the Mothra mutation testing tool and the, functionally equivalent, Pascal programs were analysed by the ASSET data flow test tool [18, 30]. The programs were tested by 7 students in 2 separate groups. The results indicated that mutation relatively adequate test suites were invariably data flow adequate but that the converse was not true. That is, a mutation relatively adequate test suite was a *stronger* test of a program. Mathur acknowledged that his experiment was based on small programs and inexperienced testers and stated that a test on large scale code was desirable.

Budd [10] compared path testing with strong mutation testing using Howden's earlier

path analysis work [46]. Budd reported that MA would detect 20 out of 22 faults analysed by Howden. This compared with the 13 detected by Howden using path analysis. All path testing, ALL-PATHS, subsumes data flow testing. Budd's earlier work therefore supports Mathur's recent empirical study to show that strong mutation analysis is a stronger or stricter code test than data flow testing.

The **FORTEST** system developed by Girgis and Woodward [32, 33] incorporated control flow coverage, data flow and weak mutation testing. FORTEST reported on the statement, branch and LCSAJ coverage of FORTRAN77 programs by the applied test suites. The tool can display the outcome of data flow path criteria [78] and weak mutation testing with respect to completeness of the test suite. Data flow testing was shown to expose some classes of faults not discovered by the other techniques such as wrongly placed statement. FORTEST applied weak mutation testing to numeric quantities only but could analyse programs containing more than one subroutine. The data flow analysis had to be performed on single unit programs only. Thus FORTEST was restricted to single numeric routine programs for the comparison between data flow, control flow and weak mutation testing strategies. Data flow testing was successful at discovering problems associated with the wrong relational operator, wrong variable reference and incorrect use of constant as well as computation faults, especially *all-c-uses*. (That is, all computation uses of variables as opposed to predicate uses.) The weak mutation criteria used found wrong variable definitions, missing computation and all domain errors. The best strategy was considered to be testing for 'off-by-one'. The authors concluded that control flow was the most efficient technique for fault discovery, especially when the ALL-LCSAJs strategy was used. However, they emphasised the need for complementing control flow with data flow and weak mutation testing strategies as they guide the development of the test suite. Although Girgis and Woodward relate some interesting findings of technique to error discovery, it should be noted that the programs used were small FORTRAN77 programs and that the relationships have not been proven for large scale code. However, they did analyse the faults found with respect to the statements that were traversed. That is, faults occurring in untraversed statements were removed from the statistics. Weak mutation discovery rates remained static and the control and data flow rates decreased. Weak mutation and data flow anomaly testing, as demonstrated by FORTEST, do not produce output regarding conditions unless the

particular statement under scrutiny has been traversed by a test case. Control flow testing still remained as the best of the three techniques measured by FORTEST.

3.8 Current Mutation Tools and Developments

Mutation Analysis has undergone a renaissance since the late 1980s. Current developments centre on cost reduction strategies for Strong MA and the applicability of Weak mutation as an alternative for non critical unit testing. The following sections outline available tools and their abilities.

3.8.1 Current Tools

MA tools should incorporate statement, branch, predicate and domain testing within the applicable mutagen set. They are usually highly automated and generate a great deal of information. Some resource management is required in the generation and execution of test cases and determination of correct output. This is sometimes manual although some systems incorporate automatic test input generators. The creation and management of mutant programs is a particular problem although the available tools either create mutants serially or alter a low level code representation of the original program to reduce the cost of compilation and storage.

One of the benefits of Mutation Analysis is that it is applicable to non procedural languages. Woodward [90] has applied a subset of mutagens to the OBJ specificational language in a prototype tool called **OBJTEST**. He identified simple operator and variable replacement mutagens, operator attribute alteration and equation removal to force equation traversal as useful tests.

Girgis developed FORTEST [32] for experimental evaluation of data flow, control flow and weak mutation on FORTRAN77 code. The tool was a prototype in which data flow anomalies could be detected on single unit programs and weak mutation performed on numeric components. As such, FORTEST was useful for the analysis

of strategies on single unit numeric code.

Marick incorporated several coverage strategies and Weak Mutation testing in his Generic Coverage Tool (**GCT**) [62]. Although not designed as a mutation tool, GCT aids the development of test cases. GCT analyses code written in C and reports on branch, switch case, loop and routine coverage. It displays inadequacies in relational operator boundary analysis and indicates the extent of multiple condition coverage. Weak mutation is used for determining the analytical ability of the test suite. Marick divides Weak mutation into operator and operand coverage. Operator coverage is determined by checking that all alternative operators are removed by the choice of appropriate test inputs. Operands are replaced by constants and local variables of the same type. GCT does not replace identifiers with globals, the reason Marick gives for this is that it would increase the number of mutants and test inputs without any increase in effectiveness. However, no experimentation has been done to ratify this reasoning. GCT employs the principles of weak sufficiency described by Marick in [61] and referred to in Section 3.5.1. Weak sufficiency forces a stronger test to be developed to ensure that the effect of an alteration has some impact on the code following. For example, the statement 'if $a < b$' is mutated to 'if $a < c$'. If '$b <> c$' and 'c' was unique then Weak mutation as proposed by Howden [45] has been achieved. Marick proposed that not only should 'c' be unique but that '$(a < b) <> (a < c)$'.

The most widely referenced mutation tool is possibly the **Mothra** software testing environment [18, 21]. Mothra was developed from expertise derived from the PIMS, EXPER, FMS and CMS mutation systems [3, 9, 12]. These early, prototype tools tested FORTRAN and COBOL code and many of the authors collaborated on the development of Mothra at Georgia Institute of Technology in the United States. Mothra was designed to be interactive and useful for both the naive and experienced tester. Mothra was developed by a large team over several years and is designed to be extensible to most High Level Languages. At present it supports FORTRAN77 testing and is said to be able to analyse code of some 10 to 100 million lines of code. Mothra's main features are :

- A test case generator called Godzilla [71] which creates path expressions from symbolic analysis of the test code. The path expressions and predicate con-

straints are used to generate test cases to traverse the path domains. Mothra also allows interactive entry of test cases.

- The translation of the source code into Mothra Intermediate Code (MIC), a postfix language, each statement forming a one-to-one correspondence with a source statement. Mothra executes the original and each mutant by interpreting the MIC instructions. A mutant generator program operates at MIC level.
- Subsets of mutagens can be chosen to test a particular fault type.
- Random samples of mutants can be generated and analysed to reduce the test.
- The removal of equivalent mutants once recognized (by the tester).
- The alteration of test case order for efficient killing of different mutant groups.
- An oracle providing user intervention or the use of a symbolic solver to determine output correctness.
- A debugger with access to the Mothra database to advise the user as to where to look for suspected bugs.
- The suspension and storage of a test to allow continuation at a future session.
- Application to other languages provided a translator exists from the language to the Mothra Intermediate Code.
- Mothra can run under an X-Windows environment, allowing icon driven commands.
- An ability to resource shift allows Mothra to take advantage of any network connection to a vector or parallel processor. This requires translations of the MIC to a vector form but decreases the overall test time.

Mothra cannot mutate references and calls to dynamic memory, user defined types and complicated control structures because of the mutations being applied at the MIC level. Consequently, Mothra is ideally suited to FORTRAN mutation and this explains why Mothra has not been applied to C code although the research group published a list of possible C mutagens for Mothra [4].

Mothra incorporates a large degree of automation and encompasses statement, branch, predicate and domain testing within the mutagens. It delivers a vast quantity of information via mutation scores. The developers admit problems with resource management, hence the interest in resource shifting. One mechanism for reducing the test cost is in the application of Weak mutation. Offutt and Lee [74] modified Mothra to compare mutated program components rather than compare output at the end of execution. Their system is called **Leonardo** from *Looking at Expected Ouput Not After Return but During Operation.* Leonardo uses all the mutagens available in Mothra (22) and is therefore the most powerful Weak mutation system available. However, Leonardo uses different definitions of component because of the complexity of some of the mutagens. This is a problem for all Weak mutation tools or methods. Some research has been conducted on the possible extension of Mothra to mutate Ada programs [5] but this has not yet resulted in an empirical study.

Mutation tools tend to be rather large requiring parsers, mutation generators, test case and mutant execution management. As larger programs are tested more time and memory management is required and some cost reduction strategies are necessary.

3.8.2 Cost Reduction Strategies

Riddell et al [80], researched limited Mutation Analysis as a mechanism for reducing the cost of a full mutation test. They studied relational, variable, arithmetic and character replacement mutagens on 35 small NAG routines written in FORTRAN66. (The Numerical Algorithms Group, NAG, supply mathematical and statistical library routines for general scientific use.) The researchers reported that branch testing could not always be effected by relational operator mutations and as such MA should complement structural strategies rather than replace them. However, the authors indicated that if more complex predicates were present in the code, Mutation Analysis was better than coverage techniques for detecting faults hidden in predicates. In comparison to coverage analysis strategies, they considered MA to be 100% more expensive in terms of resource usage and management. However, MA was more stringent and therefore of more value when analysing critical or important code. A

disadvantage of MA was seen in its lack of sensitivity in applying mutagens to all parts of the test code with no regard to structure. However, the use of limited mutation and directing the test to selected parts of the code may overcome this problem.

The Syntax Directed and Semantics Aided Mutation (SDSAM) strategy of Wu et al [92] is a mechanism for limiting mutation costs. SDSAM rules restrict a program mutation to a particular mutation type or application. For example, the rules may restrict mutations to variable references in predicates. Test Coverage Metrics (TCMs) can be applied to mutation subsets such as for variables, to give an indication of code coverage. For example, the TCM for variables is defined as the number of dead variable mutants divided by the total possible variable mutants converted to a percentage.

$$TCM_{var} = \frac{DeadVariableMutants}{TotalVariableMutants} * 100\%$$

To kill all the mutants of any component a test suite must be applied, such that the test cases are specifically defined to kill the mutants. To kill the five designated mutations of a relational operator on the statement '$x+y < c$' say, requires a minimum of three test cases with the constraints '$x+y = c$', '$x+y = c-\varepsilon$' and '$x+y > c$', where ε is the smallest number distinguishable by a system. That is, one test case on the predicate path boundary, one close to the inclusive border and one on the exclusive side. Thus, the number of test cases can be minimised and as in Weak mutation, some need not necessarily be applied if they ensure the death of a component mutation. The domain testing strategy provides an approach for revealing path boundary errors and coupled with coverage metrics give an indication of how well the test cases analyse the code. However, the system has not been developed but when compared theoretically with other strategies the authors state that the technique would be a valuable aid in determining test suite adequacy for mutant subsets such as boolean operator, variable reference and assignment.

The Static Data flow Aided Weak Mutation Analysis (SDAWM) strategy of Marshall et al [63], is a mechanism for analysing program operands. The technique involves statically analysing code with respect to data flow anomalies of program variables. By simulating the Weak mutation of variables the effect on path expressions can be determined. For example, an analysis of a particular program variable may result in

a path expression comprising an Undefined-Defined-Referenced (UDR) sequence. A, theoretical, mutation may alter the path expression to Undefined-Referenced (UR) which is anomalous and would therefore be detected by a Data flow analyser. The authors observed that mutations of referenced variables do not induce as many anomalies as that of variable definitions. The SDAWM strategy is effective for statically removing many mutants from a system, reducing the cost of a full mutation. As variable mutations lead to the greatest number of mutants, a system that can remove a great percentage of mutants and could be used as a preprocessor to either a Weak or a Strong mutation system would be of great value. A problem with this strategy is that many programs contain intentional data flow anomalies which would require handling by the developed system.

Offutt, Rothermel and Zapf [75], conducted an experiment to test the fault finding capabilities of a test suite relatively adequate for a reduced set of mutagens. Their experiment was performed on 10 small FORTRAN77 programs of between 10 and 48 lines of code. They noted, like other researchers [2, 10, 25, 65], that the scalar and array variable replacement mutagens generated the greatest number of mutants. Removing these mutations, test data was generated both automatically and manually to create test suites which were relatively adequate for the remaining mutants. These were named **Selective Mutation Adequate** test suites. When the two most prolific mutagens were removed from the test the test suites formed were 2-selective mutation relatively adequate. Removing the four most prolific mutagens resulted in 4-selective mutation relatively adequate test suites and so on. Several were generated for each program and the results averaged to remove bias in the statistics. The programs were then subjected to a full mutation and the mutants executed on the selective mutation test suites. The results showed that the 2-selective mutation test suites were almost 100% non-selective mutation adequate. The percentage cost saving, defined as

$$\frac{NonSelectiveMutants - SelectiveMutants}{NonSelectiveMutants}$$

was at least 14.83% and on average over the 10 programs, was 23.98%. The number of test cases had little effect on the results as the smallest selective mutation adequate test suite was as effective as the largest. Offutt et al [75] concluded that selective mutation is a cost efficient alternative to non selective mutation and postulated that the variable replacement mutants are easily killed by tests developed to kill other

Figure 3.5: A MetaProcedure

$$a = b + c$$

would be represented as

$$a = aorr(b, c, N)$$

where *aorr* is an arithmetic mutagen and N is the location on the program, possibly the line number, where the metaprocedure is to be applied.

mutagen types. They continued their experiment by generating 4-selective and 6-selective mutation adequate test suites, by removing constant and scalar replacement mutagens and array and constant replacement mutagens respectively. The mutation scores for non selective mutation using the 4 and 6 selective mutation test suites both averaged over 99%. The cost savings increased from an average of 41% on the 4-selective mutation test suite to over 60% on the 6-selective mutation adequate test suite. However, the experiments were conducted on small FORTRAN77 programs. Offutt et al are currently analysing larger programs to find whether their results hold true across a range of program size and complexity.

Untch, Offutt and Harrold [84], developed mutant schema representing the test program and its neighbourhood, N of simple mutants. The mutant schema is comprised of metaprocedures which represent metaoperators and metaoperands. Metaprocedures replace source statements and describe the possible mutations that can be performed on code. See Figure 3.5 for an example of a metaprocedure.

The system requires a driver to invoke the altered source file and create the meta-mutants, the mutants generated from the new source. The driver routine must direct which metaprocedures are to be invoked. The researchers compared a single, small program transformed into the Mutant Schema with the equivalent FORTRAN77 source tested under the interpretive mutation environment, Mothra. They found that the Mutant Schema test was over 4 times faster as it ran at compiled speeds. The authors also stated that the Mutant Schema allowed testing to be undertaken in an operational environment as the method was capable of producing compilable programs in the same language as the original source. This is an advantage over most tools which require the test program to be in a particular language, or subset. The

authors confirmed that work with large scale programs was underway.

Sahinoğlu and Spafford [81] demonstrated that statistical sampling of program mutants could reduce test time with high confidence. Their method involved applying mutants to a developing test suite and stopping the test when a predetermined ratio of mutants were dead. Their test was based on two small FORTRAN77 programs and they concluded that statistical sampling would be a great resource saving for programs for which 100% confidence was not required. Therefore, critical and large scale programs are not covered by this sampling method and Strong mutation testing is still required for confidence.

Several papers have been published on the use of vector or parallel processors for improving the efficiency of MA testing [15, 16, 55, 56]. The test program must be transformed into a canonical form and the test suite must be available. DeMillo, Krauser, Choi, Mathur and others [15, 16, 55, 56] experimented with various strategies for a mutation test including scheduling copies of a mutant on available processors and executing each mutant on a different test case. Another strategy was to apply different mutants on the same test case, removing the mutants that died and replacing them by another mutant. Those that survived were executed on the next test case. This later method, known as Mutant Priority, was more efficient than the previous strategy of test case priority. The most effective technique was to split the execution stream of a mutant. At the point of application of a possible mutation, the state of the test program was copied to available processors each affecting a particular mutagen. The processors all executed mutants of the same component on the same test case. The researchers found that there were problems in signalling the program state to other processors due to previous processes, other mutants, still executing. Utilisation of the processors was quite variable but could still result in a dramatic speed up of resource usage. This technique required alteration of the source file and the availability of super computers. The authors acknowledged these problems and are currently working on building translation and scheduling routines into Mothra so that users of that tool would not need to be aware of the underlying architecture. If Mothra was installed or had access to a super computer the user would benefit from an improved test time. Choi et al [15] have developed P^Mothra which is a system for allowing Mothra to run on either a hypercube, a vector or a (default) Sun

workstation. They are currently conducting experiments to study the error exposing abilities of mutation testing on large programs as executed on vector and hypercube systems.

Weiss and Fleyshgakker [86] researched serial algorithms for improving the efficiency of MA without parallel processors. Their technique involved storing the state of a component prior to mutation, as in a split stream approach, and using this state as the start state of a mutant. Their strategy determines whether a mutant would be killed under Weak Mutation on the application of a particular test case. Once a mutant is killed by Weak Mutation, it is executed to determine its outcome under Strong Mutation. Thus mutants are tested on mutation traversing test cases only and full execution is exercised only when warranted. A live Weak mutation mutant will always result in a live Strong mutation test and as such a full execution is unnecessary. The authors reported that the speed up of a test was at least proportional to the size of the program. Their analysis was theoretical but is to be used as the basis of an empirical study. They noted that if the test suite was well designed and had a high Weak Mutation kill rate then the speed up over conventional methods was minimal.

3.9 Summary

This chapter laid out the basic theory and practice of Mutation Analysis, a fault based, source code testing strategy. The two principles on which MA is based; the Coupling Effect and the Competent Programmer Hypothesis are outlined and discussed showing that there is some debate over the effectiveness and existence of the Coupling Effect. Some of the more common mutagens, or mutation operators, are outlined to show that simple faults are induced into code by small alterations on components. These simulate common programmer or design faults such as typographical errors, boundary condition or predicate faults.

The three strengths of Weak, Firm and Strong mutation testing are stated and shown to contribute to test case development and code understanding. Comparisons are made between the strengths in their ability to detect errors and they are also compared

to other testing strategies. Mutation testing is shown to be highly expensive in terms of resource management and usage but this is weighed against its greater testing ability. MA is shown to be a powerful technique, subsuming statement, branch, predicate and domain testing. It is also applicable to non procedural languages and can detect problems associated with machine upgrade and portability. MA is shown to be applicable to the unit testing of critical code.

The available research tools are described and the cost reduction techniques currently being evaluated are discussed. These techniques show that mutation analysis can be applied to large scale code testing and make it one of the most viable testing strategies available.

Chapter 4

A New Approach

'To seek out new life and new civilisations,
To boldly go where no-one has gone before.'
attrib. Gene Roddenberry

This chapter identifies some problems in testing large scale code and sets out the aims for an empirical study. A survey of common programming errors is summarised and demonstrates the diversity of errors generated by differing groups of programmers on a variety of software tasks. A strategy for mutation testing in the large is then outlined.

4.1 Introduction

Little testing research has been conducted on more than small programs or units. Most studies have concentrated on theoretical or empirical trials of small, less than 30 lines, FORTRAN, Pascal or COBOL code. These programs are interesting but lack complexity and length.

With the advent of more powerful processors and guidance from previous research a current research emphasis is to attempt analysis of larger programs such as would

be found in a commercial, industrial, or research environment. In the previous chapter, section 3.8.2, some theoretical approaches to large scale testing using Mutation Analysis (MA) were discussed. These included mutant sampling and the development of mutant schema for increasing the efficiency of a test. A full mutation test on a program of moderate size and complexity would be expensive in time and computational resources. This implies that mutation testing is either non-viable for large coded systems or its usefulness lies solely in small unit testing or in its applicability to critical code. It is important to discover whether this implication is true, whether a large scale test is practicable and if so, what problems or factors can be identified for improving test efficiency. This thesis describes an empirical study of MA applied to C code over a range of size up to some 2000 lines of code (LOC).

4.2 Problem Identification

Software testing research has concentrated on small programs for obvious reasons:

- Methods have to be clarified and tested prior to increasing program size and complexity.
- Error or fault groupings need to be understood and classified for comparisons between different testing techniques.
- Time and resource constraints of funded research do not provide for industrial trials.

Little research has focused on the testing of large scale code. In small code testing, MA has been found to be one of the most stringent testing techniques in empirical and theoretical work [32, 59, 64]. An empirical experiment of mutation testing on larger programs is both warranted and timely. If a system is deemed critical, that is, it monitors or processes information which is life threatening or saving, then Strong Mutation Analysis is perhaps the best technique for analysing the code once a functional test has been performed. However, industrial systems are large and therefore

require much time and computational resources to perform adequate testing. A mutation testing trial on large programs may provide some areas of interest with regard to testing adequacy and allow comparison of some testing techniques. Some problems of testing in the large can be immediately identified:

- Resource management. The number of mutants generated from a program is quadratic in the number of variables or statements. As larger units are tested, the number of mutants to be managed will grow to excessive amounts. Each mutant, under Strong MA, will require the same disk memory allocation and for compilation and execution, the same system resources of CPU time, data file access etc. as the original, test program. A strategy is required to manage the test resource requirement.

- Mutant Generation. In an experiment limited by time and resources, it is not possible to generate all the mutagens as described in the literature. A few groups of representative mutagens must be selected for the language used. A language and a mutagen sample must be chosen.

- Error Congregation. As errors are known to congregate, [69], a strategy which tests code in distinct basic blocks of code in a methodological fashion should aid recognition of problematic code regions and therefore increase error understanding and detection. Collating errors found under basic blocks or functions indicates problems associated with fault impact more than simply numbering errors by their statement. This requires a test strategy which can be aimed or focused on particular functions or basic block sequences.

- Unit or Integration Testing. MA has been applied mainly to single unit programs. An application to multi-unit programs is warranted because, in an industrial environment, a single programmer will write code modules comprising many subprograms, as part of an overall system. Testing may then be applied not to small, single subroutines or functions, but to a stand-alone section of a developing system containing many subprograms. A short survey of system files on a UNIX file system revealed many units of over 200 LOC within code modules of several hundred LOC or higher. Small groups of functions, or modules comprising functions, may be brought together for integration testing. A

test method must have the adaptability to analyse not only the call sequences and parameter passing required in an integration test, but should also be able to focus on a particular, perhaps troublesome, unit within a system.

- Time Constraints. A recognisable problem in any engineering product development is that of time slippage and enforced time constraints on the testing phase. Brooks [8] suggests that testing may originally be planned for 50% of development time but may result in only 10% because of phase slippage. Sommerville [82] suggests that 50% of development time is often consumed by testing. This implies that the technique applied, at least to the unit and integration testing phases, should be adaptable in terms of focusing on troublesome code regions, testing for specific faults or allowing a full test if constraints allowed.

- Large Programs. Some work has been done on the position of faults within code and their effect on execution [54, 79, 85]. Richardson and Thompson's RELAY model of error detection defines origination and transfer conditions that must be satisfied to guarantee detection of an error They analysed six classes of faults; constant and variable reference, variable definition, boolean, arithmetic and relational operator fault. The conditions required to reveal the errors were used to evaluate the test data. The PIE model of Voas attempts to identify locations in a program where faults, if they exist, are more likely to remain undetected during testing. The technique estimates the frequency with which an altered data state will cause a change in the program output. However, academic testing research has been mainly applied to small programs to view the benefits of particular testing strategies. The impact of faults on following code is of concern to output or state comparison techniques such as Mutation Analysis. As Firm MA demonstrates, the effect of an induced or present fault is often determined by what values are output and where. Some faults may be masked by subsequent execution of code and detectable only by a change in state at statement level, other faults may be proliferated throughout the execution. It is doubtful that any general rule may be derived which determines whether a particular fault in a specified construct or position within an execution sequence will be masked or proliferated by the subsequent execution. However, an examination of faults and positioning within the call sequence is still worthwhile to determine whether faults presenting early in the

execution are more likely to be masked than faults presenting near to execution termination.

4.3 Research Aims

The general aim of a new MA experiment is to apply the technique to larger, and possibly more complex, programs and to determine whether MA is viable for more than small unit testing. More specific aims centre on analysing which components are more prone to enable live mutants. It is important to determine whether mutations on conditional expressions generate more live mutants per mutation applied than mutations on non control flow components. This is probably code, complexity and output variable dependent but some general rules may be derived from an experiment.

Previous research has indicated that errors congregate. It is worthwhile to discover whether this is reinforced in an experiment involving induced faults and if so, whether the faults (live mutations) congregate in specific regions such as basic blocks, functions or along particular program paths.

Any information regarding testing of large scale code is useful for directing future tests. Full scale tests are unlikely to be applied to large systems because of the resource constraints. Testers may also feel that a full test is not worthwhile given prior unit or integration testing. Consequently, it is worth analysing single units in the light of induced fault impact on subsequently executed units, i.e. unit testing embedded within a full working system. To this end, it is necessary to experiment with a test strategy on large scale code. MA is useful here because it can simulate other testing strategies. Given time, these techniques can be compared for their error finding abilities on large code.

4.4 A Survey of Common Errors

To determine common problems or factors associated with testing large scale programs, a short, exploratory study was designed and undertaken [25]. The study was based on a publication of common errors in C code [53], and was an attempt to ascertain which errors are more prevalent than others. Due to research time constraints, the study was undertaken over a short period of some three to four months and took the form of a questionnaire. Over forty university researchers, commercial testers and programmers from both environments were interviewed. Questions were asked regarding testing strategies used, errors found and those programmed defensively against. The survey was essentially anecdotal with some inherent bias due to recent sensitisation to particular errors in the interviewee's current developmental or maintenance work. The results were therefore considered as preliminary, giving guidance to problematic constructs. A larger experiment over a range of software, candidates and time is required to eliminate bias of the interviewees to current projects and experience levels and give an improved representation of code errors. As the survey was designed to achieve immediate feedback and to give indications of error commonality this problem was acknowledged and considered acceptable. The survey data was mainly ranked in nature and hence only the basic statistical tests such as frequency, multiple response, cross-tabulation and non-parametric correlation were applicable [27]. A more detailed, long term survey of specific faults and problem conditions would allow more elaborate tests to be undertaken.

The questionnaire was divided into six sections. The first detailed the background of the interviewees; C was not the first language learned and most had a Pascal or Modula-2 background. Over half tested or used code written by others. Output correctness was indicated as the reason for terminating testing by 75% of respondents, but few also used coverage metrics implying that if output was seen to be correct for several test inputs, the code was considered tested. Code coverage metrics were most used by those constrained by time, usually the industrialists and the more experienced interviewees who tended to work on larger projects. This implies that the larger the project, the more necessity for the code to be correct as it is likely to be used by others. However, larger projects tend to have deadlines and suffer from development

phase slippage. The more experienced interviewee tended to oversee several projects at once and were therefore tightly constrained by time.

Assuming time restrictions, the interviewees divided into two groups with a ratio of 3:2 as to whether they tested the most used or the least used units in a system. Experience of particular systems was cited for this division. Over 60% tested units containing what was personally perceived to be troublesome constructs. There was a marked variation in methods for test data generation; researchers tended to choose random data generation, output class coverage (that is checking of all output partitions) was used by staff developing packages and commercial programmers preferred code coverage strategies. These results match expectation. The more commercially viable or robust a system has to be implies the greater usage of a more rigorous testing strategy. Small research groups tend not to expend much effort on testing even if the system developed is a prototype for an important engineering product. The results of the first section indicated that any automated testing tool should have the ability to focus on particular functions indicated by a variety of factors such as most or least frequently used or type of constructs or data items processed.

The next four sections of the questionnaire focused on specific coding faults. Interviewees were asked to rank replies to questions on error frequency between one and four. A rank of one implied the interviewee had never met or known about the possible fault and a rank of two declared they were aware of it. Ranks of three and four indicated that the candidate had seen an associated fault occasionally or persistently.

Some 70% of respondents found the confusion of the tokens '=' (assignment) and '==' (equivalence), a persistent problem in C code. This is a fault not directly modelled by Mutation Analysis. However, mutating either token to members of the same set, assignment operators or relational operators, would make the fault visible. Operator precedence was considered a persistent problem by 40% of the respondents and mostly by the commercial testers. A technique, such as MA, which employs conditional analysis and special values testing on constants and variable references would aid detection of this problem. Some 60% noted wrongly placed statement delimiters, the semi-colon in C. One of the most troublesome areas lay in confusion between array and pointer manipulations. Data flow anomaly and special values

testing may be the most useful techniques to employ for these problems as well as variable reference mutations. Interviewees were asked if they had encountered any of the conditions usually simulated by mutation tools. The use of the wrong identifier had been noticed by 65% of respondents, but 74% had not seen a problem with an incorrect definition of a constant. The use of the wrong relational operator was viewed by 75% of respondents but only 22% had noticed a numeric variable with the wrong sign. Some 56% had encountered a parenthesis fault, that is a failure associated with precedence, and 75% had seen braces placed in the wrong position, terminating a sequence incorrectly.

The questionnaire illustrates that programmers and testers place different emphasis on analysing distinct parts of a program or system. Although a generalisation, many interviewees tested the most used unit or attempted to cover all output classes with test cases. Few chose a metric such as code coverage to indicate test completion and those that did were professional testers. Code coverage strategies, however, do not imply error removal. Interviewees who analysed particular constructs such as semi-colon problems or the use of relational operators did so because they were sensitised to these problems by previous exposure. Few encountered problems with integer overflow or shift operators but few of the interviewees wrote mathematical or low-level code operations. More experienced C users tended to have problems with array handling, pointer chains and general logic. They were also more constrained by time implying that their skills were much in demand. Academic groups voiced fewer problems with the integration of units in comparison with the commercial interviewees. This can be explained by academics and researchers tending to work individually whereas commercial systems, being much larger, require more teamwork to design, code and build a software system.

The results from the survey lead to a conclusion that the experience of a programmer and the task of the code, such as its being an editor or a statistical package, is a factor in the type of errors made. The experience of the tester is also a factor in the detection of code faults. Once a fault is discovered, testers tend to look for more of the same type and also concentrate on the function in which the fault was discovered. The survey indicates some error groupings depending on the programmer's experience and on the program task. Large scale software testing must take these factors into account

by searching for the most common errors to give some measure of test effectiveness.

4.5 The Proposed Strategy

The language C was chosen for the trials for the following reasons:

- It was publically available on the research systems available in both the academic and research laboratories used. A large number of test programs such as system or research code could be made available.
- The research sponsors use C for system development.
- At the time of starting the research, mutation analysis had not been applied to C.

4.5.1 Experimental Mutagens

Once the language had been identified it was necessary to isolate the mutagens that would give indications of test coverage and test data adequacy as well as simulate common code errors. Due to the time constraints on the research, it was also important to choose a small group of mutagens that could easily be applied, that is those which required simple token alterations as opposed to mutagens which required a change to the parse tree such as statement deletion or movement. The set chosen included common fault mutagens discussed in the previous section such as relational and assignment operator and precedence and conditional mutations. Several mutagens were immediately identified as being useful to analyse in a prototype tool and others were added later as experience grew.

The Mothra research group classified some C language mutagens, [4], although Mothra has not yet been adapted to mutate C code. They use a four letter mnemonic for each mutagen. The first letter denotes the type of mutagen; **S** statement, **O** operator, **V** variable and **C** constant. The operator mutagens have the structure **OXYA** or

OXYN where **A** denotes assignment and **N** denotes non-assignment. Depending on the type of the operation the XY positions are replaced by **L** logical, **R** relational, **S** shift manipulation and **E** plain assignment. The variable and constant mutation mnemonics include the letters **G** global, **S** scalar, **A** array, **T** type and **P** pointer. The fourth letter is commonly an **R** denoting replacement.

The following mutagens were developed:

- **Relational Operator.**

  A member of the set $\{<, >, \leq, \geq, ==, != \}$ is replaced, one at a time, by all the other members of the set. Riddell et al [80] stated that if a relational operator transformed to its opposite operator, such as $<$ mutated to $\geq$, remained live, then the four other mutants formed from the rest of the relational operator set would also be live. That is, if a test suite was inadequate for differentiating a relational operator from its complete opposite then it was not likely to differentiate the original relational operator from the remaining relational operators. Woodward later indicated that there may be pathological cases which invalidated this assumption [89] such as operator occurrences within loops. The prototype tool developed for this thesis, was originally designed to mutate relational operator tokens in an ordered fashion, starting with the opposite operator, in order to analyse the live mutation sequences and to halt the generation of all five relational operator mutants if required. The Mothra mutagen equivalent to this operator is ORRN, that is a non-assignment operator mutation of Relational token to Relational token.

- **Arithmetic Operator.**

  The operators $\{+, -, *, /, \%\}$ were each mutated to the other members of the set. This corresponds to the Mothra mutagens OAAN, a non-assignment Arithmetic operator to Arithmetic operator mutation.

- **Assignment Operator.**

  The language C has a more concise form of assignment operations than Pascal or FORTRAN. There are several assignment operators not comparable with other languages such as '+=' which, in the statement 'a += b' dictates that the value of 'b' is added to 'a'. The assignment operators in C, {=,+=,-

=,*=,/=,—=,&=,%=,∧=} were included as a new mutagenic group as distinct from the FORTRAN and Pascal tests. These correspond to the arithmetic, bitwise and plain Mothra assignment mutation operators OAAA, OBAA, OABA, OBBA, OEBA and OEBA. In Mothra an arithmetic assignment operator would be replaced by another arithmetic assignment operator under the mutagen OAAA or by a bitwise assignment operator under OABA. In the test system developed for this thesis all the assignment operators are replaced by each other. This keeps the system simple for both user and development purposes.

- **Increment-Decrement Operator.**

  As with assignment operators, C has a cryptic form of increment and decrement actions on a variable. The statement '$a{+}{+}$' means add 1 to a, if a is an integer, after referencing the variable a. The statement '$+{+}a$' signifies an addition of 1 to the variable prior to its reference. As these operators are commonly used in numeric C programs they were included in the prototype tool. The operator set is $\{+{+}, -{-}\}$ and any reference to one of the set is replaced by post and prefix alternative forms. That is '$a{+}{+}$' would be replaced by '$+{+}a$', '$a--$' and '$-{-}a$'. The equivalent Mothra mutation operators are OPPO and OMMO for postfix and prefix alterations, the final O depicting a unary operation.

- **Logical Operator.**

  To enforce conditional and branch coverage, mutations of logical operators are required. The set in C is $\{\&, \&\&, |, ||, \wedge\}$ which signify bitwise and logical AND and bitwise and logical OR and NOT. The equivalent Mothra operators are OLLN, OBBN, OLBN and OBLN for logical and bitwise non-assignment mutations. The prototype developed simply replaces one logical operator by another in the set. This keeps the user interface and system coding simple without loss of information.

- **Variable Reference Operator.**

  Research [10, 65] has indicated that the variable reference mutations generated the most mutants. In order to analyse how difficult (or easy) these mutants were to kill, all variable references were mutated to in-scope identifiers of the same type. Global and local variables are used to replace identifiers. This is different

from Marick's tests [61] in which only local variables were substituted for identifiers. C allows user defined types which have not been analysed by previous mutation research. The prototype developed alters all variable types including user defined and pointer types. These mutations correspond to the Mothra mutation operators VGSR, VLSR, VGAR, VLAR, VGTR, VLTR, VGPR and VLPR which depict variable global and local replacement by scalars, arrays, user-defined types and pointer types. Each type, in the prototype tool, is replaced by another of the same type so at most two of the Mothra mutation operators would be applicable, the global and the local of the correct variable type. The prototype developed replaces all variables by local and globals of the same type automatically, the user does not need to initiate different classes of variable type mutations.

- **Variable Boundary Operator.**

  Previous research has indicated that analysis of domain boundaries detects faults in code [10]. Numeric variables references were altered by + or - 1 and the *abs* functions was applied and also negated to force domain checking. These mutagens correspond to the Mothra twiddle mutation VTWD and to the domain trap VDTR operation. The twiddle operation indicates a small change in boundary values.

- **Numeric Constants.**

  These can be easily altered to refer to values altered by ε to test boundary values and to the constants 0, 1 and -1 to model coincidental correctness. As constants are used to drive loops or effect conditionals these mutation operators were included in the prototype. This mutagen is similar to the Mothra CRCR operator, which replaces a constant by a constant. In the prototype integer values are altered by + or -1 and real constants are altered by + or - 1.0, + or - 1% and + or - 10% to analyse round-off problems.

- **Unary Operator.**

  These were included because of their simplicity and the effect on code domains. A simple mistake of assigning a value to its absolute or negative value can be simulated by this mutation. For example the statement '$a = 1$' could be altered to '$a = -1$'. This is similar to the Mothra mutation operator VDTR, which

forces data values to be positive, negative and zero by aborting execution on the detection of those values. The prototype uses a simpler form applicable to unary operators in the code.

- **Conditional Alterations.**

  These are a more complex form of logical operator mutation. In a statement 'if $(a\&b)$' the logical operator would be mutated to '&&', '|' and '||' by the logical operator mutagen. However, MA also provides for altering conditionals by the negation of whole and component parts to ensure conditional coverage. The statement could be mutated to 'if $!(a\&b)$', 'if $(!a\&b)$' and 'if $(a\&!b)$'. This mutagen corresponds to the Mothra OLNG mutation operator which negates controlling conditions.

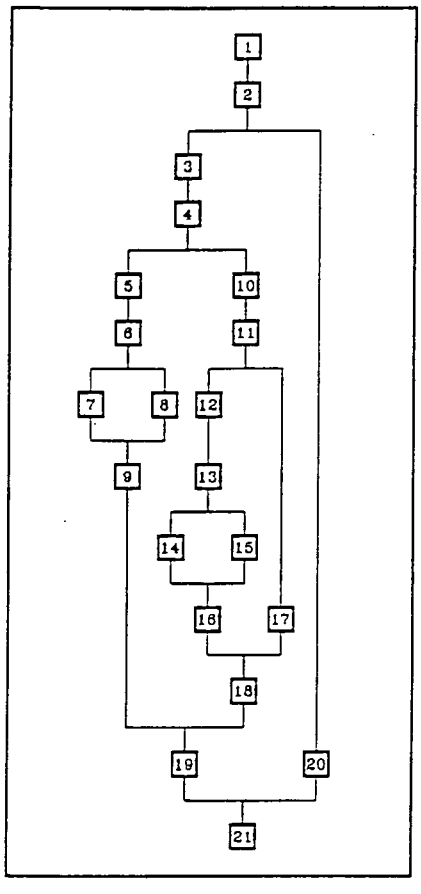
- **Pointer Arithmetic.**

  This was one of the last mutagens to be added. As C is a dynamic language and much use is made of list processing, addition and removal, an operator to simulate errors of access on a list of objects was thought viable and worthy of development. This is a very simple mutagen which adds or subtracts 1 to the reference value of a pointer variable. In C, the addition of unity to a pointer value is immediately translated as one unit of object memory allocation. Thus, if a pointer variable 'p' accesses an object '$*p$', the statement '$p = *p$' could be replaced by '$p = *(p + 1)$'. This simulates a move of the pointer to access the next object in a list, if it exists, creating an error of 'off-by-one'. Similarly, '$p = q$', where 'q' is a pointer variable of the same type as 'p', is mutated to '$p = q + 1$', forcing 'p' to point at the next item in a list. This mutagen does not correspond to any Mothra mutation operator named in [4] but is effectively a twiddle operation on pointer variables.

The above mutagens ensure branch and conditional coverage, boundary error detection, special values and analysis of numeric round-off as well as data flow anomaly testing. They do not include mutagens concerned with statement coverage such as statement removal operators. However, statement coverage can be monitored by placing probes in the code. If the paths executed by test inputs are to be monitored to detect error groupings or problematic code regions, it is cost-effective to use that information to determine statement coverage.

4.5.2 An Application Strategy for Mutagens

Figure 4.1: Linear Code Sequence Control Flow Graph for Trityp Program



The default direction of flow of control is downwards

Current MA tools apply mutagens to tokens as they are found in the source (or intermediate) code. Mutants are thus formed in a linear, or textual, order from mutation components found in the code from the first line down to the last. This strategy takes no account of call sequence or frequency of code usage. If the test is constrained by time, the textual application strategy may fail to analyse important regions of code. Code is effectively analysed by its position in a file and not its effect on execution. If it is important to deliver the most efficient test for the resources available, and this is likely to be true for large systems under development, then it is logical to determine the code regions or components which create the most live mutants in comparison to others. This does not necessarily imply that the regions with most live mutants are more prone to error than ones with only a few live mutants, but it does demonstrate problems with the test data generation techniques used.

One strategy is to find a test route which will generate the best possible test, that is the most live mutants, in as short a time as possible. A more thorough examination can be done, if costs permit, by simply allowing the test to continue through the sequences which generate fewer live mutants for the available test suite. That is, test data must be generated to examine those code regions exhibiting fewer live mutations. The test must be driven at the most efficient rate possible. With MA this means finding the highest rate of live mutants per mutations applied for the given test suite. Although this can not be known in advance for any particular program and its associated test suite, it is important to primarily simulate the common faults in the most crucial routines, however they may be defined. A mechanism is required to direct the mutation application through the source. The code control flow graph is a useful mechanism for this.

A control flow graph is a diagram of the connections between program regions. Each region can be defined as a **Linear Code Sequence**, or basic block. The term linear code sequence is preferred, and used in this thesis, as it indicates the simple flow of control from statement to statement within a code sequence. A linear code sequence (LCS) is defined as a maximal group of statements such that if the first statement is executed then so also is the last. Embedding test case coverage probes at the LCS level of the code gives a better indication of code and path coverage than embedding at the routine level. It is also more efficient than placing probes after every statement. Although monitoring LCS traversal does not directly inform the tester of the number of statements executed, unless a table of the number of statements in each LCS is derived, it clearly reveals missing paths in the control flow graph.

Using the definition of a flow graph from Fenton et al [28], a control flow graph is defined to be a finite digraph G incorporating the distinguished start (source) and stop (sink) nodes. The in-degree of a node is the number of edges entering the node. All nodes, except the source, have an in-degree of one or more. The out-degree of a node is the number of edges leaving the node. All nodes, except the sink, have an out-degree of one or more. The nodes of a control flow graph are program regions and the edges are the flow of control between those regions. The control flow graph can be generated from the LCS connections. The nodes in a test program's control flow graph can be LCSs. A node with an out-degree of two is a predicate node, the out bound

edges corresponding to the True and False control paths and the connected nodes being the LCSs at the start of the relevant code. A node with a higher out-degree is a predicate node depicting a case or switch control statement in which control is passed to one of several nodes. See Figure 4.1 for the LCS control flow graph generated from Ramamoorthy's triangle program (Trityp). The code for this program is in Appendix B.

If LCSs are mapped according to their use sequence and the control flow graph generated, a mutation within a particular sequence can be viewed through its impact on the following code. An LCS higher in the graph, i.e. nearer the root or source, may be executed earlier than one later in the graph, nearer the code termination (sink of the graph). Any component changes within it may have greater consequence than a component change in an LCS close to the sink. Alternatively, a component change close to the sink node may be the more likely to create a live mutant. A back edge, such as is formed by a loop, will complicate this topic. LCSs within loops should be considered positioned by their first use and the loop is not unravelled. Paths may be formed with multiple uses of LCSs which describe the body and condition of a loop.

Traversing the control flow graph, or equivalent tree, for mutation testing, requires nodes (LCSs) to be visited once only. To traverse the control flow graph in a predefined sequence, the data describing the graph had to be manipulated. Standard techniques exist for converting a graph into a tree [51]. However, the data was not reconstructed into a tree representation but was stored in a structured list to duplicate the format of the LCS connectivity file output from the preprocessor, see Chapter 5 for more details.

The cumulative count of live mutants can be plotted against mutations generated. Mutations generated in a higher node of the control flow graph will generate a different live mutants per mutation application graph than mutations initially generated in a lower node, due either to code masking or to error proliferation. Such plots will illustrate where live mutations remain in code and will also indicate problems associated with the impact of code alterations if live mutations are clustered at points, or along paths, in the graph. By driving the mutation application via the LCS control flow graph, the tester can gauge whether faults in the earlier called LCS (or routines

at a higher level of abstraction), critical or otherwise, are more likely to result in live mutants than faults induced in later called LCSs. The tester can also isolate which LCS, or function, exhibits the most live mutations and therefore requires more analysis. A zero kill rate for an LCS may indicate non traversal or inadequate test data. Untraversed LCSs indicate either a problem with the test data or unreachable code. These sequences can be discovered by mapping test case traversal against LCSs.

Using data, control, domain and boundary error as well as special values mutagens, it is possible to determine which group generates the greatest number of live mutant programs per mutations generated. From this information it is possible to determine problematic constructs and LCSs within the code. As sequences and functions are altered during development, the directed MA test lends itself to Revision or Regression testing [24]. Keeping account of which test cases traverse the altered code would allow a reduction in the re-application of test cases. Monitoring LCS coverage during a test should therefore improve the efficiency of a revision test.

An improved test is therefore one in which live mutations are found early in the test sequence. This raises some issues. Any improvement made by driving the test via the call sequence may be program or complexity dependent. It would be necessary to conduct many trials of different programs in order to achieve a valid conclusion. Any improvement may also be test case dependent, so it would be necessary to alter the test case order. Some test cases may be considered good for killing mutations because they have a high statement coverage, others because they are special case selectors. The test must utilise both types of test cases, but the former is possibly better for an initial test in order to remove as many live mutations as early as possible to reduce execution costs.

Another test is required to determine the persistent mutations, that is, the mutation types most likely to remain live. Knowing that most mutants are unstable and die quickly, [2, 10] it is important to simulate faults which require special analysis and test cases. That is, it is necessary to simulate common faults, be they boundary, control flow or data flow groupings. However, these 'primary' mutations may not be applicable to all programs, but to program types determined by size, complexity, coding techniques employed or components used.

There are therefore many factors in applying mutation analysis to large programs given resource restrictions. It is desirable to apply the test to the most critical parts of code and to simulate the more common errors in an initial phase. As these factors may be unknown for any program in advance of a test, it is necessary to start testing and gather information as the test progresses. Once it is known where analysis should be focused, the test should be adapted towards that aim. Using the control flow graph to guide the test is useful for understanding the impact of induced and present faults. In order to use the control flow graph for the application of mutations, the graph must be traversed in some ordered fashion. The common mechanisms for graph, or tree, traversal are Preorder, Inorder and Postorder. A node A with two children B and C, where B is the left-child would be traversed in the order ABC in Preorder traversal, BAC in Inorder traversal and BCA in Postorder traversal. It is worth comparing some of these traversal, and therefore mutagen application, strategies against the standard textual application to find if control flow driven testing is viable and efficient. In the prototype developed the first two traversal strategies were analysed, the third was left for future development. Figure 4.2 shows a small example of a control flow graph, its connectivity file representation and the list structure used to re-create tree traversal. The list for each function was traversed starting at the node representing LCS 0. The traversal sequences are defined recursively as in

```
procedure preorder(n : node);
begin
  mutate (n);
  for each connected node c (left to right) do
      preorder(c)
end;{preorder}

procedure inorder(n : node);
begin
  if n has no connected nodes c then
     mutate(n)
  else
```

```
    begin
      inorder(first connected node c);      /* left child */
      mutate(n);
      for each  other connected child node c of n do
          inorder(c)
    end
end; {inorder}
```

where the connected nodes depict left and right children and are encoded as an ordered list starting from the parent (LCS) node.

4.6 Summary

Mutation testing is commonly applied to small, single units of code of less than 50 LOC. Although an expensive technique, MA has been shown to be useful as a test strategy and as a metric for test completeness. Consequently, a study of MA applied to larger code containing more than one function is worthwhile. However, because of the resource intensive nature of MA, a test conducted in a reasonable time-scale requires the development of application strategies. It is possible that, given time constraints, particular code regions such as functions or groups of linear code sequences or code components such as all conditionals or all data structures may be chosen to undergo more rigorous testing than the bulk of the code.

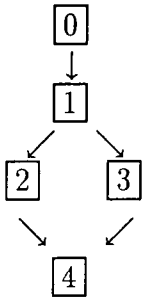
A survey of common errors in C code was undertaken to help indicate problematic constructs or code regions. The diversity of replies in the (anecdotal) survey implied that a testing technique applicable to large scale testing must be able to take several factors into account. These factors may include the experience of the programmer and the task of the written code or in time, the type and positions of faults already found in the code. A tester may always look for, and find, a particular type of error because they are sensitised to it. That is, once an error is found it appears to be worthwhile to look for others of the same type. However, this does not mean that other faults

should not be tested for, but simply that fault finding may be prioritised once some are detected. Similarly, the region in which faults are discovered should be analysed more thoroughly because of error congregation and the likelihood that the code was written by a programmer showing a trend of error creation or a misunderstanding of specifications.

A small group of mutation operators, mutagens, was then outlined. These mutagens are a small group of the possible mutation operators but represent common faults found in code. They simulate data flow, domain and boundary and special values testing. Information regarding problematic code regions such as linear code sequences or functions or error prone code constructs such as conditionals or switch statements can be gathered. A strategy for testing large program by MA was then discussed. The control flow graph of a program and its constituent subprograms could be used to drive a mutation test through code. Using the code control flow graph it is possible to determine whether a fault induced nearer the source node, or start of execution, is more likely to generate a live mutant than a fault induced near the sink node, or code termination. Such information would be useful in large scale testing as it is important to know if faults are masked or proliferated by succeeding processing. Determining untraversed statements will also be simplified by mapping linear code sequences to test cases. This information would also be useful when revision testing is undertaken. Only test cases which traverse altered code need be applied.

Figure 4.2: Example of Data Representations

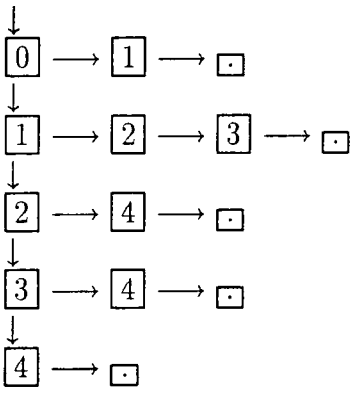
Control Flow Graph



Control Flow Connectivity Data

0 1
1 2 3
2 4
3 4
4

Control Flow List Representation



Chapter 5

The Grail Mutation System

'Mortis Causa'

This chapter outlines the **Grail** mutation tool and its constituent parts. Details regarding coding are not included to retain clarity of design and purpose.

5.1 Introduction

The ethic behind the development of the **Grail** mutation system was to compare textual mutant generation with control flow graph traversal driven mutant generation. Consequently, the constructed system had to compile and execute a test program and its mutants and to determine which of the latter remained live. An ordered list of live mutants found per mutants generated for each code traversal mechanism could then be compared to deduce the efficiencies of each. The **Grail** mutation system is so called because the Oxford English Dictionary defines the grail *'as [an] object of a prolonged quest'*.

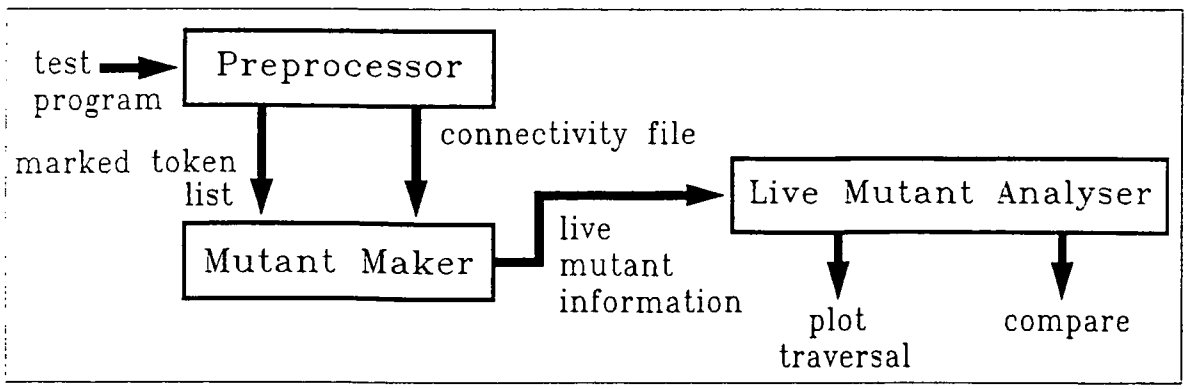
5.2 System Overview

The **Grail** mutation system is composed of three distinct parts, each of which is described in more detail in the following sections. The preprocessor parses the test code and generates a token list complete with codes for mutation component elements, that is, those tokens describing a program component such as a relational operator or a conditional statement. Also output is a file describing the connections between the linear code sequences (LCSs) of the test program and an annotated version of the test code.

The main processing section creates and executes mutants of the test program. It reads in the token list and the connectivity files from the preprocessing stage. The token list is searched for mutation components. Mutant programs are formed, compiled and executed in either a textual order or in an order dictated by processing the data in the connectivity file. The mutant output is compared with the test program output for the given test cases. Live mutants are noted in a table containing a cumulative count against the number of mutants generated.

The third section plots the data from the tables and determines a numeric value to describe the efficiency of each of the traversal mechanisms. See Figure 5.1 for a diagrammatic overview of the Grail mutation system.

Figure 5.1: The Grail Mutation System

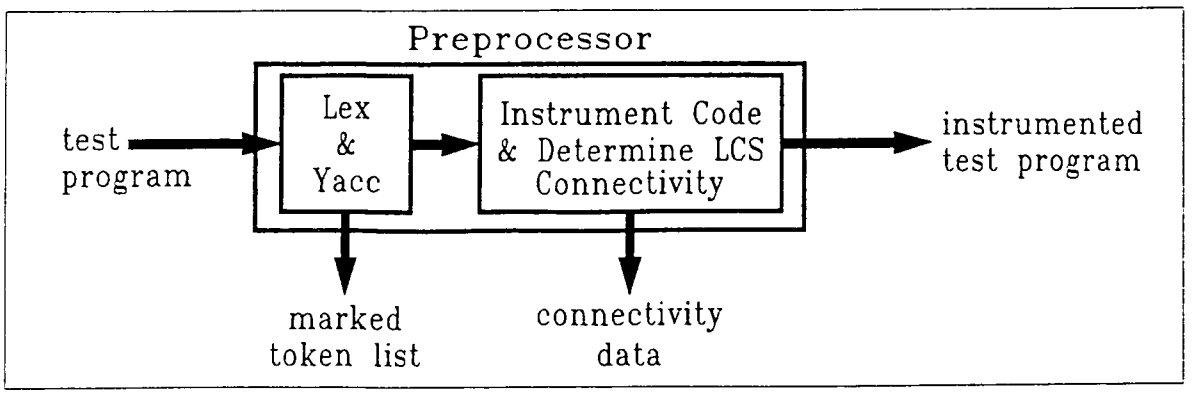


5.3 The Preprocessor

The lexical analyser and parser generators Lex [57] and Yacc [49] are used to decompose a test program allowing specific code components to be noted for mutation purposes. These code components include

- particular operators such as relational, arithmetic, logical and pointer.
- global and local type definitions for gathering variables of the same type for variable reference mutation.
- reserved words such as 'if', 'while' and 'for' to enable mutation of conditional expressions.
- function declaration and formal parameters. Neither of these are mutated and therefore must be marked to avoid confusion with function calls and variable references respectively. Local variables are noted for mutating in-scope identifiers.

Figure 5.2: The Preprocessor



Several files are output from the preprocessor. (See Figure 5.2.) One file contains a list of the test program tokens each with LCS numbers and markers. The markers describe attributes of the token such as whether it is a relational operator, a user defined function call or a system function call. Another output file is the instrumented version of the test program. A probe is placed at the start of each LCS. The instrumented source code can be compiled separately and executed with the test inputs to determine

which LCS are executed by each test input. A tester can then identify which LCSs are unexecuted by the test set and can use this information to develop further test cases. During a regression test the same information can be used to re-apply only the test inputs which traverse altered LCSs.

A third file output from the preprocessor stage is the connectivity data for the test program. This data describes which LCSs flow of control may pass to from any given one. See Figure 5.3 for an example of the format of the connectivity file.

Figure 5.3: Connectivity of Linear Code Sequences for Ramamoorthy's TRITYP

```
1                   /* 1 is the function number. As the program */
1 2 0               /* consists of a single (main) routine      */
2 3 20 0            /* there is only one function.              */
3 4 0               /* The following lines describe the         */
4 5 10 0            /* connections between each linear code     */
5 6 0               /* sequence. LCS 1 connects only to LCS 2.  */
6 7 8 0             /* LCS 6 connects to either LCS 7 or LCS 8. */
7 9 0               /* (LCS 6 is a conditional statement)       */
8 9 0               /* The zeroes at the end of each line aid   */
9 19 0              /* processing.                              */
10 11 0
11 12 17 0
12 13 0
13 14 15 0
14 16 0
15 16 0
16 18 0
17 18 0
18 19 0
19 21 0
20 21 0
21 0
0
```

The code to determine the connectivity between the LCSs was written by another researcher [39]. It cannot handle the 'goto' or the 'continue' constructs and is fragile at high nesting levels. When complex programs were tested by the Grail system, the connectivity file had to be checked and, if necessary, altered by hand.

5.4 The Mutant Maker

This section forms the main body of the **Grail** system. The user is prompted for the following inputs:

- the name of the test program.
- the number of test cases (zero is valid).
- the run-time parameters required, if any.
- choice of yes/no to store data regarding which test cases kill each mutant. If yes is chosen then all test inputs will be applied to each mutant rather than the more efficient mechanism of applying test cases until one kills a mutant.
- choice of which mutagen, or groups of mutagens, to analyse. These simulate relational, arithmetic, assignment, increment-decrement and logical operator, variable reference and boundary, constant, unary, conditional and simple pointer mutations.
- choice of desired code traversal mechanism from Textual, Preorder or Inorder.

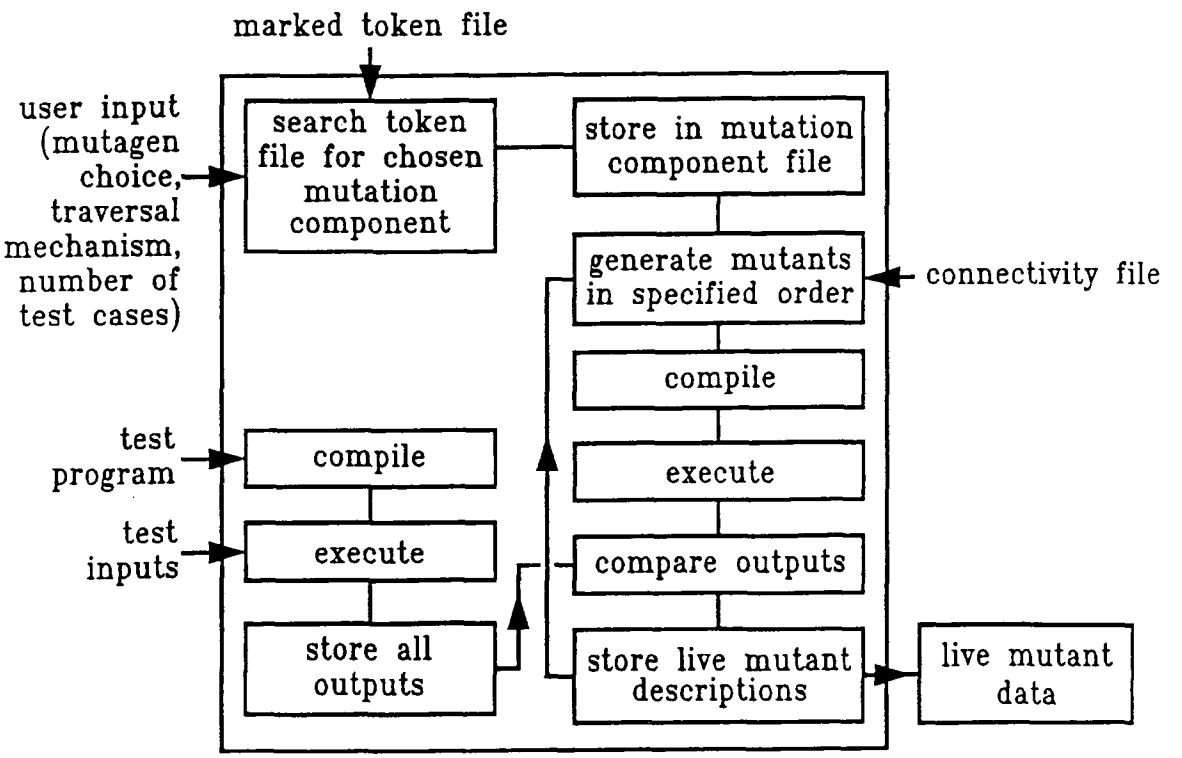
The uninstrumented test program is then automatically compiled and executed on the number of test inputs requested by the user. Each output file is stored under a unique name for later comparison with mutant output. If compilation of the test program fails the system exits after informing the user. However, if compilation and execution of the test program has proceeded without failure, the token list file is read in. The structured list formed from this input is then searched for the tokens required for the mutagen requested by the user. For example, if arithmetic operator mutation has been requested, a search of the token list is made for tokens marked as arithmetic operators. These are copied into a mutation component list which holds a description of the component, which may be more than a single token, including token number and set element offset. The set of arithmetic operators is $\{+, -, /, *, \%\}$. A set element offset of 0 describes the addition operator.

When a component is mutated, the **Grail** system copies the test code up to the component token(s) and those following. The mutation component is replaced by the

other members of the mutation set. In the case of an arithmetic operator there will be four mutants formed for every instance of an arithmetic operator in a test program. The four mutants will be identical programs to the original with the exception of the mutation component.

Once the mutant programs are created they are compiled in turn. Compilation failures are not included in the output statistics. The count of mutants generated includes only compilable program mutants. Each mutant is then executed on the available test cases. As each output file is created it is compared with the output from the original program for the same test case. If the output files differ the mutant is deemed killed and no further executions are required. The next mutant then begins execution with the available test cases. A mutant which is not killed by the available test cases is considered live and a description of the mutation component is stored. This description includes the token number, mutagen type and mutagen set offset number. See Figure 5.4 for a schematic diagram of the Mutant Maker section.

Figure 5.4: The Mutant Maker



When all the mutants of a particular component have been compiled and executed, the next component in the mutation component list is analysed if the Textual traversal mechanism has been chosen. If the control flow traversal mechanisms have been chosen, the Preorder or Inorder mechanisms, the next component mutated is determined after a check on the LCS connectivity and the function call sequence. The LCSs of a function are traversed in an order described by the connectivity data. However, an LCS containing a mutation component may also contain a call to a function. If the function call is executed before the required mutation component, the function is examined for mutation prior to analysis of the component. For example, consider the linear code sequence in Figure 5.5.

Figure 5.5: Function call within Linear code sequence

```
MutateToken()
{
 /* LCS 1 */
 MutateAndCopy();
 MutsDone++;
}
```

If the increment-decrement operator is to be mutated , to '$MutsDone--$', '$++MutsDone$' and '$--MutsDone$', then this must be exercised after the function call to $MutateAndCopy$ has been analysed. If there exist increment-decrement operators operators in the function $MutateAndCopy$, or in any of the functions it calls, then they will be mutated prior to the operator in $MutateToken$. If $MutateAndCopy$ has already been analysed, then this, and future, calls to $MutateAndCopy$ are ignored to ensure that each mutation component is mutated only once. The increment-decrement operator in $MutateToken$ is then mutated. Thus, a large amount of information has to be manipulated; the connectivity of the LCSs, positions of function calls and the marking of functions and tokens already mutated.

The whole sequence of finding mutation components, compilation, execution, storage of live mutant descriptors and marking of LCSs and functions already mutated must be repeated until all the required mutation components have been analysed. The output from each execution of the mutation system is a file describing the cumulative

count of live mutants against mutants generated. See Figure 5.6 for a sample output file.

Figure 5.6: Live Mutant Position file for TRITYP: Relational Operator mutagen on 4 test cases

```
rama      #Funcs  #Lines  #Stats  #Preds  #Loops  #TC
  3       1       37      13      5       0       4

tk-no/ tk-ref/ fn-no/ lcs-no/ #live/ #m_gen/ #groups gen
  64      R5      1       6       1       5       1
  68      R6      1       6       3       10      2
  53      R3      1       4       5       15      3
  57      R4      1       4       6       20      4
  123     R8      1       13      11      25      5
  116     R7      1       11      13      30      6
  42      R1      1       2       13      35      7
  46      R2      1       2       13      40      8
  0

  #Possible Mutants       8
```

The file includes token, function and LCS number (tk-no, fn-no and lcs-no), to describe the positions of the live mutants. Analysis of this file can indicate problem functions and LCSs. That is, those regions with a high number of live mutants. In the example in Figure 5.6, the relational operator described as token 123 in LCS 13 exhibits 5 live mutants. (N.B. the #live column is cumulative.) That is, all its mutants are live after 4 test cases have been applied. The data can be cross-referenced with the LCS to test cases data to determine whether live mutants are caused by non-traversal of linear code sequences or by non rigorous test data. In the example given, the live mutants are due to non-traversal of the LCS containing the mutation component.

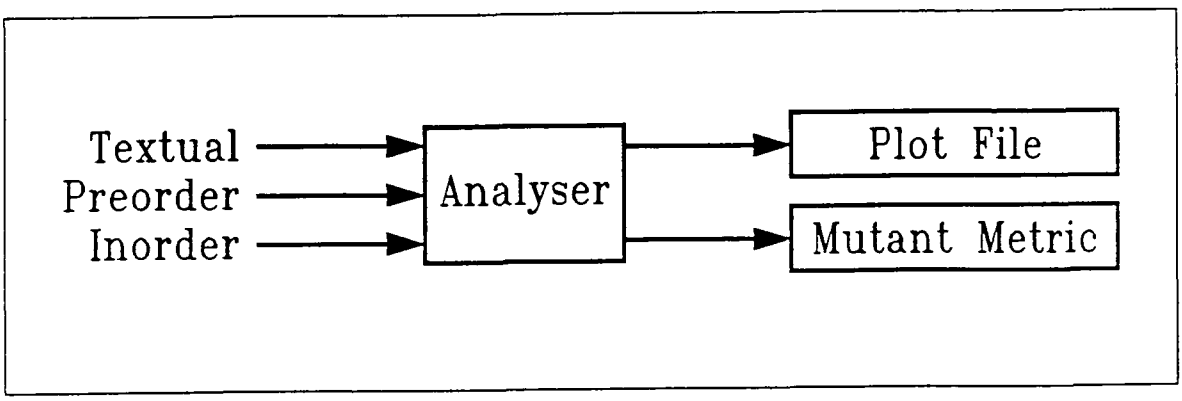
Test cases were applied cumulatively to the mutants. The **Grail** mutation system prompts the user for the number of available test cases. These are stored as the program name followed by a test case number and '*.dat*' as a suffix. If a mutant remained alive after the application of test case 1, it was executed by test case 2 and so on until the mutant was killed or all the available test inputs had been executed.

Thus, when 8 test cases were available, the **Grail** would apply test case 1 through to test case 8 only if the mutant remained alive. The **Grail** system was built to compare control flow and Textual traversal mutation. Thus, for every test case and groups of test cases, the mutation system analysed Textual, Inorder and Preorder traversal mechanisms for each of the eleven mutagens available if they were applicable to the test program. The three files output from each traversal mechanism were stored with unique names for analysis by the next stage in the **Grail** system.

5.5 Live Mutant Analysis

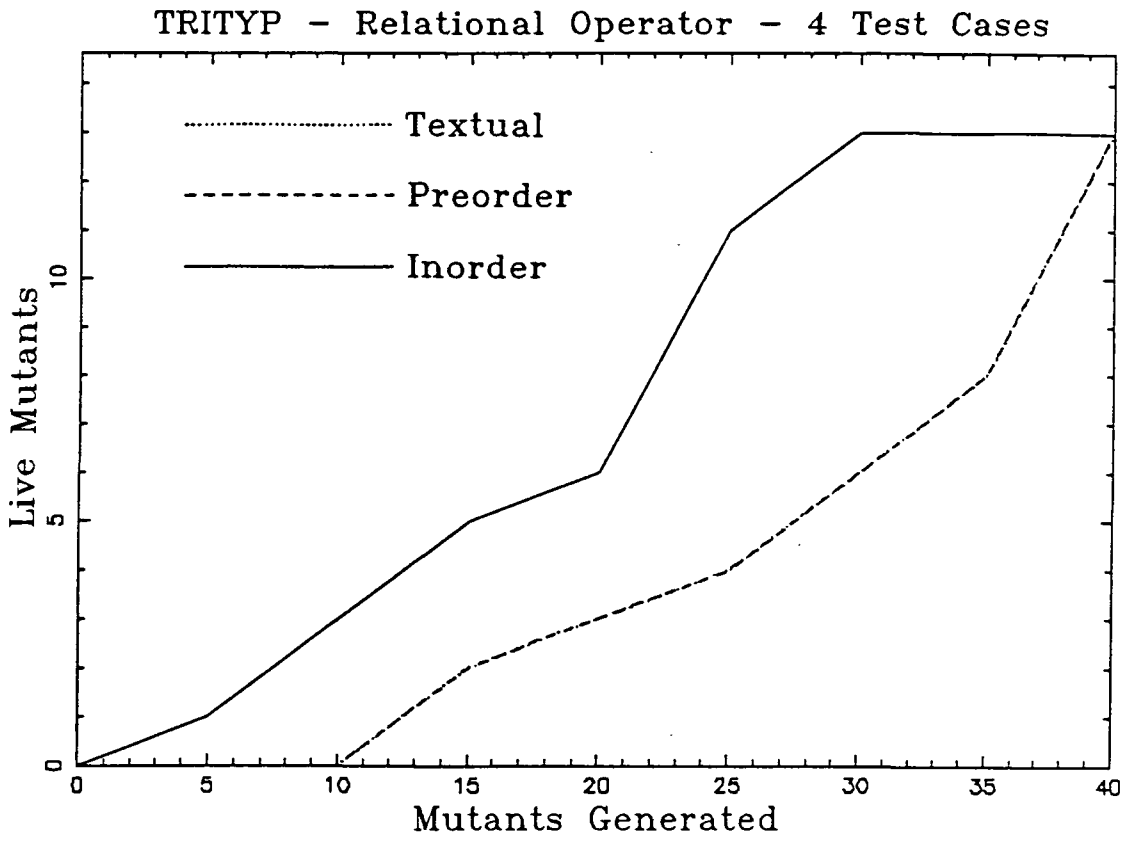
The last stage of the **Grail** system analyses the order of live mutants found by the Textual, Inorder and Preorder code traversal mechanisms. A graph of the live mutants found per mutants generated was made for each traversal mechanism and overlaid on the same plot. See Figure 5.7 for a schematic description of the Live Mutant Analysis section. Figure 5.8 demonstrates the graphs resulting from the test

Figure 5.7: Live Mutant Analysis



of the three traversal mechanisms on Ramamoorthy's TRITYP program using the relational operator mutagen on 4 test cases. The plot demonstrates the differences in locating live mutants between the three techniques. A gradient of 1 implies a live mutant is formed for every mutant generated, i.e. the code is either untraversed by the available test cases or they do not rigorously test the code. All traversal mechanisms should result in an equal number of live mutants, given that all LCSs and functions

Figure 5.8: TRITYP: Relational Operator mutagen on 4 test cases



are callable. However, with time and resource constraints a major factor in testing. it is important to discover the positions of live mutants as early as possible in the test. Thus, the traversal mechanism which generates the greatest gradient for live mutants found against mutants generated in the early stages of testing is considered more efficient than one in which the majority of live mutants are found towards the end of the mutant generation. The **Grail** attempts to isolate which regions of code are more prone to produce live mutants; the LCSs near to the source node or near to the sink or along particular paths. In the example given in Figure 5.8 the Inorder traversal mechanism shows a faster rate of live mutant generation than the Textual or Preorder traversal mechanisms which exhibit the same rate of live mutant discovery.

To compare the three mechanisms on a mathematical basis, a function was derived to describe the efficiency of each traversal mechanism. This Mutation Metric, see Figure 5.9, will result in a higher score for mechanisms which locate the live mutants as early as possible in the test. The scores are normalised by division of the score for the best

possible case, that is, one in which the live mutants are generated prior to all the dead mutants. A Mutation Metric of 1.0 signifies the test was the most efficient possible. A near zero score signifies that the test was inefficient. The weighting, NMAX - N + 1, decreases as the test progresses to ensure that a test which locates live mutants earlier than another, will result in a higher Mutation Metric. The traversal mechanisms can

Figure 5.9: Mutation Metric

$$\frac{\sum_{N=1}^{NMAX}(L[N]*(NMAX-N+1))}{\sum_{N=1}^{LMAX}(N*(NMAX-N+1))+\sum_{N=LMAX+1}^{NMAX}(LMAX*(NMAX-N+1))}$$

N = # mutants generated.
NMAX = total mutants generated.
LMAX = total live mutants possible.
L[N] = #live mutants after N mutants generated.

then be compared on a more scientific basis. In the example given in Figure 5.8, the Mutation Metric for Inorder traversal mutant generation was 0.587. Preorder and Textual traversal mutant generation resulted in a Mutation Metric of 0.231. Thus, the Mutation Metric describes the efficiency of each traversal technique with regard to detection of live mutants. An Inorder traversal resulting in a more efficient mechanism for detecting live mutants in a simple program, such as Ramamoorthy's TRITYP, indicates that most live mutants reside in near-sink LCSs.

5.6 Summary

This chapter discusses the three constituent parts of the **Grail** mutation system. The preprocessor stage separates the program tokens into mutagen types. The test code is also probed to allow analysis of test case traversal of the linear code sequences. The main processing section, the Mutant Maker, searches a token list for mutation components and then generates mutant programs in a specified traversal mechanism order. The mutants are compiled and executed and live mutant descriptors are stored.

The output from the Mutant Maker is a file containing a cumulative count of live mutants found against mutants generated. This data is input to the third stage of the system which plots out the data and generates a numeric, called the Mutation Metric, to describe the efficiency of the traversal mechanism chosen.

Chapter 6

Grail Analysis of Single Function Programs

'Mortui Non Mordent'

Several programs of less than 50 lines of code were analysed by the **Grail** mutation system. Two are discussed in some detail in this chapter. The results demonstrate that there are benefits in efficiency to be gained by using control flow code traversal and mutation. The source code, test data and linear code sequence control flow diagrams for each program are in Appendix B.

6.1 Introduction

Several small programs, of size less than 50 lines of code (LOC), were tested using the **Grail** mutation system. Two of these are discussed in some detail in the following sections. These programs are well known in the testing literature; Ramamoorthy's *Trityp* and Hoare's *Find* [19, 71]. Test data has been published for the *Trityp* and *Find* programs, thus removing any bias in the experiments due to test data anomalies introduced by the author.

Each program was tested on the available test cases with all the applicable mutagens. The test cases were applied cumulatively to the mutants. If a mutant was live after execution with test input 1, it would then be executed with test input 2 and so on until it died or remained live after all the available test inputs had been executed. The applicable test cases were sufficient to kill the vast majority of mutants generated from these mutagens. The test case order was also altered to determine the effect of improving the mutant kill rate on each of the traversal mechanisms. Alarm calls were embedded in each mutant program to abort continual loops if they were formed from a mutation. The alarms were set at 10 seconds CPU time for the smaller programs.

The tables within the text summarise the results from the tests on the two programs. One set of tables display the Mutation Metric for each of the three traversal techniques tested; Textual, Preorder and Inorder. Two columns indicate the gain in efficiency from using a control flow technique. The Preorder Gain (PreGain) is defined as $\frac{Preorder-Textual}{Textual}\%$ where Preorder and Textual refer to the Mutation Metric generated for each technique. The Inorder Gain (InGain) is defined as $\frac{Inorder-Textual}{Textual}\%$ where Inorder refers to the Mutation Metric generated for Inorder traversal and mutation of code components.

The tables include the number of live mutants, #Live, and the Mutation Metric for each of the three traversal mechanisms; Textual (Tex), Preorder (Pre) and Inorder (Ino). The InGain and PreGain columns display the efficiency gain of using Inorder or Preorder traversal and mutation over Textual traversal and mutation. The results column (Res) specifies which of the three mechanisms was the most efficient at detecting live mutants. This can be Tex, Pre or Ino to describe the three traversal techniques, or Equ (Equal) or T/P (Tex and Preorder have the same higher Mutation Metric).

6.2 Ramamoorthy's Trityp

Trityp reads in the three sides of a triangle. The program outputs the type of the given triangle; Equilateral, Isosceles, Acute, Obtuse or Right Angled. The data is

considered invalid if it is not entered in descending numerical order. The program is 37 LOC with five conditional expressions. The following tables show the results of the initial tests on *Trityp* using the input data published in [19]. These include the relational and logical operators and the variable reference and boundary mutations. Three other mutagens were applied to *Trityp*; the Assignment and Arithmetic operators and the Conditional statement mutagen. They generated very few live mutants and were therefore discarded from the analysis.

6.2.1 The Relational Operator Mutagen

Table 6.1: Trityp Results : Relational Operator

Relational Operator Mutagen						
#Input	**#Live**	**Tex**	**Pre**	**Ino**	**InGain%**	**Res**
1	40	1.000	1.000	1.000	0.0	Equ
2	25	0.691	0.691	0.903	30.7	Ino
3	19	0.475	0.475	0.780	64.2	Ino
4	13	0.231	0.231	0.587	154.1	Ino
5	9	0.264	0.264	0.613	132.2	Ino
6	7	0.293	0.293	0.653	122.9	Ino
7	7	0.293	0.293	0.653	122.9	Ino
8	5	0.236	0.236	0.565	139.4	Ino

Trityp has 8 relational operators which generate 40 mutant programs.

Table 6.1 summarises the data generated from tests on the relational operator in *Trityp*. The Preorder gain column was not required for this test as Preorder and Textual exhibited the same Mutation Metric. The results in Table 6.1 show that Inorder code traversal and mutation located the live mutants at a faster rate than either Preorder or Textual traversal, and, as code coverage is increased, higher efficiency gains were to be made. The latter mechanisms have the same results because the linear code sequences (LCSs) containing the relational operators are executed in the same order. The order of LCS traversal is shown in Figure 6.1. The LCSs containing relational operators are numbers 2, 4, 6, 11 and 13. In the Textual and Preorder traversal mechanisms the LCSs are mutated in the sequence 2-4-6-11-13. In the Inorder traversal mechanism they are mutated in the sequence 6-4-13-11-2.

Figure 6.1: Linear Code Sequence Call Graph for Trityp Program

```
Textual Traversal  : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Preorder Traversal : 1 2 3 4 5 6 7 9 19 21 8 10 11 12 13 14 16 18 15 17 20
Inorder Traversal  : 21 19 9 7 6 8 5 4 18 16 14 13 15 12 11 17 10 3 2 20 1
```

Table 6.2 shows the live mutations in relation to the LCSs containing the mutated component. The table helps to explain why the Inorder mechanism located live relational operator mutations earlier in its test sequence than the other mechanisms. The first test input only exercises LCS 2 out of all the LCSs containing relational operators. The mutation metric would be expected to differ and be in favour of the Textual and Preorder traversal because, in these mechanisms, the relational operators in LCS 2 are tested before the relational operators in other LCSs. In the Inorder mechanism, LCS 2 is analysed after all the other mutation LCSs. However, the data in test case 1 generates no dead mutants and is therefore not thoroughly analysing the relational operators in LCS 2. (The reason for this is that test case 1 (2 12 27) includes the condition $b < c$ as well as $a < b$ and so the mutations on either relational remain alive.)

LCS 6 and LCS 13 dominate the rest of the live mutant count. These sequences are tested earlier by the Inorder mechanism and thus dictate that Inorder is the optimal mechanism of the three tested. Test case 3 is the first to traverse LCS 6 and test case 5 is the first to force execution of LCS 13. This indicates that it is important to prioritise or choose test cases which traverse the mutated LCSs.

Table 6.2: Trityp Results : Relational Operator Live Mutants in LCSs

LCS	# Mutants	Test Cases							
		1	2	3	4	5	6	7	8
2	10	10	4	2	0	0	0	0	0
4	10	10	4	3	3	3	3	3	2
6	10	10	10	7	3	3	3	3	2
11	5	5	2	2	2	1	0	0	0
13	5	5	5	5	5	2	1	1	1

Another issue raised by this test is the position of equivalent mutations. There are five equivalent relational operator mutations in LCSs 4, 6 and 13. In a small program, such as *Trityp*, the presence of equivalent mutations will bias the determination of the best mechanism. If the live mutations reside in LCSs close to the source LCS then Textual or Preorder code traversal will be more likely to result in faster live mutant detection than an Inorder traversal. In *Trityp* the bias is towards Inorder because the live mutations are in LCSs primarily executed by that mechanism.

6.2.2 Reordering of Test Cases

The **Grail** system can output a list of the test cases which kill each mutant. Analysis of the list revealed that to kill all the non-equivalent mutations only four test cases were necessary: 3, 4, 5 and 8. Test case 5 killed 18 mutants, test cases 3, 4 and 8 killed 12 each. However, test case 5 does not uniquely kill any mutant whereas test cases 3, 4 and 8 uniquely kill 2, 2 and 1 mutant respectively. Thus, the test cases can be reduced and ordered by their mutant killing ability. A second ordering was created to test if the best traversal mechanism was affected by the killing ability of the test cases. The second ordering applied the test cases in the order 5, 4, 3 and 8. The results showed that Inorder was still the best of the three mechanisms tested. See Table 6.3 for the results of the optimal ordering of test cases to kill relational operator mutants. Relational operator mutations appear unaffected by the optimal ordering of test cases with regard to traversal mechanism. However, there were only a small number of (live) mutants with over 10% of the mutants generated being equivalent.

Table 6.3: Trityp Results : Relational Operator with Optimal Ordering of Test Cases

Relational Operator Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
5	22	0.682	0.682	0.868	27.3	Ino
4	12	0.301	0.301	0.663	120.2	Ino
3	10	0.242	0.242	0.568	134.7	Ino
8	5	0.188	0.188	0.477	153.7	Ino

6.2.3 The Logical Operator Mutagen

Table 6.4 displays the results of testing the logical operators. In *Trityp* there are only three occurrences of logical operators. These are in LCSs 2, 4 and 6. Inorder traversal of the code will result in the LCS mutation sequence 6-4-2.

Table 6.4: Trityp Results : Logical Operator

Logical Operator Mutagen						
#Input	**#Live**	**Tex**	**Pre**	**Ino**	**InGain %**	**Res**
1	12	1.000	1.000	1.000	0.0	Equ
2	11	0.934	0.934	1.000	7.1	Ino
3	6	0.802	0.802	0.441	-45.0	T/P
4	5	0.755	0.755	0.342	-54.7	T/P
5	5	0.755	0.755	0.342	-54.7	T/P
6	5	0.755	0.755	0.342	-54.7	T/P
7	3	0.452	0.452	0.452	0.0	Equ
8	3	0.452	0.452	0.452	0.0	Equ

Trityp has 3 logical operators which generate 12 mutant programs.

Table 6.5 shows the positions of live logical operator mutants in *Trityp*. Test case 1 does not kill any mutants even though it traverses an LCS containing a mutated operator. This is for similar reasons to the mutation of the relational operator in LCS 2 in that one of the two conditional parts is always true with test case 1. Test case 2 (5 4 3) results in Inorder traversal as the best of the three mechanisms in contrast to the other, cumulative, tests. Test case 2 traverses two of the mutated LCSs, numbers 2 and 4. Only one mutant is killed from LCS 2 which means that the best mechanism for finding live mutants is biased towards the mechanism which mutates LCS 4 before LCS 2. Hence Inorder is the best given the small sample of tokens and LCSs mutated. Once test case 3 is executed all LCSs containing logical operators are traversed. At this stage the Textual and Preorder mechanisms show a higher rate of live mutant detection. Five mutations in LCS 4 and 6 are killed by test case 3 thus making LCS 2 the sequence with the most live mutants. As LCS 2 is executed first by the Textual and Preorder mechanisms they generate a higher Mutation Metric.

Table 6.5 shows that there are very few mutants. The determination of the best

Table 6.5: Trityp Results : Logical Operator Live Mutants in LCSs

LCS	#Mutants	Test Cases							
		1	2	3	4	5	6	7	8
2	4	4	3	3	3	3	3	1	1
4	4	4	4	2	1	1	1	1	1
6	4	4	4	1	1	1	1	1	1

mechanism will therefore be biased towards the LCSs containing equivalent mutants, each have one, and towards untraversed LCSs. The test cases which kill the most live mutants are test case 3 and 8 which kill the same mutants and test cases 4 and 7. In the test case application order 3, 7 and 4 the best mechanism is initially the Textual and Preorder on test case 3 before becoming equivalent on the next two test cases. On this small sample of logical operator mutations, the best traversal mechanism fluctuates depending on the killing ability of the test cases.

6.2.4 The Variable Reference Mutagen

There are 34 variable references in *Trityp* to four named variables. Each variable reference is mutated to one of the other three in-scope identifiers thus generating a total of 102 mutants. The results of mutating variable references is displayed in Table 6.6. The LCSs containing variable references are LCS 1, 2, 4, 6, 10, 11 and 13. The Inorder traversal mutates the LCSs in the sequence 6-4-13-11-10-2-1. In some cases the Inorder traversal mechanism is superior to the other mechanisms for detecting live mutants earlier in the test, but there are two occasions where the Textual and Preorder mechanisms are more efficient.

Table 6.7 shows the positions of the live mutants within *Trityp* after the cumulative application of the test cases. Although LCS 1 has 18 possible mutants, none survive the first execution of that sequence by test case 1. Inorder mutation therefore generates a higher Mutation Metric because LCS 1 is the last sequence mutated by that mechanism. The best mutation mechanism tends to be Inorder because of the number of live mutants in LCSs 6 and 13. These LCSs are mutated early in the Inorder mech-

Table 6.6: Trityp Results : Variable Reference

Variable Reference Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
1	84	0.780	0.780	1.000	28.2	Ino
2	36	0.581	0.581	0.858	47.6	Ino
3	29	0.508	0.508	0.767	51.0	Ino
4	25	0.478	0.478	0.672	40.6	Ino
5	21	0.524	0.524	0.620	18.3	Ino
6	17	0.610	0.610	0.571	-6.4	T/P
7	13	0.531	0.531	0.688	29.6	Ino
8	6	0.548	0.548	0.368	-32.8	T/P

Trityp has 34 variable references which generate 102 mutant programs.

Table 6.7: Trityp Results : Variable Reference Live Mutants in LCSs

LCS	#Mutants	Test Cases								
		1	2	3	4	5	6	7	8	9
1	18	0	0	0	0	0	0	0	0	0
2	12	12	8	8	8	8	8	4	4	0
4	12	12	8	4	4	4	4	4	0	0
6	12	12	12	12	9	5	5	5	2	2
10	36	36	0	0	0	0	0	0	0	0
11	6	6	6	2	2	2	0	0	0	0
13	6	6	6	6	6	4	0	0	0	0

anism and, because they are untraversed until test case 3 and 5 are applied, generate a high number of live mutants. It is not until test case 6 is applied that the Textual and Preorder mechanisms are faster at live mutant detection because the live mutants reside in LCSs 2, 4 and 6. These are mutated earlier under Textual and Preorder traversal. Test case 7 kills four mutants in LCS 2 leaving LCS 6 with the highest number of live mutants. Thus, Inorder again becomes the better mechanism for live mutant detection. Test case 8 removes all the mutants in LCS 4 and the majority in LCS 6 leaving LCS 2 with the most mutants. The Textual and Preorder mechanisms again become the better mechanism. One more test case was added by the author to reduce the live mutants to equivalent mutants. The equivalent mutants were in LCS

6 : 'if $a == b$ && $b == c$' could be written as 'if $a == c$ && $b == c$' or 'if $a == b$ && $a == c$'. These results show that the efficiency of variable reference mutation is very susceptible to code coverage. There are, usually, many variable references in a program and a high degree of code coverage is therefore essential to kill the majority of these mutants.

Using the **Grail** system to determine the killing ability of each test input, test case 2 killed the most with 66 dead mutants. The other test cases uniquely killed seven or less mutants. To kill all the non-equivalent mutants all but test case 1 is required. The test cases were applied in the order 2-8-3-6 to determine the best mechanism after optimal ordering based on test case killing ability. The results are shown in Table 6.8.

Table 6.8: Trityp Results : Variable Reference with Optimal Ordering of Test Cases

Variable Reference Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
2	36	0.581	0.581	0.858	47.7	Ino
8	29	0.503	0.503	0.767	52.5	Ino
3	22	0.419	0.419	0.614	46.5	Ino
6	16	0.514	0.514	0.531	3.3	Ino

Thus variable reference mutations are susceptible to the killing ability and the ordering of the test cases, but tend towards an Inorder traversal and mutation mechanism under both a random and an efficient test. The efficiency benefit of an Inorder traversal diminishes as LCS coverage and mutant killing ability increases. This is due to most LCSs containing variable references.

6.2.5 Variable Boundary Mutagen

The variable boundary mutagen works on variable references and alters them by + or - 1 or replaces a reference by the absolute value if it is an integer. Table 6.9 shows the results of applying the variable boundary mutagen to *Trityp*. The table shows

that Inorder code traversal mutation locates live mutants at a faster rate than the other traversal mechanisms tested.

Table 6.9: Trityp Results : Variable Boundary

Variable Reference Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
1	99	0.876	0.876	0.999	14.0	Ino
2	63	0.765	0.765	0.878	14.8	Ino
3	57	0.706	0.706	0.844	19.5	Ino
4	43	0.585	0.585	0.652	11.4	Ino
5	42	0.588	0.588	0.637	8.3	Ino
6	41	0.599	0.599	0.622	3.8	Ino
7	38	0.543	0.543	0.640	17.9	Ino
8	34	0.483	0.483	0.558	15.5	Ino

Trityp has 34 variable references which generate 114 legal mutant programs.

Table 6.10 shows the locations of the live mutants under variable boundary mutation. The sequence of LCS mutation is the same as variable reference mutation: 6-4-13-11-10-2-1 for Inorder traversal. Inorder is the best of the three traversal mechanisms tested because the greatest number of live mutations occur in sequences untraversed until later test cases but mutated earlier under Inorder traversal. For example, the mutants in LCSs 6 and 13 are live until test case 4 and 5 respectively, but are mutated first and third under Inorder traversal. LCS 10 holds a large number of equivalent mutants (8) and would be expected to bias the determination of the best mechanism. However, LCS 10 is mutated in the same sequence position under all traversal mechanisms.

To test whether the benefits of Inorder traversal weaken when the test cases are ordered into their most efficient killing ability, the tests were repeated as before. The test case order was 2-4-3-8. Test case 2 alone killed 51 of the mutants. Table 6.11 shows the results. Inorder traversal and mutation remains the most efficient of the three techniques tested for variable boundary mutation. The efficiency gained diminished as code coverage increased, possibly because of the high number of LCSs containing variables.

Table 6.10: Trityp Results : Variable Boundary Live Mutants in LCSs

LCS	#Mutants	Test Cases							
		1	2	3	4	5	6	7	8
1	18	3	3	3	3	3	3	3	3
2	16	16	14	12	10	10	10	7	7
4	16	16	12	8	8	8	8	8	4
6	16	16	16	16	4	4	4	4	4
10	32	32	8	8	8	8	8	8	8
11	8	8	2	2	2	2	2	2	2
13	8	8	8	8	8	7	6	6	6

Table 6.11: Trityp Results : Variable Boundary with Optimal Ordering of Test Cases

Variable Boundary Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
2	63	0.765	0.765	0.878	14.8	Ino
4	47	0.628	0.628	0.708	12.7	Ino
3	43	0.585	0.585	0.652	11.4	Ino
8	39	0.528	0.528	0.577	9.3	Ino

6.2.6 Trityp Summary

Although the *Trityp* program is small, it is well documented and has been published with test cases. These were applied to the program and all applicable mutagens were analysed. The tests demonstrated that the relational operator, variable reference and variable boundary mutagens benefit from Inorder code traversal and mutation. That is, under Inorder mutation, live mutants are normally found earlier in a test sequence when those mutagens were enlivened. When the test cases were ordered for their maximum killing abilities, Inorder was always the best of the three traversal mechanisms tested. Logical operator mutation showed a variable best traversal mechanism. There are very few logical operators in *Trityp* and the results were heavily biased by code coverage and the presence of equivalent mutations.

6.3 Hoare's Find

Hoare's *Find* program reads a list of integers into an array. The user inputs a pivot position for the array and *Find* sorts the array such that all the values stored in positions lower than the pivot are less than the value stored in the pivot location. All values stored in locations higher than the pivot are larger than the pivot value. Using the data published in [19], *Find* was subjected to the same tests as *Trityp*. The code, test data and the LCS control flow graph are in Appendix B. The applicable mutagens were the relational and assignment operators and the constant, variable reference and boundary replacements.

6.3.1 Relational Operator Mutagen

The results of the tests with the published data is shown in Table 6.12. In contrast to *Trityp* all the relational operator mutagen tests on the published data demonstrate that Textual or Preorder mutation is more efficient than Inorder. However, Table 6.13 shows that there were very few live mutants after the application of test case 1. Of the seven live mutants three were equivalent, in LCSs 2, 7 and 28. The mutants that were killable were in sequences mutated early under Inorder traversal, thus making Textual and Preorder more efficient for locating live mutants.

Table 6.12: Find Results : Relational Operator

Relational Operator Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
1	7	0.390	0.390	0.253	-35.1	T/P
2	7	0.390	0.390	0.253	-35.1	T/P
3	5	0.485	0.485	0.290	-40.2	T/P
4	5	0.485	0.485	0.290	-40.2	T/P
5	5	0.485	0.485	0.290	-40.2	T/P
6	4	0.588	0.588	0.323	-26.5	T/P
7	4	0.588	0.588	0.323	-26.5	T/P

Find has 9 relational operators which generate 45 mutant programs.

Table 6.13: Find Results : Relational Operator Live Mutants in LCSs

LCS	# Mutants	Test Cases						
		1	2	3	4	5	6	7
2	5	1	1	1	1	1	1	1
5	5	1	1	1	1	1	1	1
7	5	1	1	1	1	1	1	1
9	5	0	0	0	0	0	0	0
12	5	0	0	0	0	0	0	0
15	5	1	1	0	0	0	0	0
19	5	1	1	1	1	1	0	0
22	5	1	1	0	0	0	0	0
28	5	1	1	1	1	1	1	1

There are only three of the published test cases necessary to remove all but one non-equivalent mutant. These are test cases 1, 2 and 5. The optimal application of test cases for the most efficient kill rate is as in the original test without the other four test cases. The results are therefore the same as in Table 6.12. This example shows that a small number of live mutants with a high percentage of equivalent mutants will radically affect the efficiency of the mechanism chosen. This is similar to the logical operator tests for *Trityp*.

6.3.2 Assignment Operator Mutagen

Find has 17 assignment operators. The assignment operator set used by the **Grail** mutation system is {=,+=,-=,*=,/=,—=,&=,%=,∧=}. Only three test cases are shown in Tables 6.14 and 6.15 because the number of live mutants did not reduce further with the published test cases.

The mutation of '$m = 1$' and similar statements, to '$m+ = 1$', '$m| = 1$' (bitwise or) and '$m\wedge = 1$' (exclusive or) will be equivalent when m is zero. It cannot be assumed that the values of uninitialised variables will be zero but when they are, there will be a high number of equivalent assignment operator mutations. In the C version of Hoare's *Find* [42] there are 14 equivalent assignment operator mutations if and only if the memory locations used for the variable storage have a stored value of zero. With

Table 6.14: Find Results : Assignment Operator

Assignment Operator Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
1	22	0.523	0.523	0.232	-55.6	T/P
2	16	0.625	0.625	0.100	-84.0	T/P
3	14	0.661	0.661	0.076	-88.5	T/P

Find has 17 assignment operators which generate 136 valid mutant programs.

any other values the result of assignment operator mutations in C is unpredictable. Assignment operator mutation can therefore indicate problems with memory initialisation. However, if memory locations are initialised via assignment statements, then those statements may also be mutated unless the initialisation procedure is deliberately not mutated.

The Inorder sequence traversal for the assignment operators in *Find* is 10-13-16-20-23-24-6-28-2-1. Most of the equivalent mutations reside in LCS 1 which include the initialisation statements. As LCS 1 is mutated first by the Textual and Preorder traversal mechanisms, they result in the more efficient mechanism. LCS 1 is mutated last by the Inorder mechanism. A Textual test on the assignment operators has a minimum gain of 55.6% in efficiency over an Inorder test (see InGain column in Table 6.14).

Therefore assignment operator mutation is greatly affected by the presence of equivalent mutations depending on memory garbage. If these exist, they will bias the results of optimal traversal technique in favour of Textual or Preorder mechanisms. If the equivalent assignment operator mutations were removed from the above results, Inorder mutation would be the more efficient technique due to the presence of live mutations in LCSs 16 and 24.

A test for the optimal ordering of the test cases was not required because the test cases were already in the best order possible for assignment operator mutant assassination.

Table 6.15: Find Results : Assignment Operator Live Mutants in LCSs

LCS	# Mutants	Test Cases		
		1	2	3
1	16	6	6	6
2	8	3	3	3
6	24	2	2	1
10	8	0	0	0
13	8	0	0	0
16	40	6	1	0
20	8	0	0	0
23	8	1	0	0
24	8	4	4	4
28	8	0	0	0

6.3.3 Variable Reference Mutagen

The results for variable reference mutation are shown in Tables 6.16 and 6.17. Only five test cases are shown because the number of live mutants could not be reduced below 10. The results show that Inorder traversal and mutation is more efficient than the other two traversal techniques tested. The Inorder mutation sequence is 3-2-10-9-13-12-16-15-7-20-19-23-22-24-6-5-29-28-27-4-1. Most mutants do not survive the application of the first test case and could be killed by any of the test cases. Out of the 416 mutant programs created only 75 can be killed by less than two test cases and 140 by less than four test cases. This corresponds to Mathur's work [65] which indicates that most variable reference mutations are very unstable and easily killed.

Referring to Table 6.17, the LCSs with the most live mutants are LCSs 20, 22 and 24. These are executed earlier in Inorder mutation than in Textual or Preorder and hence bias the result towards Inorder. The Mutation Metric for each test is low suggesting that none of the techniques is particularly good at locating live mutants. A few of the live mutants are in LCSs close to the sink or the source LCS. The majority are in LCSs distant from either of the extremal nodes of the control flow graph. This suggests that a level-order test may be a better mechanism for locating variable boundary problems. However, even with the low Mutation Metric, an Inorder test has an efficiency gain of at least 115% over Textual (See Table 6.16).

Table 6.16: Find Results : Variable Reference

Variable Reference Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
1	36	0.118	0.118	0.282	138.9	Ino
2	24	0.120	0.120	0.259	115.8	Ino
3	21	0.067	0.067	0.253	277.6	Ino
4	14	0.069	0.069	0.204	195.6	Ino
5	10	0.038	0.038	0.219	476.3	Ino

Find has 61 variable references which generate 416 mutant programs.

Most of the mutants can be seen to be unstable and die after the first test input is executed. Only eight of the 21 LCSs being mutated require any analysis after the first test case. This demonstrates that information regarding where live mutants reside could be used to reduce the cost of testing. By locating the LCSs containing live mutants and therefore considered to be potential fault regions, testing can be directed more efficiently. The **Grail** simply applies test cases until a mutant dies. In an industrial test, it may be that a test is halted and stored and restarted at a later date. When it is known where live mutants reside, either in particular LCSs or in functions, test cases can be developed to exercise those regions which contain live mutants.

The optimal ordering of test cases for variable mutation was unnecessary because the majority of mutants are killed by test case 1. Test case 2 was the next best case killing another 12 unique mutants followed by test case 4 killing an extra 7 mutants. This would have removed the live mutants from LCS 20 but would not have affected the outcome of Inorder being the best of the three traversal techniques.

6.3.4 Variable Boundary Mutagen

The results for variable boundary mutation on *Find* are in Tables 6.18 and 6.19. As in the variable reference mutation, the Inorder traversal mechanism is the most efficient.

Table 6.17: Find Results : Variable Reference Live Mutants in LCSs

LCS	# Mutants	Test Cases				
		1	2	3	4	5
1	48	0	0	0	0	0
2	24	0	0	0	0	0
3	8	0	0	0	0	0
4	8	0	0	0	0	0
5	16	2	1	0	0	0
6	48	1	1	1	1	0
7	16	2	2	0	0	0
9	16	0	0	0	0	0
10	8	0	0	0	0	0
12	16	0	0	0	0	0
13	8	0	0	0	0	0
15	16	0	0	0	0	0
16	64	3	1	1	1	1
19	16	0	0	0	0	0
20	16	7	7	7	0	0
22	16	8	7	7	7	7
23	16	8	0	0	0	0
24	16	5	5	5	5	2
27	8	0	0	0	0	0
28	24	0	0	0	0	0
29	8	0	0	0	0	0

Inorder traversal generates mutations in the sequence 3-2-9-12-16-15-7-20-19-23-22-24-6-5-29-28-4-1. Most of the live (and equivalent) mutants reside in LCSs 7, 16 and 22. Each of these is traversed earlier or in the same position under Inorder sequence mutation and hence the bias towards Inorder. There is not as much variation between the traversal techniques as in variable reference mutation, the largest variation being a 22.6% improvement in efficiency.

The test cases were reordered for their optimal killing ability for variable boundary mutations. Test case 1 was the best mutant killer with 87 dead and test case 6 killed 73. However, only 3 of the 73 were not killed by test case 1. This means that after two test cases have been applied to the mutants, 42 still remain alive, the same as in the original test.

Table 6.18: Find Results : Variable Boundary

Variable Boundary Mutagen						
#Input	#Live	Tex	Pre	Ino	InGain %	Res
1	45	0.499	0.499	0.508	1.8	Ino
2	42	0.464	0.464	0.514	10.8	Ino
3	40	0.467	0.467	0.509	9.0	Ino
4	39	0.475	0.475	0.510	22.6	Ino
5	38	0.464	0.464	0.486	4.7	Ino
6	36	0.466	0.466	0.476	2.1	Ino

Find has 61 variable references which generate 132 valid mutant programs.

6.3.5 Find Mutation Summary

In contrast to the control flow testing of the relational operators in *Trityp*, *Find* relational operators demonstrate that Textual code traversal and mutation locates live mutants more efficiently. However, the relational operator mutants in *Find* are easily killed by the test cases and the results are based on very few live mutants. This implies that as more live mutants are killed the need for traversal techniques other than Textual may diminish. However, the presence of equivalent mutants and live mutation groupings within functions may affect this. The assignment operators in *Find* also demonstrated that Textual traversal was more efficient at locating live mutants. It was noted that a large number of live mutants were formed from the mutation of initialisation statements which may be mutated earlier under Textual traversal. These mutations are live or dead depending on what memory values are in storage. Variable reference and variable boundary mutation both exhibited Inorder traversal and mutation as the more efficient technique. This agreed with the variable mutation results from *Trityp*.

Table 6.19: Find Results : Variable Boundary Live Mutants in LCSs

LCS	# Mutants	Test Cases					
		1	**2**	**3**	**4**	**5**	**6**
1	6	1	1	1	1	1	1
2	4	2	2	2	2	2	2
3	4	1	1	1	1	1	1
4	2	0	0	0	0	0	0
5	8	4	2	2	2	2	2
6	12	3	3	3	3	3	3
7	8	4	4	4	4	4	4
9	8	2	2	2	2	1	1
12	8	2	2	2	2	2	2
15	8	2	2	2	2	2	2
16	24	4	4	4	4	4	4
19	8	4	4	4	4	4	2
20	4	2	2	2	1	1	1
22	8	7	7	5	5	5	5
23	4	2	1	1	1	1	1
24	4	2	2	2	2	2	2
28	8	2	2	2	2	2	2
29	4	1	1	1	1	1	1

6.4 Summary

Trityp exhibited Inorder traversal and mutation as the most efficient of the three techniques at locating live mutants. This applied to all the mutagens tested with the exception of the logical operator which showed different techniques as being efficient depending on the code coverage and the presence of equivalent mutants. *Find* showed that Inorder was the best technique tested for the variable mutations but demonstrated Textual or Preorder was more efficient for the relational and the assignment mutations.

In both programs, when Textual, or Preorder, was the more efficient technique, this occurred when the number of live mutants was low and there were a large number of equivalent mutants. The positions of the equivalent mutations affected the determination of the best traversal technique when the number of mutants was low. Concentrations of live mutants in particular LCSs affected the test results. The

traversal mechanism which analysed an LCS with a large group of test components early in its test sequence was more likely to detect live mutants.

Code coverage was also an important issue. Inorder traversal and mutation was commonly the most efficient technique for detecting live mutants. However, as code coverage increased, the benefits of testing with control flow decreased.

In some cases, there was only a small gain in efficiency in using control flow directed traversal and mutation. This may be dependent on the code size. For a small improvement in efficiency the cost of developing the information required to do control flow mutation analysis should be weighed against the standard technique of Textual mutation. However, even a 5% gain in efficiency in testing a large program may well be worth the cost of control flow testing.

Chapter 7

Grail Analysis of Multi-Function Programs

'Nascentes Morimur'

A further group of larger, multi-function programs were tested. Three are discussed in some detail in this chapter. These tests demonstrate the differences between, and the problems of, testing on a textual basis as opposed to a control flow mechanism.

7.1 Introduction

The first program is taken from Kernighan and Ritchie's standard textbook on C, *The C Programming Language* [50]. The program is not written as such in their book but is given as a series of examples in using pointers and functions. The constructed program, called *Lines* in this thesis, contains 7 functions and over 100 lines of code (LOC). The second program was written by a mathematician and is the code for a new algorithm to solve backtracking problems [77]. Known, in the tests, as *BackT*, the code contains 186 predicates, 28 functions and 1414 LOC. The third test is on seven modules of the **Grail** code, some 1876 LOC. This code, referred to as *Grail*, contains 220 predicates and 35 functions.

7.2 Lines

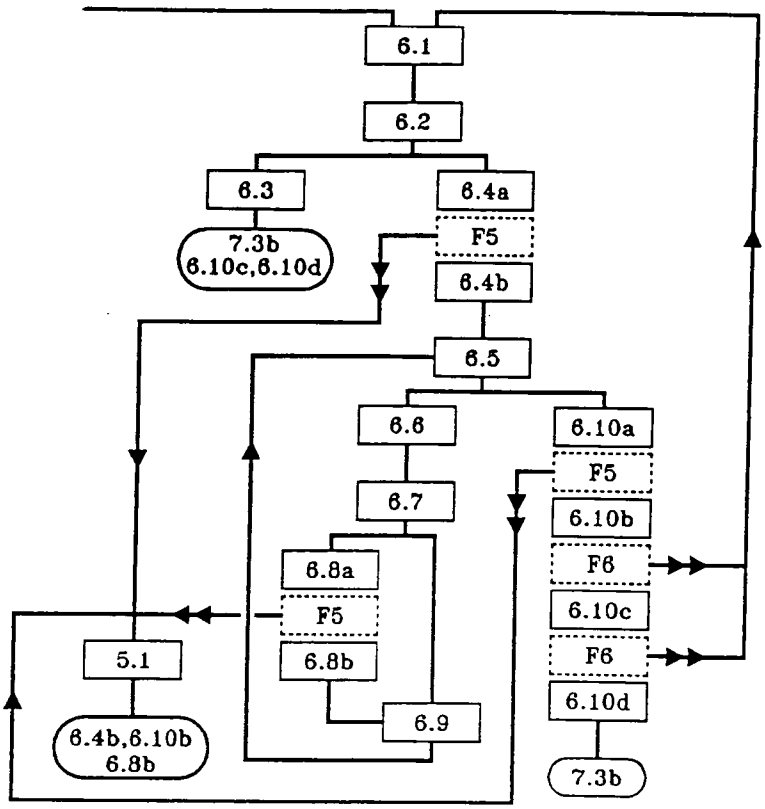
The program reads in lines of characters up to a specific line length, sorts them and prints them out in ascending order of character comparison. See Appendix B for the code, the control flow diagram and the test cases. The code is more complex than either *Trityp* or *Find*. *Lines* contains 7 functions, 117 LOC and 11 conditionals. It generates some interesting comparisons between Textual and control flow based mutation testing. In the control flow diagram, partially reproduced in Figure 7.1, there are LCSs which contain embedded function calls. (The full diagram is in Appendix A.) These are indicated by the dashed lines as in LCS 6.4. When such an LCS is traversed for control flow aided mutation, the code prior to the function call is initially mutated. The function called is then mutated with respect to its control flow before control is passed back to the calling LCS. The remaining code within the calling LCS is then mutated. Functions or LCSs already mutated are marked and bypassed to ensure that components are not mutated more than once. The control flow diagram also shows that a function may return control to one of several LCSs, see LCS 6.3.

Five test cases were manually constructed to kill over 75% of the generated mutants. There were 102 equivalent mutants caused by the variable boundary and the constant replacement mutagens. Another 20 mutants were live under the assignment operator mutation because of zeroes retrieved from memory locations. The number of live mutants varied on different executions of the same test cases because of memory garbage problems. This leaves less than 10% of the mutants as non equivalent live mutants.

7.2.1 Relational Operator Mutagen

Tables 7.1 and 7.2 display the initial results of the relational operator tests on *Lines*. The three columns Tex, Pre and Ino in Table 7.1 display the Mutation Metric for each of the three traversal techniques tested. PreGain and InGain show the efficiency gain in using Preorder or Inorder traversal and mutation over the standard Textual method. The Res column displays the technique with the highest Mutation Metric,

Figure 7.1: Lines : Section of Control Flow Diagram



that is, the one which located live mutants more efficiently than the other two tested mechanisms.

Table 7.1 shows that the Textual traversal and mutation of code is more efficient than the two control flow methods tested. Table 7.2 shows that after the first four test cases had been applied the live mutants were in LCSs which would be mutated earlier under Textual and Preorder than in Inorder traversal. Hence the initial bias towards Textual mutation as the more efficient technique. By test case 5, whichever technique traverses Function 2 LCS 2 first will become the more efficient technique. This corroborates the evidence in the previous chapter. A test of a program with very few live mutations will result in the choice of the technique which traverses the LCSs containing the larger fraction of the remaining live mutations earlier in its mutation sequence. Thus, when testing programs, if any information about the grouping of live mutants is known, a test should be directed primarily towards the functions or LCSs containing the live mutants.

Table 7.1: Lines Results : Relational Operator

Relational Operator Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
1	17	0.554	0.531	0.435	-4.1	-21.5	Tex
2	16	0.570	0.535	0.444	-6.1	-22.1	Tex
3	15	0.566	0.521	0.416	-7.9	-26.5	Tex
4	13	0.509	0.436	0.439	-14.3	-13.8	Tex
5	11	0.363	0.424	0.408	16.8	12.4	Pre

Lines has 13 relational operators which generate 65 mutant programs.

Table 7.2: Lines Results : Relational Operator Live Mutants in LCSs

Funct	LCS	# Mutants	Test Cases				
			1	2	3	4	5
1	2	5	2	2	2	2	0
2	2	15	4	4	4	3	3
2	5	5	1	1	1	0	0
3	2	5	1	1	1	1	1
3	4	5	2	2	1	1	1
3	7	5	2	2	2	2	1
4	2	5	1	1	1	1	1
6	2	5	1	1	1	1	1
6	5	5	0	0	0	0	1
6	7	5	1	1	1	1	1
7	2	5	2	1	1	1	1

The test cases were re-ordered for their maximum killing ability. Test case 5 killed 51 of the 65 mutants and test case 3 killed 17 of which 3 were distinct. These two test cases were sufficient to kill the same mutants that were killed by the original five test cases. The remaining live mutants are not necessarily all equivalent mutants. Test case 5 had the highest coverage of the LCSs, test case 3 one of the lowest. Between them they execute over 90% of the LCSs. Table 7.3 shows the results of testing with these test cases. Preorder traversal and mutation is the best technique with over 17% efficiency gains. Thus, relational operator mutation appears susceptible to the ordering of the test cases in multi-function programs.

Table 7.3: Lines Results : Relational Operator

Relational Operator Mutagen with Optimal Ordering of Test Cases							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
5	14	0.399	0.481	0.406	20.5	1.7	Pre
3	11	0.362	0.424	0.408	17.1	12.7	Pre

Lines has 13 relational operators which generate 65 mutant programs.

7.2.2 The Arithmetic Mutagen

Tables 7.4 and 7.5 show the initial results of testing the arithmetic operators in *Lines*. Inorder traversal and mutation was radically more efficient in detecting live mutants than Preorder or Textual mutation.

Table 7.4: Lines Results : Arithmetic Operator

Arithmetic Operator Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
1	10	0.516	0.516	0.756	0.0	46.5	Ino
4	7	0.323	0.323	0.696	0.0	115.5	Ino
5	6	0.269	0.269	0.706	0.0	162.5	Ino

Lines has 10 arithmetic operators which generate 30 valid mutant programs.

Examination of Table 7.5 reveals why Inorder is the best mechanism for arithmetic operator mutation in *Lines*. The few live mutants that do exist reside in LCSs that are executed earlier in the Inorder sequence. Seven of the live mutants, in test cases 1 through 3, are in Function 6, LCS 4. This LCS is mutated second in the Inorder traversal sequence but fourth in the Textual traversal sequence. The efficiency of Inorder increases as more test cases are applied because all the live mutants reside in the first 2 LCSs mutated under that mechanism. Again, this shows that when the number of live mutants is low, it is important to focus in on the program region, that is, the function or LCS, which contain those live mutants. An efficient test is one in which the code traversal mechanism used can be aimed primarily at the problematic region, that is the code containing live mutants.

Table 7.5: Lines Results : Arithmetic Operator Live Mutants in LCSs

Funct	LCS	# Mutants	Test Cases				
			1	2	3	4	5
1	2	1	0	0	0	0	0
1	3	1	0	0	0	0	0
3	9	4	0	0	0	0	0
6	4	12	7	7	7	4	3
6	10	8	3	3	3	3	3
7	3	4	0	0	0	0	0

The test cases were again reordered to maximise their mutant killing ability. Test cases 4 and 5 were necessary to kill the same number of mutants as all five test cases. Table 7.6 shows the results of using the minimal number of test cases.

Table 7.6: Lines Results : Arithmetic Operator With Optimal Ordering of Test Cases

Arithmetic Operator Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
4	8	0.350	0.350	0.646	0.0	84.6	Ino
5	6	0.269	0.269	0.706	0.0	162.5	Ino

Lines has 10 arithmetic operators which generate 30 valid mutant programs.

Again, Inorder code traversal and mutation is the best technique, but with an increased efficiency. It should be noted that the number of live mutants is low as is the Mutation Metric for all of the traversal techniques.

7.2.3 The Assignment Operator

Tables 7.7 and 7.8 show the results of testing the assignment operator in *Lines*. Only three test cases are shown because test cases 2 and 4 did not kill any extra mutants.

In Table 7.7 the number of live mutants increased when five test cases are applied. Some three mutants which were killed by test case 3 are enlivened after the test is re-run with the five test cases. The problem is associated with values retrieved from

Table 7.7: Lines Results : Assignment Operator

Assignment Operator Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
1	23	0.524	0.468	0.247	-10.7	-52.8	Tex
3	20	0.553	0.450	0.229	-18.6	-58.6	Tex
5	23	0.523	0.467	0.208	-10.7	-60.2	Tex

Lines has 18 assignment operators which generate 90 valid mutant programs.

memory. As mentioned in Section 6.3.2, the assignment operators will be mutated to operators adding, subtracting or effecting logical work on a stored variable. If the variable under mutation is uninitialised then it is possible that any value can be retrieved from memory in the test. Thus, when the test is re-run it is possible to find the number of live mutants increasing. In assignment operator testing in C, a mutant is not necessarily dead when it has been killed by a test case. It may come back to life depending on the values retrieved from memory. These mutants are termed **Zombie** Mutants because they can come back to life at any time.

Table 7.8: Lines Results : Assignment Operator Live Mutants in LCSs

Funct	LCS	# Mutants	Test Cases				
			1	2	3	4	5
1	3	2	0	0	0	0	0
2	1	8	2	2	2	2	2
2	2	8	0	0	0	0	0
2	3	8	0	0	0	0	0
2	6	8	8	8	7	7	8
2	7	8	6	6	5	5	6
3	1	8	2	2	2	2	2
3	2	8	0	0	0	0	0
3	9	8	2	2	2	2	2
6	4	16	0	0	0	0	0
7	2	8	3	3	2	2	3

Textual traversal and mutation was at least 10% more efficient at locating live mutants than the other techniques. This is associated with the memory allocation functions at the start of the program. Function 2, *getline*, has 5 of the 18 assignment operators and

its mutants therefore affect the result. *Getline* is mutated primarily by the Textual traversal method. Therefore, it is important to direct a test of particular operators, or operands, to functions in which clusters of these components exist.

The tests were re-run and the number of live mutants varied between 20 and 23 across all the test cases. The test cases were not applied in descending kill rate ability because test case 1 is sufficient to kill all the non-equivalent mutants, assuming the memory locations accessed by *Lines* had a stored value of zero.

Table 7.9: Lines Results : Variable Reference Operator

Variable Reference Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
1	31	0.333	0.313	0.531	-0.06	59.4	Ino
4	26	0.239	0.222	0.558	-0.07	133.5	Ino

Lines has 84 variable references which generate 155 valid mutant programs.

7.2.4 Variable Reference Mutagen

Tables 7.9 and 7.10 show the results of the initial tests with variable reference mutations. Only two test cases are shown, 1 and 4, because the other test cases did not kill any other unique mutants. Inorder is the most efficient of the three techniques at locating the live mutants. Referring to Table 7.10, this is because the majority of live mutants reside in functions 5 and 6, *swap* and *qsort*. These two functions are mutated at the start of the Inorder traversal mechanism but are mutated towards the end of the mutation sequence under Preorder and Textual traversal.

The test cases were already in the best order for killing mutants. Only test cases 1 and 4 need be applied to kill the same mutants that all five test cases killed. In this case, as code coverage increased the efficiency of Inorder over Textual traversal and mutation increased. This is opposite to *Trityp* but corresponds with the more complex single function program *Find*. *Lines* and *Find* show that in a more complex program and with test cases ordered for maximum killing ability, the efficiency of one traversal and mutate technique over another can increase with LCS coverage.

Table 7.10: Lines Results : Variable Reference Live Mutants in LCSs

Funct	LCS	# Mutants	Test Cases	
			1	4
2	1	2	0	0
2	2	8	0	0
2	3	8	0	0
2	5	8	0	0
2	6	8	3	2
2	7	8	2	0
3	1	8	0	0
3	2	8	0	0
3	4	8	1	0
3	7	8	1	1
3	9	8	0	0
3	10	8	0	0
5	1	16	4	4
6	2	16	0	0
6	4	16	5	5
6	5	16	0	0
6	7	16	3	3
6	8	16	3	2
6	10	16	9	9
7	2	8	0	0
7	3	8	0	0

7.2.5 Variable Boundary Mutagen

Tables 7.11 and 7.12 show the results of testing variable boundaries in *Lines* with the five available test cases. Test case 2 was redundant and did not kill any more than test case 1. Inorder was the least efficient of the three techniques for detecting live mutants and Textual code traversal and mutation was at least 14% more efficient than it. Table 7.12 shows that the majority of live mutants occur in function 1, *alloc*, function 3, *readlines* and function 6, *qsort*. The 18, reducing to 14, live mutants in function 1 must be compared with the initial mutations under the Inorder traversal. The first mutations generated under Inorder are in function 6 LCS 2 and function 5 LCS 1. Inorder locates 9 live mutants in comparison to the 18 found by Textual mutation. (See Appendix C for a plot of live mutant detection rate for variable boundary mutation in *Lines*.) Textual continues to lead the live mutant count until the last five mutant groups are generated. Again this leads to the conclusion that

Table 7.11: Lines Results : Variable Boundary

Variable Boundary Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
1	96	0.770	0.757	0.604	-1.7	-21.5	Tex
3	93	0.753	0.735	0.598	-2.4	-20.6	Tex
4	90	0.738	0.716	0.589	-2.0	-20.1	Tex
5	85	0.688	0.676	0.590	-1.7	-14.2	Tex

Lines has 84 variable references which generate 218 valid mutant programs.

once large clusters of live mutants are detected, a test technique which can focus onto those code regions would be more efficient than one working down through code in some set mechanism.

The test was repeated with the two test cases that killed the majority of mutants. See Table 7.13. Test case 5 killed 118 mutants and test case 3 killed another unique 6. The other test cases only uniquely killed 1 mutant or less.

Under ordered test case application, Textual code traversal and mutation is still the more efficient technique but the gain is less than 1%. This is possibly due to the high number of variable references within most programs and suggests that test cases should be ordered to kill as many variable mutations as early in the test as possible. This would then remove a large number of live mutants and would allow the tester to concentrate on determining the reasons for the existence of the remaining live mutants.

7.2.6 Constant Replacement Mutagen

The results from the constant replacement mutation are shown in Tables 7.14 and 7.15. Only three test cases are shown because the other two did not kill any unique mutants. Textual and Preorder are more efficient than Inorder traversal and mutation on the constant operands of *Lines*. However, the differences between the three techniques are small. The test cases were already in the best order for killing mutants.

Table 7.12: Lines Results : Variable Boundary Live Mutants in LCSs

Funct	LCS	# Mutants	Test Cases			
			1	3	4	5
1	2	10	10	10	10	7
1	3	12	8	8	8	7
2	2	6	3	3	1	1
2	3	4	1	1	1	1
2	5	4	1	1	1	1
2	6	4	4	4	4	4
2	7	8	5	5	5	5
3	2	4	1	1	1	1
3	4	12	7	4	4	4
3	7	4	2	2	2	1
3	9	16	4	4	4	4
3	10	4	1	1	1	1
5	1	20	5	5	5	5
6	2	8	5	5	5	5
6	4	24	13	13	13	13
6	5	12	2	2	2	2
6	7	8	2	2	2	2
6	8	10	2	2	2	2
6	10	36	15	15	14	14
7	2	4	1	1	1	1
7	3	16	4	4	4	4

The slight efficiency advantage gained by the Preorder code traversal and mutation for test case 1 is due to the number of live mutants in functions 3 and 7. As the mutants are killed with the following test cases, Textual code traversal and mutation becomes more efficient at locating live mutants. This is because of the presence of 46%, reducing to 40%, of the live mutants in the first three written functions. Two functions, 3 and 6, accounted for more than half of the constant references. This result again indicates that a test should isolate the component clustering before initiating mutation.

7.2.7 Summary of Lines

The multi-function program *Lines* demonstrates the need for maintaining information about the test program for a full mutation test. Test cases which have a higher

Table 7.13: Lines Results : Variable Boundary With Optimal Ordering of Test Cases

Variable Boundary Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
5	100	0.731	0.730	0.653	-0.001	-0.10	Tex
3	94	0.684	0.673	0.648	-0.02	-0.05	Tex

Lines has 84 variable references which generate 218 valid mutant programs.

Table 7.14: Lines Results : Constant Replacement

Constant Replacement Mutagen							
#Input	#Live	Tex	Pre	Ino	PreGain%	InGain%	Res
1	55	0.794	0.804	0.725	1.2	-8.7	Pre
2	50	0.748	0.739	0.688	-1.2	-8.0	Tex
4	46	0.688	0.656	0.676	-4.6	-1.7	Tex

Lines has 19 constant references which generate 95 valid mutant programs.

code coverage are useful for an initial test in order to remove as many of the live mutants as possible. A second group of test cases should be applied if they traverse the code sequences which contain groupings of live mutants. That is, a database of live mutant position should be held to help direct a test onto the code regions, functions or LCSs displaying those live mutants. Traversing test cases only need be applied, improving the efficiency of the test. Similarly, groupings of components can be detected to pinpoint code regions which are likely to have a high fraction of the generated (live) mutants. In this sense a test can be focused onto code which has particular components or which displays a higher live mutant tendency. The ordering of test cases appears to be useful in the testing of larger, more complex, programs in order to improve the test efficiency.

7.3 A Backtracking Algorithm

BackT.c has 1414 LOC, incorporating 186 predicates and 28 functions. It is the code for a multi-purpose backtracking algorithm and is to be published in [77]. Run

Table 7.15: Lines Results : Constant Reference Live Mutants in LCSs

Funct	LCS	# Mutants	Test Cases		
			1	2	4
1	4	5	5	5	5
2	1	5	1	1	1
2	2	5	5	5	1
3	1	5	1	1	1
3	2	5	3	3	3
3	5	5	5	2	2
3	8	5	5	5	5
3	9	5	1	1	1
4	2	5	1	1	1
6	4	10	3	3	3
6	7	5	3	3	3
6	10	10	5	5	5
7	2	5	5	3	3
7	3	10	7	7	7
7	4	5	5	5	5

time parameters were required when executing *BackT*. The **Grail** input section was adapted to allow run-time parameters to be stored and applied to each mutant. These parameters were not mutated. The run-time parameters dictated which of the programs functions were to be executed. Consequently, several functions of the code were untraversed by the test data. Of the test cases supplied, each of which traversed at least 41% of the code LCSs, two were of interest because they traversed distinct functions. The smaller test case took only 2 seconds of CPU time to execute, the larger took some 26 CPU seconds. The alarm settings within the **Grail** tool were adapted to allow the mutants up to 50 seconds CPU time to execute. Thus, the running time for these experiments was in the order of several hours for each mutagen and for each traversal mechanism. The maximum time for the small test file and parameter list was 11 hours and 41 minutes for the Textual traversal and mutation of the variable references. The larger test file and parameter list took 25 hours and 44 minutes on the same mutagen and traversal mechanism.

As the tests on *BackT* were progressing, it was noted that 7 of the 28 functions were not executed by any of the test files under the parameters given. These were functions that were called when particular estimates were required or when the search tree was minimised. As these were not executed by the test cases, it was thought

that an interesting experiment would be to remove the untraversed functions and run a mutation test on the reduced code, known as *BackT1*. The traversal mechanisms could then be compared on the full program with untraversed functions and on the reduced program with full function coverage. Both programs are therefore functionally equivalent. Full LCS coverage was not achieved with the test cases applied. *BackT* had 623 LCSs and *BackT1* had 370. Each test case traversed more than 260 LCSs. Thus, *BackT1* had an LCS coverage per test case of over 70%. The following sections refer to both the full program, *BackT* and to the reduced program, *BackT1*. The test cases for *BackT* and *BackT1* were not applied cumulatively because of the parameter list requirement and because each test case took several hours to be applied.

7.3.1 The BackT Results

Table 7.16: BackT Results

Muta -gen	# Refs	# Muts	# TC	# Live	Tex	Pre	Ino	Pre-Gain%	In-Gain%	Res
Rel	166	830	1	523	.733	.686	.878	-6.4	19.8	Ino
			2	589	.727	.628	.871	-13.6	19.8	Ino
Arith	112	365	1	263	.818	.800	.858	-2.2	4.9	Ino
			2	289	.852	.844	.868	-0.9	1.9	Ino
Assign	311	2044	1	1172	.768	.715	.860	-6.9	12.0	Ino
			2	1327	.763	.734	.860	-3.8	11.3	Ino
Inc-	87	261	1	158	.728	.744	.881	2.2	21.0	Ino
Dec			2	179	.693	.653	.868	-5.7	25.2	Ino
Logic	27	108	1	77	.887	.843	.887	-4.9	0.0	T/I
			2	78	.878	.845	.889	-3.7	1.2	Ino
Var-	651	2564	1	1432	.557	.468	.793	-16.0	42.4	Ino
Ref			2	1409	.502	.410	.802	-18.3	59.8	Ino
Var-	651	1592	1	1048	.767	.717	.850	-7.8	10.8	Ino
Bnd			2	1031	.740	.688	.851	-7.0	15.0	Ino
Const	259	1293	1	920	.891	.876	.930	-1.7	4.4	Ino
			2	1010	.888	.864	.926	-2.7	4.3	Ino
Cond	116	126	1	80	.564	.342	.766	-39.4	35.8	Ino
			2	76	.517	.329	.733	-36.4	41.8	Ino
Point	35	34	1	29	.976	.873	.931	-10.5	-4.6	Tex
			2	29	.976	.873	.931	-10.5	-4.6	Tex

Table 7.16 shows the results of applying the mutagens on *BackT*. The results were

placed together in one table because of the similarity of the results. The first column of the table refers to the mutagen applied and is followed by the number of references to the operator or operand tested. The column, **#Muts**, refers to the number of valid mutant programs generated and is followed by the number of live mutants after the application of the test cases referred to under the **#TC** column. The remaining columns are as in the previous sections.

Inorder traversal and mutation is consistently the best of the three mechanisms tested. Checking the sequence of functions and LCSs mutated by the three traversal mechanisms, it was discovered that Inorder traversal executed LCSs in functions 11, 13, 16 and 27 at the start of mutation. These functions were all untraversed by the test cases and led to a higher number of live mutants detected earlier by the Inorder mechanism. In the case of the relational operator mutagen, after 100 mutants had been formed in test case 1, Inorder traversal and mutation had resulted in the detection of 58 live mutants. Textual traversal and mutation had detected only 20 live mutants. When 200 mutants had been formed, Inorder had detected 110 live against Textual's 94. When the untraversed functions are removed, as for *BackT1*, Inorder traversal lead to the detection of 50 live mutants for 100 generated in comparison to 20 located under Textual traversal and mutation. Therefore, there is still an efficiency gain in using control flow traversal and mutation. See Appendix C for a sample of the plots of the live mutant detection rate for *BackT* and *BackT1*.

One side effect of testing via control flow, is the detection of dead code, that is, unreachable code. Code can be unreachable due to the logic of conditional expressions leading to LCSs that are impossible to execute. See Figure 7.2 for an example of dead code formed by flaws in the code predicates.

Mutation Testing can not detect these cases except in as much as it would be impossible to generate test data to flow through the dead code in the original program. However, the mutants formed from the predicate alterations may effect execution of dead code. Some dead code can be detected by testing via control flow. Code may be unreachable by the control flow of the program. That is, there may be LCSs which are not reachable from any other LCS. See Figure 7.3 for an example of this type of dead code. Textual traversal and mutation will always generate at least as many

Figure 7.2: Dead Code formed by Ill-Constructed Predicates

```
.
.
 if (num > 10)
  {
 if (num = 10)
   NeverExecuted();
  else
   CanBeExecuted();
  }
.
.
```

The call to NeverExecuted can not be executed because the second conditional expression can never be True.

Figure 7.3: Control Flow Detection of Dead Code

```
.
.
 if (condition)
  return(0)
else
  return(1);
 NeverExecuted();
.
.
```

The call to NeverExecuted can not be executed although analysis of the conditional expressions gives no indication of this.

mutants, both live and dead, as Preorder or Inorder traversal generated code. Textual traversal and mutation will generate more mutants from code which is unreachable under control flow traversal and mutation. Therefore, to benefit fully from control flow driven testing, the LCSs traversed must be checked against the number of LCSs in the test code. Discrepancies, in the form of dead code, can then be detected. The mutants within the dead code can all be assumed to be live to force the tester to deal with the code sequence. The figures given in the tables are the number of live and total mutants from the Textual traversal data files.

Table 7.16 shows that all the traversal mechanisms generated a high Mutation Metric

of over 0.800 when the arithmetic operators were analysed. In the larger program with the more untraversed code, Inorder was the most efficient mechanism for the detection of live mutants. When the test was repeated on *BackT1* with the greater percentage of LCS coverage, the Textual mechanism was more efficient. See Table 7.17 for the tests on *BackT1*. Thus, the detection of arithmetic live mutants appears

Table 7.17: BackT1 Results

Muta -gen	# Refs	# Muts	# TC	# Live	Tex	Pre	Ino	Pre-Gain%	In-Gain%	Res
Rel	110	550	1	243	.666	.593	.736	-10.9	10.5	Ino
			2	224	.546	.501	.729	-8.2	33.5	Ino
Arith	71	231	1	129	.774	.760	.705	-1.8	-8.9	Tex
			2	143	.803	.812	.743	1.1	-7.5	Pre
Assign	191	1252	1	467	.601	.664	.680	10.5	13.1	Ino
			2	535	.550	.670	.671	21.8	22.0	Ino
Inc-	50	150	1	68	.666	.702	.747	5.4	12.1	Ino
Dec			2	80	.572	.596	.707	4.2	23.6	Ino
Logic	22	88	1	57	.901	.842	.827	-6.5	-8.2	Tex
			2	58	.886	.841	.829	-5.1	-6.4	Tex
Var-	560	1600	1	469	.395	.391	.583	-1.0	47.6	Ino
Ref			2	445	.300	.266	.620	-11.3	106.6	Ino
Var-	560	1078	1	534	.767	.717	.847	-6.5	10.4	Ino
Bnd			2	517	.652	.640	.749	-1.8	14.8	Ino
Const	154	768	1	495	.846	.826	.863	-2.4	2.0	Ino
			2	485	.818	.806	.855	-1.5	4.5	Ino
Cond	65	71	1	25	.599	.230	.786	-61.6	31.2	Ino
			2	21	.499	.240	.735	-51.9	47.3	Ino
Pointer	11	17	1	17	.970	.878	.883	-9.5	-8.9	Tex
			2	17	.970	.878	.883	-9.5	-8.9	Tex

dependent on code coverage by the test cases. Analysis of the code reveals that the majority of arithmetic operators occur in five functions. Three of these functions are untraversed by the test data and two of these are mutated early in the Inorder traversal sequence. Therefore the results for the *BackT* tests are biased towards whichever traversal mechanism causes input untraversed code to be mutated early in the test sequence. This will always be the Inorder mutation sequence for *BackT*.

Referring to Tables 7.16 and 7.17, Inorder is the most efficient of the three traversal

mechanisms on all four assignment operator tests. This is different to *Lines* in which the Textual mechanism was the most efficient of the mechanisms because of the concentration of assignment statements in the first three written functions. However, the assignment operators in *BackT* and *BackT1* are not concentrated in a few functions. When the operators, not necessarily assignment operators, are not concentrated in a small number of LCSs or functions, Inorder traversal and mutation appears to the most efficient of the three mechanisms tested. The improvement in using Inorder traversal and mutation increased when testing *BackT1*, suggesting that as coverage increased, the benefits of control flow testing also increased.

Inorder traversal and mutation is again the most efficient of the three mechanisms when the increment-decrement operators were analysed. Inorder had a minimum efficiency gain over Textual mutation of 12%. There are only 87 increment-decrement operators in *BackT*. However, even with this low occurrence, Inorder is the best of the three mechanisms for detecting the live increment-decrement mutants. The operators are not concentrated in a few functions but are used in every function to increment loop driver variables. The efficiency benefit over Textual decreased with higher LCS coverage in *BackT1*. This is again due to the Inorder traversal mechanism dictating the testing of input untraversed LCSs early in its mutation sequence.

Textual is commonly the more efficient mechanism for locating and generating mutants of the logical operator. Only in one trial, on *BackT*, was Inorder the better mechanism with a 1.2% improvement. There are very few logical operators in the two programs, concentrated in one function in particular. Thus, the traversal mechanism which forces mutation of this function, function 20, earlier in its sequence will be the best mechanism. This agrees with the earlier work, in Chapter 6, in which the test cases and mechanisms which traverse the mutated LCSs first, where there are few mutants or they are concentrated in few LCSs, can reverse the trend of all other tests.

Inorder is much more efficient than either Textual or Preorder traversal and mutation when the variable references and boundaries are analysed. Preorder is the least efficient traversal mechanism. This agrees with the results of the tests on the smaller programs in which Inorder was the more efficient mechanism. Variable reference mu-

tation under Inorder traversal on *BackT1* is more efficient than in *BackT*, agreeing with the earlier results from *Lines* and *Find* in which increased code coverage indicated a greater divergence between the techniques. It should also be noted that the Mutation Metric for variable reference mutation is low on all four tests, with the exception of Inorder traversal on the larger *BackT*. The high score for the Inorder traversal test on *BackT* is again due to the traversal of input untraversed functions at the start of the mutation sequence. The remaining lower Mutation Metric scores suggests that another traversal mechanism or test technique may be more efficient in locating live mutants.

The tests on the constants in the two programs resulted in Inorder as the best traversal and mutation mechanism, but with only an improvement of 4.5% at most. Analysing the positions of the constants, it was found that the majority of them were in eight functions, three of which were untraversed. Two of the remaining five functions, numbers 5 and 20, are executed early in the Inorder mechanism and therefore increase the live mutant count for that mechanism. After 100 mutants had been formed, the Textual mechanism had detected 61 live mutants as opposed to the Inorder mechanism's 100. When the untraversed functions were removed, Inorder detected 90 in comparison to the 61 detected by the Textual mechanism. This again shows that it is important to isolate the functions which contain particular constructs or components. A test should be directed towards those functions with a higher concentration of mutation components.

The same argument follows for the mutation of the conditional statements in *BackT* and *BackT1*. The benefit of testing with Inorder traversal and mutation was higher, as much as 47.3%, because the conditional statements are concentrated in a smaller group of functions. The six functions which contain 10 or more conditionals include three which are untraversed by the test data. The remaining three, functions 4, 17 and 26 are executed earlier under the Inorder sequence. The Mutation Metric for all the conditional tests was low, implying that another, as yet untested, traversal mechanism may be more efficient. The conditional tests again demonstrate that it is important to determine the positions of components and to *focus* the test in on the encompassing function.

The results for the simple pointer manipulation show that Textual traversal and mutation is more efficient than the control flow mechanisms. Analysis of the two programs results in the same argument as in the previous paragraphs. There are very few pointer references; only 35 in *BackT* and 11 in *BackT1*. There are only two functions which have more than two pointer references; functions 2 and 14. Function 14 is untraversed, therefore the mechanism which mutates function 2 earlier in its sequence will find the majority of the live mutants. This is the Textual mechanism.

7.3.2 Summary of BackT

The program *BackT* was analysed and it was found that seven of its 28 functions could not be traversed by the available test cases. Two input cases were chosen as representative test inputs and executed on both *BackT* and a reduced, functionally equivalent program, *BackT1*, which had the seven input untraversed functions removed.

Testing with the **Grail** mutation tool revealed that Inorder traversal and mutation was commonly the more efficient mechanism for detecting the live mutants of *BackT*. It was noted that the higher efficiency was due to some of the input untraversed functions being tested earlier in the Inorder mutation sequence than in the Textual or Preorder mutation sequences. Consequently, a higher number of live mutants were detected earlier in an Inorder test leading to a higher Mutation Metric.

The effect of the input untraversed functions on the determination of the best traversal mechanism was removed in the tests on *BackT1*. However, Inorder was again the most efficient mutation sequence mechanism except for the arithmetic, logic and pointer manipulation mutagens. On analysis of the test code, it was discovered that each of these three components were grouped, or concentrated, in a small number of functions. The mechanism which traversed the functions containing the greater concentrations of these mutation components early in its test sequence, was more likely to quickly detect larger numbers of live mutants. This implies that it is important to locate concentrations of mutation components and to *focus* the test onto the functions containing those concentrations. The Mutation Metrics for the variable reference and the conditional expression tests were low in comparison to the remaining mutagens.

It was suggested that another traversal mechanism should be analysed.

Table 7.18: Grail Results

Oper	# Refs	# Muts	# TC	# Live	Tex	Pre	Ino	Pre-Gain%	In-Gain%	Res
Rel	160	800	1	578	.926	.850	.864	-8.2	-6.7	Tex
			2	561	.901	.810	.836	-10.1	-7.2	Tex
			3	446	.831	.726	.739	-12.6	-11.1	Tex
Arith	14	52	1	20	.689	.499	.382	-27.6	-44.6	Tex
			2	20	.612	.467	.467	-23.7	-23.7	Tex
Assign	551	2556	1	1885	.962	.938	.950	-2.5	-1.2	Tex
			2	2175	.969	.948	.956	-2.2	-1.3	Tex
Inc-	54	150	1	126	.898	.847	.860	-5.7	-4.2	Tex
Dec			2	124	.893	.831	.838	-6.9	-6.2	Tex
Logic	34	136	1	125	.975	.945	.945	-3.1	-3.1	Tex
			2	121	.952	.927	.941	-2.6	-1.2	Tex
Var-	709	5027	1	2092	.712	.680	.702	-4.5	-1.4	Tex
Ref			2	2289	.713	.694	.711	-2.7	-0.003	Tex
			3	1391	.557	.519	.501	-6.8	-10.0	Tex
Var-	709	1301	1	1057	.914	.868	.873	-5.0	-4.5	Tex
Bnd			2	1040	.904	.862	.871	-4.6	-3.6	Tex
Const	318	1440	1	1188	.928	.917	.924	-1.2	-0.004	Tex
			2	1168	.916	.922	.920	0.7	0.4	Pre
Unary	33	28	1	27	.953	.967	.999	1.5	5.1	Ino
			2	25	.944	.892	.926	-5.5	-1.9	Tex
			3	13	.954	.998	.878	4.6	-8.0	Pre
Cond	200	249	1	165	.620	.534	.556	-46.6	-10.3	Tex
			2	160	.567	.480	.516	-15.3	-9.0	Tex
Pointer	308	462	1	238	.846	.593	.582	-29.9	-31.2	Tex
			2	218	.788	.484	.466	-38.6	-40.9	Tex
			3	130	.630	.539	.509	-14.4	-19.2	Tex

7.4 The Grail Test

The **Grail** system comprises 11 modules of C source code plus several header files. The total code is some 4978 LOC. The preprocessor section which performs the lexical analysis, locates the mutation tokens and determines the connectivity between the LCSs, is another 4925 LOC.

The **Grail** system could not be tested in its entirety because of time and system constraints; a mutation of a tool which mutates test programs would require strict management. Several of the modules comprising the main **Grail** system were prepared for testing. These modules were placed in a single file as **Grail** cannot, as yet, test multiple module systems. This single file comprised the data entry, display and search and locate mutation components routines. These routines represented all the functions necessarily performed prior to initiating mutation and comprised over 1800 LOC in 35 functions. The code also contained 210 conditional expressions, 57 of which were boolean expressions controlling loops. The test program is referred to as *Grail.*

Two test cases, each with at least 37% LCS coverage were applied to all the mutagens. In some cases three test cases were applied. The third test case had an LCS coverage of 57%. Each test comprising a mutagen and a traversal mechanism took between 14 minutes for the Unary operators and 18 hours 5 minutes for the variable references. The alarm call embedded in each mutant allowed 20 seconds of CPU time to complete processing. The original program took only 8 seconds of CPU time.

7.4.1 The Grail Test Results

Table 7.18 shows the results of the tests on the relational operators in *Grail.* The results show that Textual traversal and mutation was consistently the best of the three tested mechanisms.

7.4.2 Grail Code Layout

Understanding the layout of the code gives insight into the reasons why Textual traversal was consistently the more efficient traversal and mutation mechanism. The *Grail* code is very simple in form. See Appendix B for the list of function names and their associated tasks. The first eleven functions are memory allocating routines for the different dynamic structures used in the code. The next five routines initialise

memory and they are followed by two functions which read in the program data. The next function compiles and executes the test program on the available test inputs and the following two functions deal with the user input. There next follows ten simple routines for checking the class of a token such as whether it is a relational operator or an integer structured variable. The next three functions form the majority of the code; the *view-mutants* function displays each mutation as a token number and offset within its describing set and the *search-prog* routine locates the mutation components in the test program. The *struct-create* routine forms the dynamic structure describing the function and LCS connectivity from the token and function name data files created by the preprocessor. This dynamic structure is traversed when mutations are generated via the control flow. The final routine is the main program which calls the user-input, storage, search and view routines.

The *view-mutants*, *search-prog* and *struct-create* routines are the longest in the code at 182, 761 and 224 LOC each. They, with the *main* routine, are the last four routines in the *Grail* program. The *view-mutants* and *search-prog* routines are also very simple in structure. They are constructed of eleven conditionals, one for each mutagen. In *view-mutants*, each conditional defines a logical block of code which includes a loop to print the list of mutation components found. In *search-prog*, the code reached by executing each conditional includes a switch statement to determine the type or set element member of the component. For example, if a relational operator is located in a function, the function number, LCS number, token number and a description of the operator must be stored in a mutation component list. The description is the offset of the component within the **Grail** operator set. The relational operator set is $\{==, \geq, \leq, !=, <, >\}$. An operator $<$ is described by the offset, 4, within the relational operator set. Consequently, when that component is mutated, the mutants formed do not include the fourth member of the relational operator set. Variable types must also be stored to ensure that variable references are mutated to other variable references of the same type. The list of standard types, such as integer and float, are added to when user-defined types are detected and marked by the parser. The *view-mutants* and *search-prog* routines are important to this discussion because, unless all mutagens are enlivened, the code coverage within each of these routines is very low, around 25% for *view-mutants* and 16 to 54% for *search-prog* depending on whether simple operators or variable references are to be located. The increased code

for variable reference location is due to each reference being checked for whether it is global or local, its type and also whether it is a parameter or a declaration. The latter cases are not mutated and are not added to the mutation component list formed by the *search-prog* routine.

After the first tests were conducted on the *Grail* code it was noted that Textual traversal was the more efficient code traversal and mutation mechanism. A check was then made on the LCS coverage and it was noted that for any individual mutation, several functions would not be traversed and, as mentioned previously, the two largest functions would have very low LCS coverage. This large number of untraversed LCSs and functions affect the Mutation Metric for each of the traversal mechanisms. The mechanism which mutates the input-untraversed LCSs early in its sequence will locate live mutants faster and consequently achieve a higher Mutation Metric.

A further test was then constructed. Only the mutations on input traversed LCSs were considered, effectively creating a path execution mutation test. The **Grail** tool could not support this test as it is written, so the original output from it was adapted manually to show the results of only mutating along the execution path. A copy of the original output was taken, see Appendix A for an example, and the untraversed LCS lines deleted. A simple program was written to construct the execution path results by comparing the LCSs in the execution path file with the LCSs in the original file. The correct number of live mutants and mutants generated were then determined for the path file. The results of analysing along the execution path are shown in Table 7.19.

7.4.3 Grail Results Explanation

The relational operator mutagen shows that the Textual traversal mechanism is more efficient than either of the other tested mechanisms. The efficiency gain is between 7.2% and 18.4% on Inorder traversal on the same test input. The lower gain was achieved on the full *Grail* test and the higher was on the execution path test, known as *PGrail*. The relational operators used in *Grail* are concentrated in five functions which read in the data and display and search the data for mutation components.

The mutations in the display and search routines are more likely to fail as they deal directly with output and the creation of the mutation component list. Errors in this list would create dead mutants as the list is partially printed to show the distribution of mutants. Thus, it is the relational operators in the data reading and storage manipulation routines that are more likely to remain alive. These routines are typed at the start of the code and consequently Textual traversal locates the live mutants earlier in its mutation sequence. The efficiency benefit of Textual traversal and mutation is increased when the execution path mutations are considered. This is again due to a concentration of live mutants in the initial memory allocating routines. It should be noted, however, that the Mutation Metric for the traversal techniques is reduced when considering the execution path only mutations. This suggests that there may be a more efficient mechanism for locating live mutants than the mechanisms tested.

There are very few arithmetic operators in the *Grail* code. All but three of the 14 references are in function number 32, two of the remaining three references are in function 34. Function 32 is mutated later than function 34 in the Preorder and Inorder mutation sequences and consequently the Textual traversal mechanism detects the majority of the live mutants earlier in the test. The execution path tests slightly increased the benefit of testing with Textual traversal and the Mutation Metric was again reduced in all test cases.

A similar argument follows for the few logical operators in the test code. The ones which die occur in the data structure creation routine which is function 34. This is mutated early in the control flow testing sequences and thus gives a lower live mutant count in the lower quartile ranges than the Textual traversal mechanism. The greater majority of the logical operator mutants remain live showing that the test cases are not analysing the operators thoroughly, or that the majority of them may be redundant due to robust coding techniques.

The increment-decrement operators occur throughout the code. The mutations which die are in the storage location routines, as they affect the number of structures formed to hold the component list, and the initial user input routines where they are used to perform validity checks on the data. These routines are mutated earlier under

the control flow traversal mechanisms and result in lower live mutant counts in the first and second quartile. Consequently, the live mutants of the increment-decrement operator are found earlier in the Textual traversal test sequence.

The assignment operators occur throughout the code and the Mutation Metrics for the full test are very high, each mechanism resulting in a metric score of over 0.9. There is, however, only a slight efficiency advantage to be gained by using the Textual traversal mechanism. This is due to the grouping of assignment operators in the functions which set up the component lists and initialise memory. Another concentration occurs in the functions which test for the set element position of a mutation component. In any one test, the majority of these latter functions are untraversed and as they occur in the written code before the larger search and view routines, they result in a higher score for the Textual traversal mechanism. Once the untraversed functions are removed for the execution path test, the Mutation Metric scores again reduce and the efficiency benefit increases. The remaining concentrations for live assignment operator mutations are in the storage allocation and initialisation routines. As in the tests on *Lines* the fact that the assignment operators are grouped in functions at the start of the code sway the result in favour of Textual traversal. The difference between the Mutation Metrics for each traversal mechanism is low because in the *Grail* code there are 551 assignment operator references and over 140 in the execution path only tests and the references occur throughout the code.

The variable reference mutations on the full *Grail* code show little difference between the three traversal techniques. This is again due to the large number of variable references throughout the code. However, the Mutation Metric for Textual traversal is the highest at 0.712 and 0.713 which indicates that a better mechanism may not have been analysed. A large percentage of the variable references occur in functions 33 and 34, the *search-prog* and the *struct-create* functions. As the greater part of the *search-prog* routine is untraversed on all the test cases, the majority of live mutants, some 80%, are resident in this one function. *Search-prog* is mutated earlier in the Textual traversal sequence than in the control flow mechanisms on the full code test. As *search-prog* is executed it calls other functions to detect the mutation component type and then to add information to the mutation component list. Therefore, under control flow guided traversal and mutation, this large function is partially mutated

before a called routine is mutated. The large number of variable references and therefore live mutants, are then distributed over the whole test. Under execution path mutation the effect of this distribution is seen as an increased efficiency of the Textual traversal. This test therefore shows the necessity of focusing a test on the functions which display concentrations of particular components. Test case 3 showed Preorder to be the most efficient traversal mechanism for the execution path test. This test case incorporated the search for variable references and caused execution of a large part of the code in the *search-prog* function. Consequently a large number of variable references were analysed, (See Table 7.19). A smaller percentage of mutants remain alive in both the full execution and the path only test. The part of the *search-prog* routine which locates variable references is executed early in the Preorder execution path only test and consequently Preorder is the most efficient mechanism for detecting live mutants.

The results again demonstrate that testing a large program via Textual or control flow mechanisms will always be affected by concentrations of test components in functions.

The variable boundary mutations on the full test program again show Textual to be the more efficient traversal and mutation mechanism. This is again due to the number of variables in the untraversed functions which test the type and offset of the mutation components and in the search routine. The execution path results show a slight efficiency benefit in using the control flow traversal and mutation techniques. This is due to over 30% of the live mutants occurring in function 34, which is the last function tested under Textual variable mutation, but is called and therefore mutated earlier in the control flow traversal test sequences. This result agrees with the variable reference mutation in that it is important to isolate concentrations of mutation components.

The constant mutation tests on the *Grail* code display the only occurrence of a more efficient Preorder test on the full program. However, in the full code test the efficiency improvement of Preorder, or Textual in test case 1, is very small. Constants are used throughout the *Grail* code and each traversal mechanism displays a high Mutation Metric. Under the execution path tests, the effect of the untraversed LCSs containing constants are removed and the best mechanism is the one which traverses the

largest concentration of constants. This is in the *view-mutants* and the *search-prog* routines. The *view-mutants* is tested last under control flow aided mutation and the first function tested, the user input function *start-up*, displays fewer live mutants. This routine also incorporates initialisation of the mutagen count variables, all set to zero, and displays a menu of mutagen types with constants for the user to choose from. For example, the line '*RelationalOperator* – 1' is printed to inform the user that an input of 1 enlivens the relational operator mutagen. (See Appendix A for a sample run of the **Grail** system.) Any alteration of these displayed constants results in a dead mutant. An alteration of a constant used for initialisation of a summation variable will also result in a dead mutant when fewer or greater mutation components are displayed under the *view-mutants* routine.

There are only 33 references to unary operators in the *Grail* code, and less than 50% of those are executed under any one test case. As such the results are very dependent on one or two mutations. The live mutant count under Textual traversal lags the Inorder live mutant count by one after a particular mutation, in function 22, dies. Thus, whichever technique traversed function 22 first in its mutation sequence will result in that technique being the least efficient of the three techniques tested. When the effects of the input untraversed live mutants were removed, the control flow techniques became more efficient at detecting live mutants. In all the tests, the Mutation Metric was high as the majority of the mutations remained live under all the test inputs. As the majority of mutants were live, and there were so few mutants, the results varied depending on the results of very few mutations.

The Mutation Metric for each of the tests on conditional expressions was low, the highest score being 0.62 on a Textual test. Function 33, *search-prog* again holds a large number of conditional expressions, generating over 50% of the created mutants and over 65% of the live mutants. However, Textual is the more efficient mechanism for locating live mutants because of a group of mutants in the token check routines, function numbers 22 to 29. These functions are called from the *search-prog* routine which means that the control flow driven traversal techniques locate the live mutants later in their test sequence. The first few functions mutated by the Preorder and Inorder techniques generate very few live mutants, indicating that conditional expression mutations are more likely to die if they are executed early in the code. That

is, conditional expression mutations in near source LCSs are more likely to die than those in near sink LCSs. The execution path test showed a larger difference between the traversal mechanisms tested in all but one case. As before, the Mutation Metrics decreased showing that another traversal or test mechanism may be more efficient than the ones analysed.

The simple pointer arithmetic tests revealed Textual traversal and mutation as the most efficient test mechanism on the full test, but Preorder on the path execution tests. Appendix C includes a plot of the live mutant detection rate for the pointer tests. The plot for test case 2 shows that the Textual traversal mechanism lags in the detection of live mutants over the first quartile of mutants generated. This is because the Textual test mutates function 12 first which only has two live mutants out of 18. This is function *fnames-fnos-read* which reads in the names of functions and a number describing their written order. This data was generated from the parser and was used to help identify and check for uses of function names in the token data. Any pointer fault here, when the structure to hold the function names is being initialised, could result in a function name being over-written. This will be obvious on the output as the *Grail* system prints out all the function names it locates. The Preorder and Inorder traversal sequences mutate the functions dealing with the data storage allocation and these generate more live mutants than the data read functions.

There are very few live mutants in the last function tested by the Textual traversal mechanism, the *struct-create* function. This function creates the structured list of tokens and their descriptions, such as mutagen operator type, from the token data file created by the preprocessor. Errors in the structure formed from this data will again be obvious on the output. If data elements are not added or are overwritten by faults in the pointer manipulations, some mutation components will be lost. The *Grail* prints out the LCSs traversed as it generates mutants. Consequently, most mutations in the storage and traversal of the LCSs and functions do not survive. If the trace of LCSs analysed were removed, the number of live mutants would fall. The Textual traversal sequence mutates function 33, the *view-mutants* routine before the other sequences. This function has a lot of live mutants because of the high percentage of untraversed LCSs. Hence, Textual is the more efficient mechanism overall for the detection of live pointer arithmetic mutants. The execution path tests

on the pointer manipulations showed an increased Mutation Metric and, in test cases 1 and 3, Preorder is the more efficient traversal mechanism as they detect the live mutants from the memory allocation and initialisation routines. Test case 2 also showed higher Mutation Metrics on the execution path tests. The difference between the Textual and the Preorder test is negligible and is caused by a higher live mutant detection rate in the first quartile by the Textual mechanism. This is again due to live mutants occurring in storage allocation routines.

The pointer manipulation tests show that it is important to test memory allocation and initialisation routines in C. Faults in the traversal of dynamic structures may be more easily detected but this also depends on the amount of data output. Copious output data is more likely to cause mutants to die than output statements generated when, for example, a particular search token has been located. If a large structure is being searched for a few items of information, faults in the traversal mechanism will generate a large number of live mutants. Consequently, when a large data structure is being created and traversed, it would be advisable to have numerous output statements, effectively forming a trace, for mutation purposes. This is effectively a form of Firm Mutation Analysis when the program state is analysed at logical points in the code.

7.5 Summary of Grail Test

The mutation of part of the *Grail* code under the **Grail** mutation system has given more insight into the testing of large programs. The initial results showed that the Textual traversal and mutation mechanism was more efficient at detecting live mutants in code. This is in contrast to the tests on *Lines* and *BackT*. However, an examination of the code and the test cases revealed some of the reasons why the control flow tests were not as efficient as might have been expected.

The tested section of *Grail* code is made up of 35 functions, the largest three being placed at the end of the source code file. This would initially appear to make control flow testing more efficient than a Textual test. However, these functions are large

because they are constructed of code dealing with each different mutagen type. Unless more than one mutagen is enlivened by the user of **Grail** in any one execution, only a small percentage of the LCSs in those functions will be executed. Thus, the control flow mechanisms partially mutate each of those functions before control is passed to one of the smaller routines dealing with memory initialisation or mutation component analysis. The two larger functions, *search-prog* and *view-mutants* are also executed towards the end of execution. The Textual traversal mechanism must mutate the functions dealing with memory allocation and initialisation and data input before it mutates the larger functions. These earlier typed routines do generate a large number of live mutants. The main benefit of Textual traversal is derived from the mutation of the larger functions. Textual traversal and mutation is applied to each logical block dealing with the mutagen types enlivened by the user, or test input. As the test cases used only enliven one mutagen in any execution, the Textual traversal mechanism mutates large logical blocks of input untraversed code in sequence before either of the control flow mechanisms.

A secondary test was then performed on the live mutant data generated from the *Grail* test. The live mutants from the input traversed LCSs were analysed to determine if Textual traversal was still more efficient than the other tested mechanisms when generating mutants along the execution path only. The tests on the execution path of *Grail* resulted in Textual traversal again being the more efficient mutation mechanism. The cases where Textual was not the most efficient mechanism included the variable reference and boundary, unary and pointer manipulation experiments. The variable boundaries were due to concentrations of live variable boundary mutations in the function which created the data structure. The test case which resulted in Preorder as the best mechanism for detecting variable reference mutations was the only one which traversed a large section of code which included a concentration of variable references. The unary operators were very few in number and as such the best mechanism depended on the order of traversal of one LCS. The pointer result was due to the high number of live pointer mutants occurring in data allocation and initialisation routines which were executed earlier under Preorder traversal.

The *Grail* tests therefore agreed with the tests on the other multi-function programs tested. Large logical blocks of input untraversed code will affect which traversal

mechanism is the more efficient at detecting live mutants. As coverage is increased, the Mutation Metric is commonly reduced and the divergence between the three techniques increased. That is, control flow testing as opposed to Textual traversal testing does result in a different rate of live mutant detection. However, in some tests, the arithmetic and conditional tests, the Mutation Metric was less than 0.5. This suggested that the most efficient traversal or location mechanism may have not been tested in these experiments.

7.6 Summary

The results for the tests on larger, multi-function programs were discussed in this chapter. The first program, *Lines*, contained 117 LOC, 7 functions and 11 conditionals. The test results showed that test inputs with a higher code coverage should be used in the initial stages of testing to remove as many live mutants as possible. When high coverage test inputs, or test inputs with a high mutant killing ability, were applied early in the test, one of the control flow traversal and mutation techniques was commonly the more efficient at detecting live mutants. In the tests where Textual traversal and mutation was the most efficient mechanism, the assignment and the constant mutagen tests, it was noted that these components tended to be concentrated in only a few functions. Consequently, the traversal mechanism which mutated those functions earlier in its test sequence than the other mechanisms, is more likely to detect groups of live mutants. It was proposed that a test mechanism which detects groupings of mutation components and mutated those functions or LCSs early in a test sequence would provide a more efficient test. That is, a test mechanism is required which *focuses* mutation testing onto particular functions.

The next two programs were each over 1400 LOC. *BackT* had 1414 LOC with 28 functions and 186 predicates, while the section of the **Grail** tool tested had 1876 LOC with 35 functions and 210 predicate expressions.

The tests on the *BackT* program showed Inorder traversal and mutation as the most efficient technique. Seven of the 28 functions were discovered to be impossible to exe-

cute with the available test cases and were removed to form a functionally equivalent, but much smaller program. The results for the tests on the smaller program, called *BackT1*, showed that Inorder was commonly the most efficient traversal and mutation mechanism. However, the arithmetic, logical and pointer mutagen tests resulted in Textual traversal having the highest Mutation Metric. Each of these three mutation components were grouped in a small number of functions. The mechanism which traversed, and hence mutated, the larger groups of these components early in its test sequences would result in the higher Mutation Metric. This result agrees with the test on the assignment and constant mutagens on *Lines* in which it was concluded that it is important to *focus* a test on concentrations of components.

The *Grail* test results showed Textual traversal and mutation to be the most efficient live mutant detection mechanism. On analysis of the code, it was again discovered that this bias was due to large numbers of input untraversed LCSs. On any one test case, only 37 to 56 % of the LCSs were traversed. A secondary test was performed to analyse the live mutants along the execution path of the test inputs. Textual was again the most efficient traversal technique overall, but the variable reference and boundary, unary and pointer manipulations showed a higher Mutation Metric for one of the control flow traversal techniques. This was due to either concentrations of the mutation components in particular functions or to the small number of mutations performed.

The tests on the control flow and Textual traversal and mutation techniques on larger programs have shown that it is important to isolate where the mutation components are concentrated in the code. A more efficient test technique may be to isolate the functions which contain large groupings of particular code components and then to mutate those functions prior to those with fewer of the components. Input untraversed functions and LCSs can radically affect the results of testing via control flow. Mechanisms to test along the input execution path only will improve the efficiency of the overall test. However, the mutations from the untraversed LCSs must be taken into account when judging the worth of the test inputs and in determining a Mutation Score for the test set.

Table 7.19: Grail Execution Path Only Test Results

Oper	# Refs	# Muts	# TC	# Live	Tex	Pre	Ino	Pre-Gain%	In-Gain%	Res
Rel	74	370	1	158	.782	.666	.661	-14.8	-15.5	Tex
	80	400	2	164	.727	.607	.593	-16.5	-18.4	Tex
	110	550	3	196	.695	.628	.634	-9.6	-8.7	Tex
Arith	12	48	1	16	.575	.376	.340	-34.6	-40.9	Tex
	12	48	2	16	.468	.333	.333	-28.8	-28.8	Tex
Assign	149	918	1	578	.904	.871	.861	-3.6	-4.8	Tex
	156	1090	2	708	.914	.887	.873	-2.9	-4.4	Tex
Inc-	18	54	1	30	.735	.664	.629	-9.6	-14.4	Tex
Dec	19	57	2	31	.715	.676	.576	-5.4	-19.4	Tex
Logic	10	40	1	30	.904	.783	.783	-13.4	-13.4	Tex
	11	44	2	30	.831	.767	.734	-7.7	-11.7	Tex
Var-	248	1187	1	410	.718	.630	.556	-12.2	-22.6	Tex
Ref	261	1242	2	436	.699	.612	.528	-12.4	-24.45	Tex
	403	4754	3	586	.529	.589	.448	11.3	-15.3	Pre
Var-	161	535	1	293	.774	.788	.784	1.8	1.3	Pre
Bnd	168	559	2	303	.756	.765	.798	1.2	5.5	Ino
Const	153	733	1	493	.866	.847	.825	-2.2	-4.7	Tex
	171	823	2	543	.849	.825	.822	-2.8	-3.2	Tex
Unary	14	13	1	12	.980	1.00	.989	2.0	0.9	Pre
	16	16	2	13	.966	.985	.985	2.0	2.0	P/I
	16	16	3	13	.944	1.00	.943	5.9	.001	Pre
Cond	80	98	1	17	.334	.309	.262	-7.5	-21.6	Tex
	85	105	2	17	.242	.165	.091	-31.8	-62.4	Tex
Pointer	162	324	1	100	.760	.812	.781	6.8	2.7	Pre
	163	324	2	79	.694	.693	.658	-.001	-5.2	Tex
	206	410	3	78	.626	.706	.668	12.7	6.7	Pre

Chapter 8

Conclusions

'Where is the wisdom we have lost in knowledge?'
'Where is the knowledge we have lost in information?'
T.S. Eliot

This chapter summarises the topic of Mutation Analysis and assesses the contribution made by the research described in the preceding chapters. Some research areas for future development are also discussed.

8.1 Review

This section outlines the field of Mutation Analysis and the research contribution made in this thesis.

8.1.1 Introduction

The dynamic source code testing technique known as Mutation Analysis, MA, has been described in some detail in Chapter 3 in this thesis. MA is a simple technique

which involves changing source code components to syntactically correct alternatives. The altered code, the mutant program, is then executed on the same test data as the original test program. If the output from the original and the output from the mutant programs are identical, then MA has been successful in showing a weakness in either the test program or the test data.

For example, if the statement '$a = b + c$' is mutated to '$a = b - c$', the test data must include a test input in which 'c' has a non-zero value. This, in itself, might not be sufficient to distinguish the two programs, in which case the tester must analyse the values of 'a' and 'b'. If the output from the test program differs from the mutant output, the mutant is deemed dead and removed from the test. The test data is considered *relatively adequate* in that it can identify a simple fault in the original, supposedly correct, test code. If the test data cannot distinguish between the test program and the mutant, the latter is considered live. The test data is regarded as poor in quality and must be enhanced to kill the mutant. If the mutant can not be killed by any derived test data it may be equivalent, or, the code component which has been mutated may reside in code which is unreachable by any test data. At the end of a mutation test, the test code has been thoroughly analysed for simple code faults and a test set has been formed which gives confidence in the code.

MA originated in the U.S.A. in the 1970 [11, 22, 36]. It was considered resource intensive and too expensive for industrial use. Metrics derived to calculate the costs of a mutation test indicated that the number of mutants varied with the square of the number of statements or variable references in code [2, 10, 80] The cost is therefore prohibitive for the testing of large scale industrial or commercial code. Larger programs take more system resources than the small programs usually analysed by academics. Each mutant must have access to all the resources of the test code indicating that a large scale code test must be managed efficiently. Recent work [81] has been directed towards sampling mutants to generate a statistically adequate test set. Another approach is to use vector processor technology to speed up the test by executing several mutants in parallel [16, 55].

8.1.2 Research Contribution

The approach taken in this thesis was to experiment with control flow driven mutation. It was theorized that inducing mutations along the execution paths, or possible execution paths, may improve the detection rate of live mutants. If a test is constrained by time, cost or available resources, an efficient test would be one in which a large number of live mutants would be detected as early as possible in the test. The live mutant detection rate was expected to vary depending on whether a test was performed using the code control flow graph to guide the mutant production, or, the standard practice of mutating statements in typed, or Textual, order was used. Chapters 4 and 5 of this thesis discussed the approach taken and the design of the tool built to analyse the efficiency of control flow driven mutation over Textual mutation.

A C mutation system was built to allow eleven different mutagens to be applied to single file programs. A mutagen is an operator which alters the code components forming mutation components. For example, the arithmetic mutagen will alter a '+' token to one of '$-, /, *, \%$' to create syntactically correct mutant programs. The mutation system built, called the **Grail**, was comprised of three distinct sections. A preprocessor to the mutation system detected the linear code sequences (LCSs) within the source code. LCSs were defined as being linearly connected statements such that if the first statement was executed then so also was the last. The preprocessor also derived the connectivity between the LCSs. This information was used by the **Grail** mutation system to effectively traverse the control flow graph of the test program. Mutations were then induced in either Inorder or Preorder traversal sequence order.

For the smaller, and less complex programs, the number of live mutants and the mutants generated by each of the three techniques was always the same. However, the number of live mutants found per mutant generated, the live mutant detection rate, varied between the three techniques at different stages in the test. To describe the efficiency of any one test, a Mutation Metric was derived. This results in a score of 1 if all the live mutations resident in a test code are found at the start of the test. That is, every mutant formed at the start of a test is a live mutant. Mutation Metrics of near zero indicate that the live mutants were found towards the end of the mutation sequence. That is, all the dead mutants are detected before the live

mutants. The three traversal techniques were then compared for efficiency using the Mutation Metric. The higher the Mutation Metric, the more efficient the traversal mechanism for detecting live mutants.

Summary of Results

The tests performed on five programs were detailed in Chapters 6 and 7 of this thesis. The test programs varied in size from 37 to 1876 lines of code (LOC). The first two, Ramamoorthy's *Trityp* and Hoare's *Find* are well known in the testing literature [19, 71]. The other three programs were large, multi-function programs containing between 7 and 35 functions.

The results from the **Grail** experiments show that mechanisms for traversing source code and inducing mutations in a non Textual sequence order can improve the efficiency of a test. However, the results also show that several factors affect the test efficiency:

- Code Coverage. Some results showed that as code coverage increased, the difference between the traversal techniques, their Mutation Metrics, decreased. This was due to, in these tests, a large number of LCSs containing live mutants. Other results showed that as code coverage increased, the efficiency benefit of using a control flow traversal technique increased. This occurred when the live mutants were concentrated in a few functions or LCSs. If code coverage was low, as may be found per test input on large code, then the results of an individual experiment depended on the number of mutation components that had been traversed. If few were traversed there was a high live mutant count which increased the Mutation Metric for each of the mechanisms.

- Concentrations of Mutation Components. If there were only a few LCSs or functions which contained the majority of the mutation components, then one traversal sequence usually had a much greater Mutation Metric than the other two tested techniques. The mechanism which traversed, and thus mutated, the few mutation LCSs in advance of the other traversal mechanisms produced a

higher Mutation Metric. If the LCSs were written at the start of the test program, as with the assignment operators in *Lines*, see Chapter 6, then Textual traversal and mutation was more likely to have the highest Mutation Metric. Alternatively, there may be a concentration of mutation components in a function or LCS which is traversed and mutated by one of the control flow mechanisms in advance of the Textual mechanism. This scenario would lead to a control flow technique obtaining a higher Mutation Metric.

In the smaller test programs, the position of equivalent mutations affected the outcome of the experiments. If there were groups of equivalent mutations in a few LCSs, then whichever mechanism traversed, and therefore mutated, those LCSs in advance of the other mechanisms would have the higher Mutation Metric.

- Location of Dead Code. Dead code is code that can never be executed. It is usually considered to be code that is unreachable due to flaws in code predicates. However, testing via control flow demonstrated that some LCSs, or even functions, were unreachable from any other LCS. Textual traversal and mutation can generate more mutants than either Inorder or Preorder traversal and mutation. The latter mechanisms cannot reach some LCSs and therefore the number of live mutants generated was lower when this condition occurred.

- Matrix of LCS - Test Input. Although not an automated part of the **Grail** mutation system, a data file was created to hold information on the traversal of LCSs by test data. This allowed test inputs to be chosen if they traversed LCSs containing mutation components. The matrix was also useful for developing test data and for locating dead code. A cell of the matrix containing a zero indicated that no test input had achieved traversal of a particular LCS.

- Zombie Mutants. In some tests, notably in the larger programs, some mutations died under application of a test case, but came back to life on another test. The mutants were termed Zombie mutants and were due to memory garbage problems. The mutagen which mostly demonstrated the Zombie mutants was the assignment operator. The statement '$a = 0$' could be mutated to '$a+ = 0$' in C. The mutant could be either live or dead depending on the value stored in the memory location accessed by variable 'a'. Thus, these mutants had to

be removed from the test and the tester was forced to initialise all variables. It was noted that the initialising statements should not be mutated, or that their outcomes had to be ignored.

8.2 Assessment of Work

The general aim of the research undertaken was to apply MA to large, multi-function code and to investigate its viability as a useful testing technique.

The experiments performed, although not compared with other test techniques for efficiency or fault detection ability, show that MA is an effective technique for locating weaknesses in the test process as well as the test code. MA enables the tester to develop more probing test inputs and to achieve high code coverage by encouraging the demise of live mutant programs. Dead code and concentrations of live mutations can be located in large programs and the efficiency of the detection can be improved by the informed choice of a code traversal strategy.

The most efficient code traversal and mutation strategy is dependent on code coverage and groupings of live mutations. Such information can be harnessed to provide a tester with a mechanism for conducting further code tests. Knowing where mutation components are grouped, or which LCSs are untraversed, aid the development of an adequate test set. Analysis of mutation via code control flow traversal can indicate which mutations are likely to live or die depending on their position in the code. For example, in the larger test programs, mutations of conditional statements tended to die if they were executed near the start of execution. Some mutations, such as the variable reference mutations tended to be fragile and die very quickly.

An important conclusion from this research is the need for large scale code tests to be managed:

- Information regarding test input to code traversal should be automated and manipulated to aid test data generation as well as Revision and Regression testing.

- Mutation component concentrations, as detected by the **Grail** mutation system, should be analysed to reduce the cost of a test by directing it towards particular functions or LCSs. That is, once concentrations of code components, or even particular code constructs, are located, a test could be focused onto those concentrations.

- The predisposition of code components to liveness under mutation was not fully explored in the experiments undertaken. Full code coverage would have been necessary for this to be analysed and this was not possible in the time scale available. However, it was noted that variable reference and arithmetic mutations die quickly. Assignment operators have a lot of live, and possibly equivalent, mutations in the C language. Simple pointer manipulations are not easy to kill unless a lot of list processing information is printed. A state based comparison may be necessary for pointer mutations.

- The survey of common errors indicated that different code tasks show different fault groupings. A program written to reduce scientific data is more likely to contain round-off or boundary errors than a program written to search for a name in a database. Consequently, particular mutations, or other testing strategies, may be applicable to different types of code. Also, programmers and testers are biased by their own sensitivity to faults. They tend to code around or look for particular constructs. This information may help to improve the efficiency of a test by primarily analysing these problematic constructs.

These conclusions, and the results from Chapters 6 and 7, meet the criteria for success as laid out in Chapter 1, Section 2.

8.3 Future Directions

The research described in this thesis indicates that to test, and manage the test, of a large system, a large quantity of information regarding the code should be stored and manipulated.

- Details of LCS and function coverage would aid the development of test data and aid in the re-application of test inputs for Revision or Regression tests. This could include data regarding concentrations of test components or live mutations.

- A test information database could also store data regarding particular code components or constructs regarded as suspect by the programmer or designer. This could be enhanced by data from a larger survey of common errors in C code. A long term study of industrial and commercial systems could be undertaken to determine more precisely the commonality of faults.

- A history of the test and the systems development could also be useful. Information regarding previous bug reports, faults found, or even the programming group who coded a module could be used to choose sample mutations or to direct the test onto specific functions.

All this information, even if only readable to the human tester, could improve the efficacy of a test by focusing it onto problematic functions or constructs.

Further tests on a variety of code could determine general guidelines for the applicability of different mutagens to code. To this end, the **Grail** mutation system could be extended for more research on the likelihood of mutations surviving. This could affect the sampling of mutations as it would be practical to choose mutations that are more likely to survive test input execution. It would be expected that these more exacting samples would be a more stringent test of a program than randomly chosen mutation samples. Test suites that were sample mutation adequate could then be compared with strong mutation adequate and data flow adequate test suites.

The ability to focus a test onto specific units within a system, such as those designated critical, may be a useful device for reducing the cost of a test. Focusing onto particular LCSs, functions or code constructs could allow a form of dynamic impact analysis to be performed. Mutating a newly inserted function using control flow mutation starting at the point of execution of the function, could show the effect of faults along the execution path. This would allow mutation analysis, one of the most stringent testing techniques, to be applied on a partial basis to a large system. In contrast to

sampling mutants, this system would sample code to undergo stricter analysis and could form part of an integration test.

Nil Desperandum

Appendix A

A Sample Execution of the Grail

The following demonstrates a sample execution of the Grail Mutant Maker on the *Trityp* program. Explanatory notes have been added in parenthesis. The data file is also given.

```
M1.0 Token Collecting                 {M1 is the Mutant Maker}
Enter the Program name : Trityp
Enter the number of test cases : 4
Mutant kill - test case data : n
Run-time options : y/n n
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************************
Choose the components to be mutated
End input with a  0
%%%%%%%%%%%%%%%%%%%%
Relational Operators 1
Arithmetic Operators 2
Assignment Operators 3
   Inc/Dec Operators 4
   Logical Operators 5
  Variable Reference 6
   Variable Boundary 7
Constant Replacement 8
   Unary Replacement 9
S_Pointer Arithmetic 10
    Context Negation 11
*******************************************
5
0
Enter 1 for Textual Pattern
Enter 2 for PreOrder Pattern
Enter 3 for InOrder Pattern
2
Compilation of P0 successful          {Compilation of Test Program
TC output completed                    has been successful. Available
                                       Test cases have been executed.}
```

```
FUNCTIONS:                          {Names of code functions.}
main

Token Positions of Logical ops
44 0                                {Token positions and mutation
55 1                                 component set offset.
66 0                                 0 = && ('and') ,  1 = || ('or')}
total_logop : 3
MUTS_TO_DO 3

PreOrder Section
mutate_seg 1 0 10 12               {Mutate_seg is the function which
mutate_seg 1 1 13 40                analyses each LCS. The first call to it
mutate_seg 1 2 41 48                shows that it is mutating function 1
Mutating token 44 1                 (main), LCS 0. This is the LCS bound by
Live Mutant && 44 1 1               tokens 10 and 12. The three live mutants
Live Mutant && 44 1 2               are of token number 44, the && symbol.
Live Mutant && 44 1 3               The live mutants are described by their
muts done :  1  total :  3          offset, 1 = '||', 2 = '&' and 3 = '|'.
-------------------------------------------
mutate_seg 1 3 49 51                That is,  the logical 'or'
mutate_seg 1 4 52 59                and the bitwise 'and' and 'or'.}
Mutating token 55 1
Live Mutant || 55 2 2
dead mutant 2 55 2
Live Mutant || 55 2 3
Live Mutant || 55 2 4              {Token 55 is live after test case 0 has
dead mutant 1 55 4                  been applied but some of the mutants are
Live Mutant || 55 2 0               killed by test cases 1 and 2.}
dead mutant 2 55 0
muts done :  2  total :  3
-------------------------------------------
```

```
mutate_seg 1 5 60 62
mutate_seg 1 6 63 70
Mutating token 66 1
Live Mutant && 66 3 1
dead mutant 2 66 1
Live Mutant && 66 3 2
Live Mutant && 66 3 3
dead mutant 2 66 3
Live Mutant && 66 3 4
dead mutant 1 66 4
muts done :  3  total :  3
-----------------------------------------
mutate_seg 1 7 71 78
mutate_seg 1 9 86 87
mutate_seg 1 19 151 152
mutate_seg 1 21 160 159
end of tokens                        {These were checks on the test
NO CONNECTIONS 1 21                   code token list. LCS 21 is the
mutate_seg 1 8 79 85                  last LCS in Trityp.}
mutate_seg 1 10 88 114
mutate_seg 1 11 115 118
mutate_seg 1 12 119 121
mutate_seg 1 13 122 125
mutate_seg 1 14 126 133
mutate_seg 1 16 141 142
mutate_seg 1 18 150 150
mutate_seg 1 15 134 140
mutate_seg 1 17 143 149
mutate_seg 1 20 153 159
*****************************************
```

RESULTS:

```
&& 44 L1 1                  {The five live mutants are :
&& 44 L1 2                   Token number 44, the '&&' symbol;
&& 44 L1 3                   mutants '||', '&' and '|'.
|| 55 L2 3                   Token number 55, the '||' symbol;
&& 66 L3 2                   mutant '|'.
no of live muts : 5          Token number 66, the '&&' symbol;
                                   mutant '&'.}
```

****

DATA FILE:

```
      rama      #Funcs  #Lines  #Stats  #Preds  #Loops  #TC
        2       1       37      13      5       0       4
tk_no/ tk_ref/ fn_no/ seg_no/ #live/ #m_gen/ #groups gen
44      L1      1       2       3       4       1
55      L2      1       4       4       8       2
66      L3      1       6       5       12      3
0

#Possible Mutants       3
```

Appendix B

Program Details

The following details the code, test inputs and control flow diagrams for the three smaller programs. The function name list for the Grail test is also included.

B.1 Trityp.c

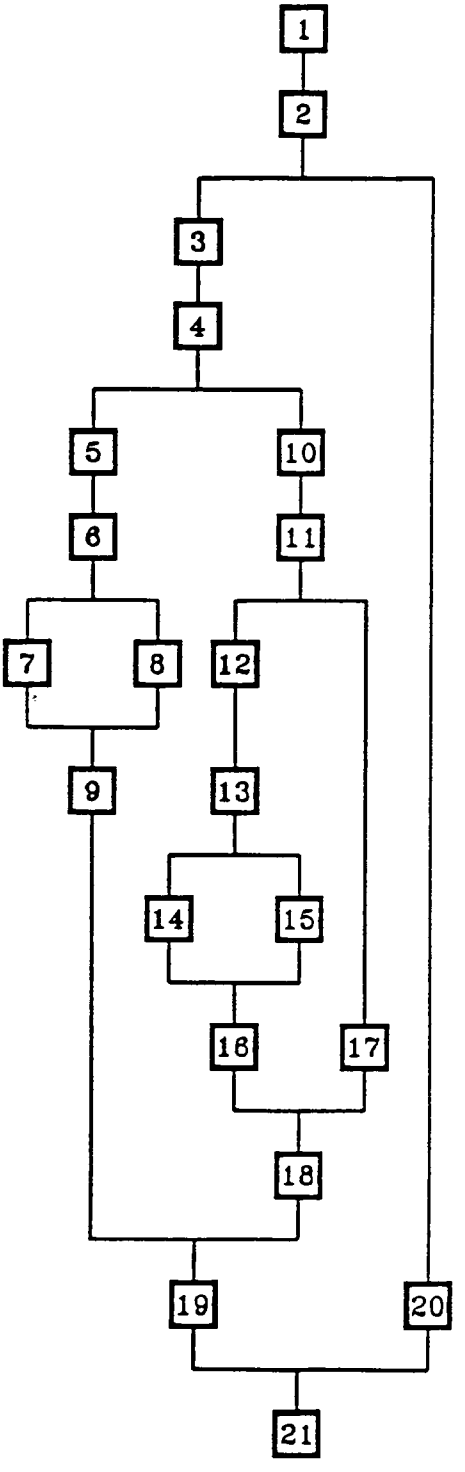
```
/*************************************************************/
/* Ramamoorthy's triangle - Trityp                           */
/* This program read in 3 sides of a triangle and outputs    */
/* the type of the triangle.                                 */
/*                                                           */
/* C version of A.J. Offutt's Program 2                      */
/*************************************************************/
#include <stdio.h>

int a,b,c,d ;
main()

{
 scanf("%d%d%d",&a, &b, &c);
 /* printf("%d\t%d\t%d\n",a,b,c); */
 if ( a >= b && b >= c)
    {
     if (a == b || b == c)
         {
         if ( a == b && b == c)
             printf("%s\n", "Equilateral");
         else
             printf("%s\n","Isosceles");
         }
     else
         { a = a * a;
           b = b * b;
           c = c * c;
           d = b + c;
           if ( a != d)
              { if ( a < d )
                    printf("%s\n","Acute");
                else
                    printf("%s\n","Obtuse");
              }
           else
               printf("%s\n","Right Angled Triangle");
```

```
        }
    }
 else
    printf("%s\n","Triangle Sides not in order");
}
```

Test Inputs for Trityp

Test Input	a	b	c
1	2	12	27
2	5	4	3
3	26	7	7
4	19	19	19
5	14	6	4
6	24	23	21
7	7	5	6
8	5	5	3

Trityp LCS Control Flow Graph



B.2 Find.c

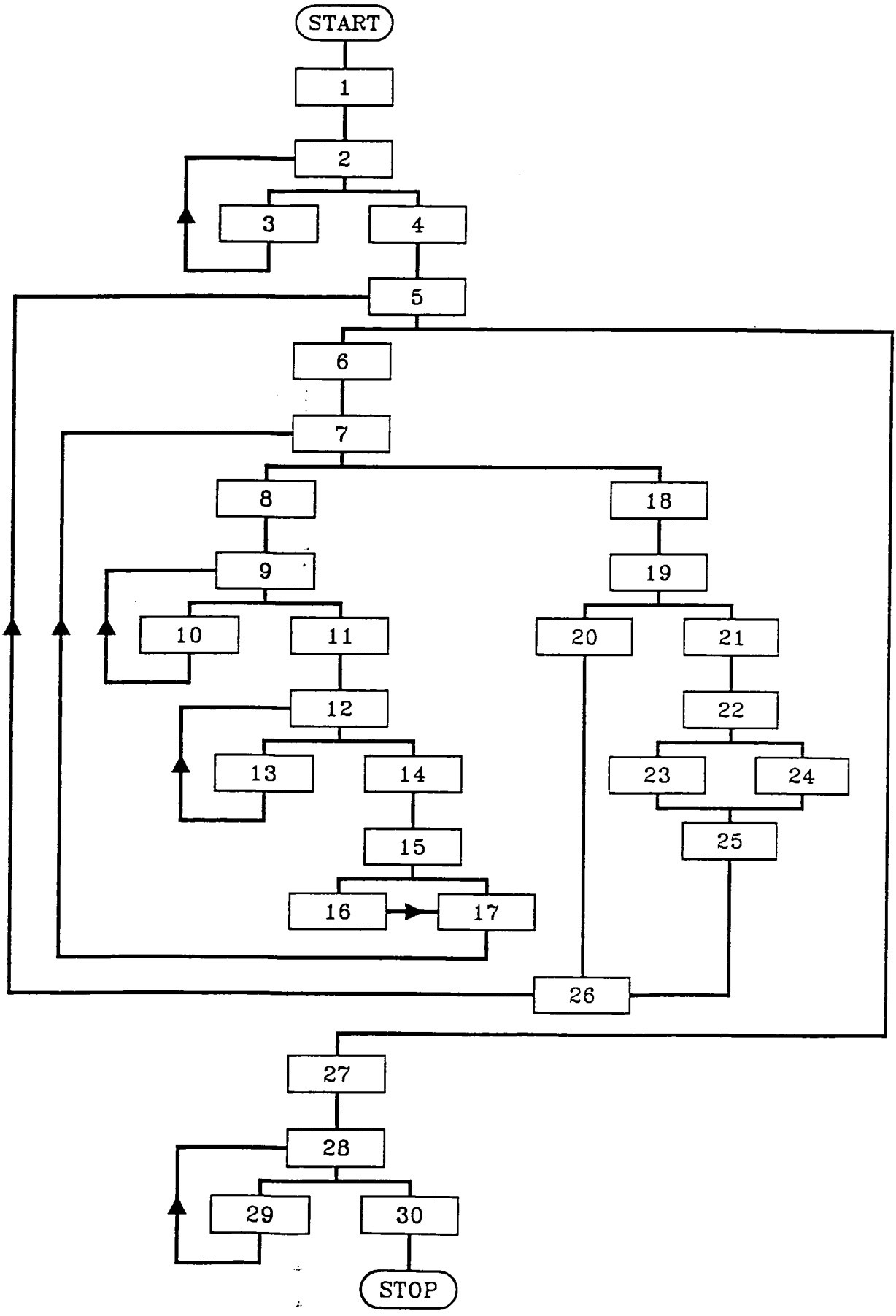
```
#include <stdio.h>

int a[11], n, i, j, m, ns, r, f, w, ii;

main()
{

  printf("%s","How many numbers : ");
  scanf("%d",&n);
  m = 1;
  ns = n;
  n++;
/*  printf("%d\t%d\n",n,ns); */
  printf(" %s", "Enter the numbers : ");
  for (i = 1; i < n; i++ )
      scanf("%d",&a[i]);
  printf("%s", "Enter pivot : ");
  scanf("%d",&f);

 while (m < ns)
 { r = a[f];
   i = m;
   j = ns ;
   while ( i <= j )
   { while ( a[i] < r )
     i += 1;
     while ( r < a[j])
     j -= 1 ;
     if ( i <= j )
     {  w = a[i];
       a[i] = a[j] ;
       a[j] = w ;
       i += 1;
       j -= 1;
     }
   }
   if ( f <= j )
```

```
        ns = j ;
    else
       { if ( i <= f )
            m = i ;
         else
            m = ns ;
       } ;
  }
  for ( i = 1 ; i < n ; i ++ )
       printf("%d", a[i]) ;
  return(0);
}
```

Test Inputs for Find

Test Input											
1	9	-19	34	0	-4	22	12	222	-57	17	5
2	3	7	9	7	3						
3	4	2	3	1	0	3					
4	4	-5	-5	-5	-5	1					
5	4	1	3	2	0	3					
6	4	0	2	3	1	3					
7	1	0	1								

Find LCS Control Flow Graph



B.3 Lines.c

```
/*************************************************************/
/* Kernighan & Ritchie 'The Ansi C Book'                     */
/*                      2nd Edition            pgs 108-110 */
/*                                                           */
/* Reads in lines and outputs them in sorted order.          */
/* Uses pointers and arrays.                                 */
/*************************************************************/

#include <stdio.h>
#include <string.h>

#define MAXLINES 10 /* max #lines to be sorted */
#define MAXLEN   30 /* length of input line */
#define ALLOCSIZE 100 /* available space */

static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;

char *lineptr[MAXLINES];

char *alloc(n)
int n;
{
 if (allocbuf + ALLOCSIZE - allocp >= n)
   {
    allocp += n;
    return allocp - n;
   }
 else
   return 0;
 }

int getline (s, lim)
 char s[];
 int lim;
{
```

```
 int c,i;
 i = 0;

 while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
       s[i++] = c;
 if (c == '\n')
       s[i++] = c;
 s[i] = '\0';
 return i;
}


int readlines(lineptr, maxlines)
char *lineptr[];
int maxlines;
{
 int len, nlines;
 char *p, line[MAXLEN];

 nlines = 0;
 while ((len = getline(line, MAXLEN)) > 0)
       {
       if (nlines >= maxlines)
          return -1;
       if ((p =alloc(len)) == NULL)
          return -1;
       line[len-1] = '\0';
       strcpy(p,line);
       lineptr[nlines++] = p;
       }
 return nlines;
}


writelines( lineptr, nlines)
char *lineptr[];
int nlines;
{
 while (nlines-- > 0)
       printf("%s\n", *lineptr++);
```

```
}


swap(v, i, j)
char *v[];
int i,j;
{
 char *temp;
 temp = v[i];
 v[i] = v[j];
 v[j] = temp;
}


qsort(v, left, right)
char *v[];
int left, right;
{
 int i, last;
 if(left >= right)
   return;
 swap(v, left, (left+right)/2);
 last = left;
 for ( i=left+1; i <= right; i++)
     if (strcmp(v[i], v[left]) < 0)
        swap ( v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
}


main()
{
int nlines;
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
     qsort(lineptr,0, nlines-1);
     writelines(lineptr, nlines);
     return 0;
```

```
    }
  else
    {
     printf("error : input too big to sort\m");
     return 1;
    }
}
```

Test Inputs for Lines

Test Input 1:

```
these are
the lines
i wish
to
sort
```

Test Input 2:

```
these are
the lines
i wish
to
sort for
future tests
on this
program
for
test on the file
lines.c
```

Test Input 3:

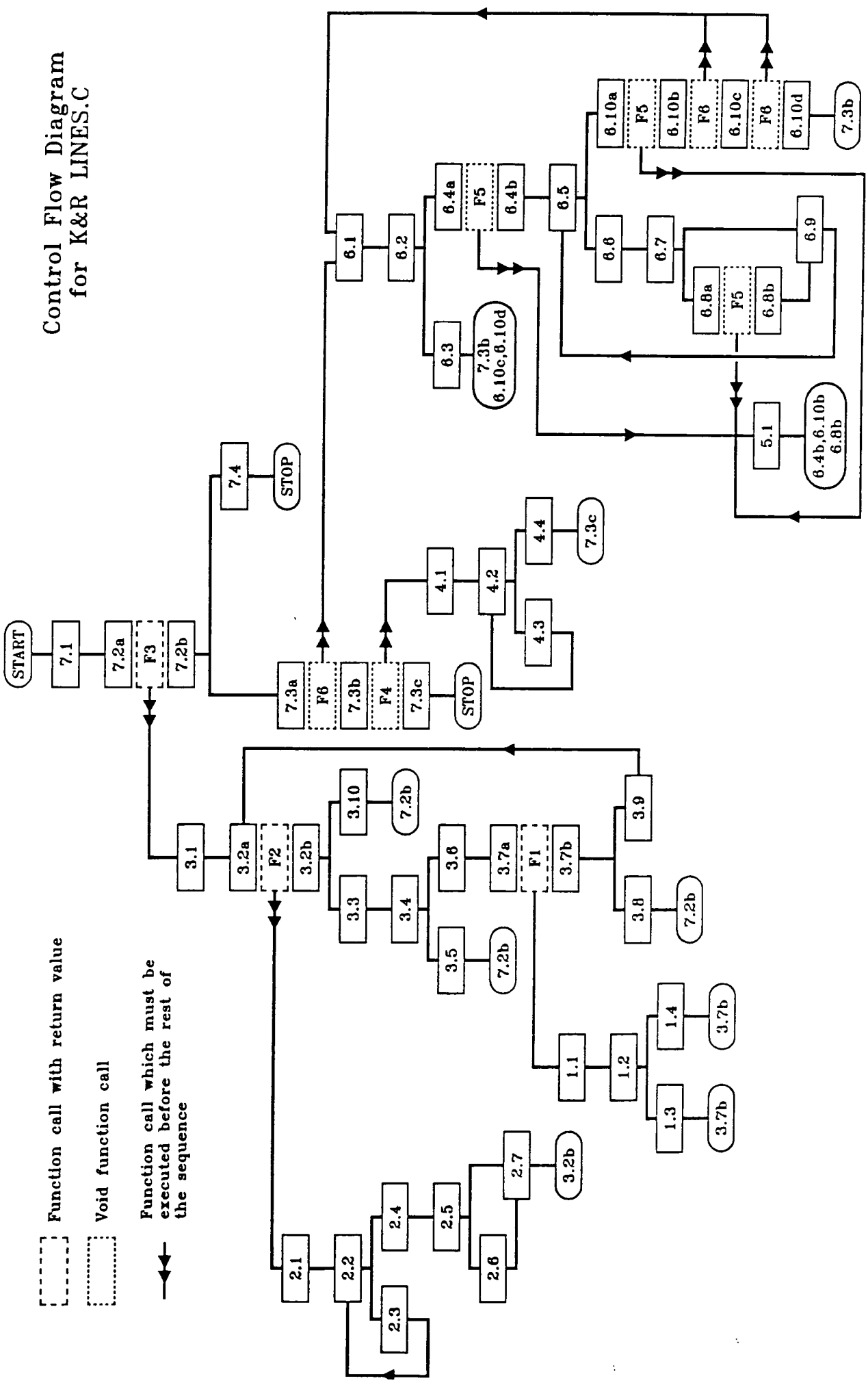
```
lines
lines
lines
lines
lines
lines
lines
lines
lines
lines
lines
```

Test Input 4:

```
this is a test on
the line length which should only be thirty
characters.
```

Test Input 5:

```
this is a test on one long single line to try to kill off some mutants for the research I'm doing!.
```


Control Flow Diagram
for K&R LINES.C
Function call with return value
Void function call
Function call which must be executed before the rest of the sequence
START
7.1
7.2a
F3
7.2b
7.4
STOP
7.3a
F6
7.3b
F4
7.3c
STOP
4.1
4.2
4.3
4.4
7.3c
6.1
6.2
6.3
7.3b 6.10c,6.10d
6.4a
F5
6.4b
6.5
6.10a
F5
6.10b
F6
6.10c
F6
6.10d
7.3b
6.6
6.7
6.8a
F5
6.8b
6.9
5.1
6.4b,6.10b 6.8b
3.1
3.2a
F2
3.2b
3.3
3.10
7.2b
3.4
3.6
3.5
7.2b
3.7a
F1
3.7b
3.9
3.8
7.2b
1.1
1.2
1.3
3.7b
1.4
3.7b
2.1
2.2
2.3
2.4
2.5
2.6
2.7
3.2b

B.4 Grail.c

The function list for the Grail code is as follows:

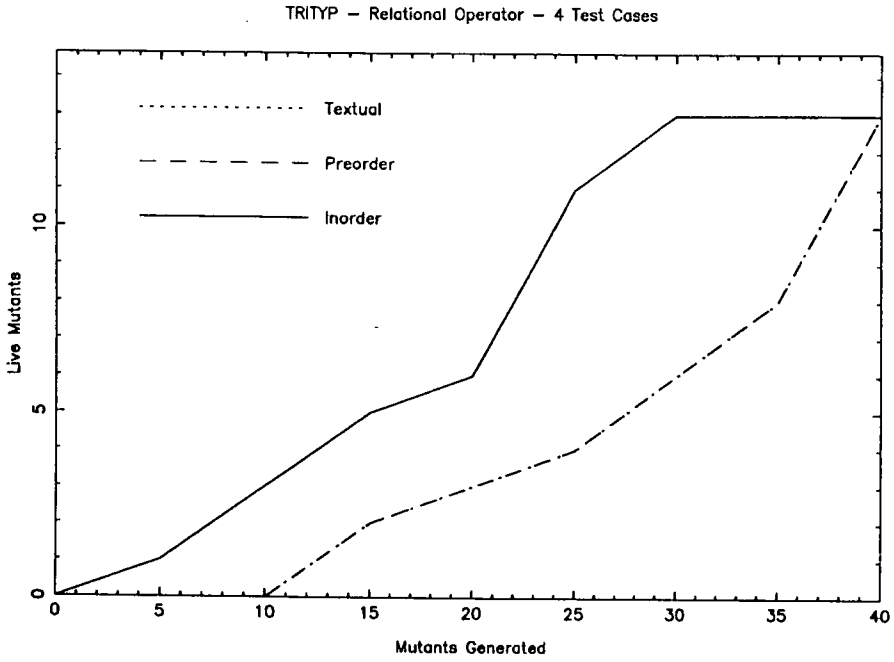
```
malloc-exit 1            {The first eleven functions are
new-op-st 2               memory allocation routines.}
new-sg-st 3
new-ft-st 4
new-fc-st 5
new-tk-st 6
new-p-st 7
new-par-st 8
new-strct_st 9
new-cond-st 10
new-comp-st 11
fnames-fnos-read 12     {Reads in function names and their
set-store-st 13          numbers, e.g. malloc-exit and 1}
create-ftst-queue 14
set-store-prog 15       {Functions 13 to 18 read in the token
add-queue 16             file and allocate memory, by calling
read-data 17             Functions 2 to 11, to create the
read-in-prog 18          necessary structures.}
phase0 19               {Compiles the test program.}
start-up 20             {Reads in user input.}
set-up-comps 21         {Initialises mutation component queue.}
is-in-relop 22
is-in-arith 23
is-in-assign 24
is-in-incdec 25
is-in-logop 26          {Functions 22 to 31 check the token
is-in-typedef 27         types and mutation set offset.}
is-in-reserve 28
which-numeric 29
is-unary 30
is-in-unary 31
```

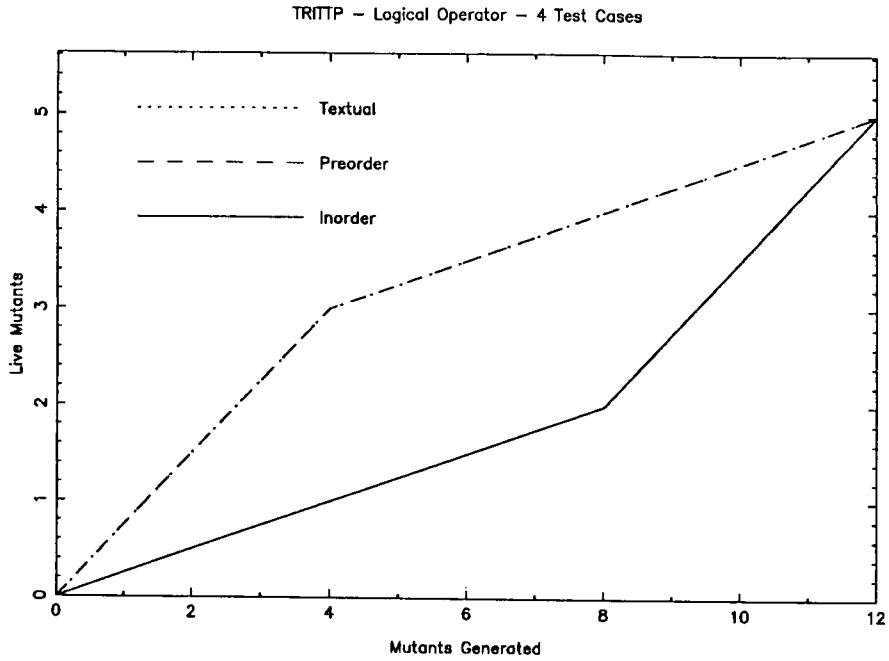
```
view-mutants 32          {Displays the live mutants.}
search-prog 33           {Searches data for mutation components.}
struct-create 34         {Creates the function-LCS structure.}
main 35                  {Main routine; calls other functions.}
```

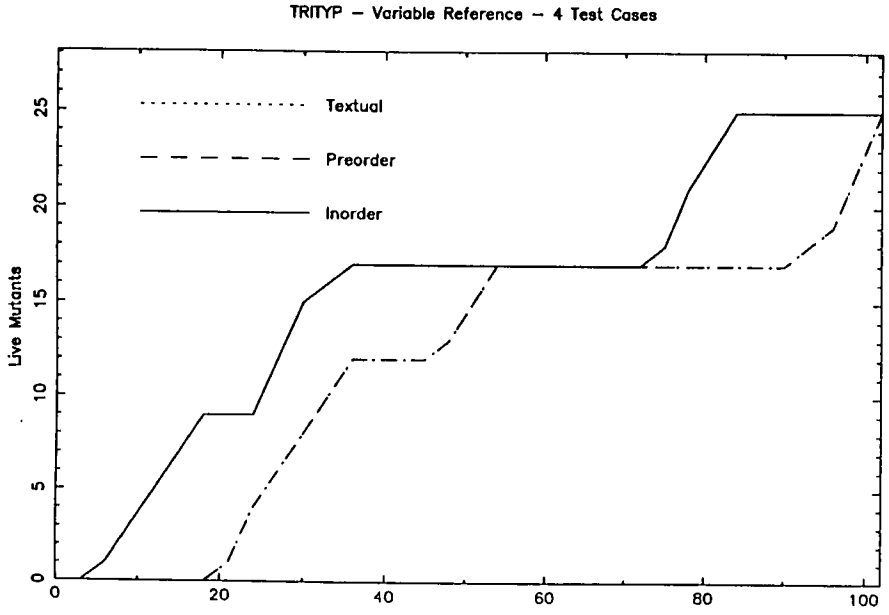
Appendix C

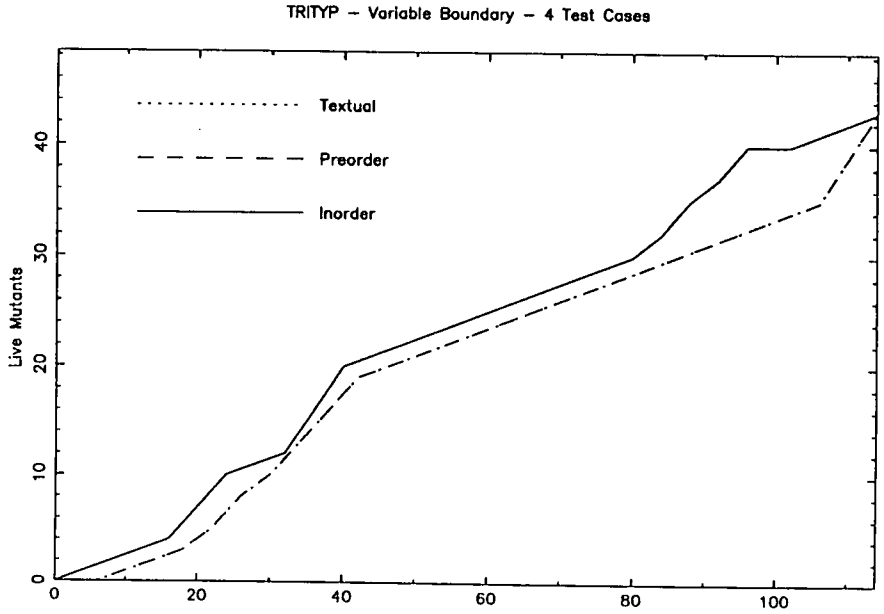
Sample Plots of Live Mutant Detection Rates

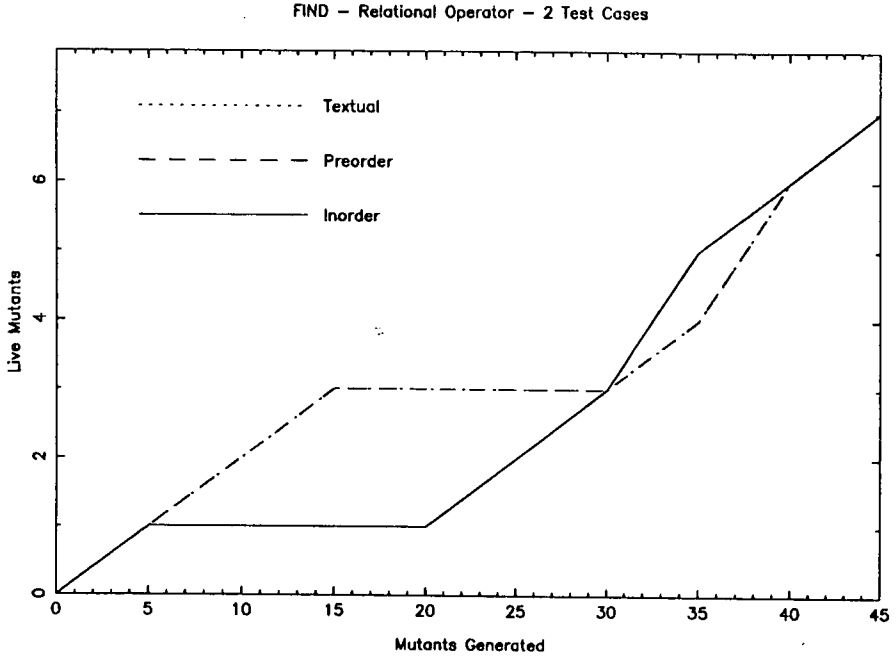
The following pages show a variety of the plots produced by the Live Mutant Analysis stage of the **Grail** prototype mutation tool.

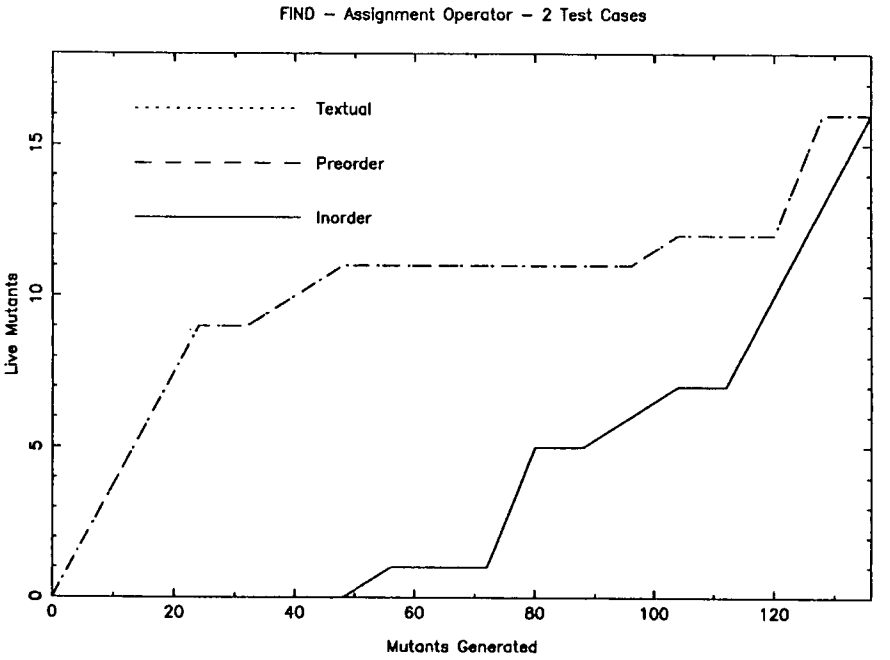

TRITYP – Relational Operator – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
5
10
15
20
25
30
35
40

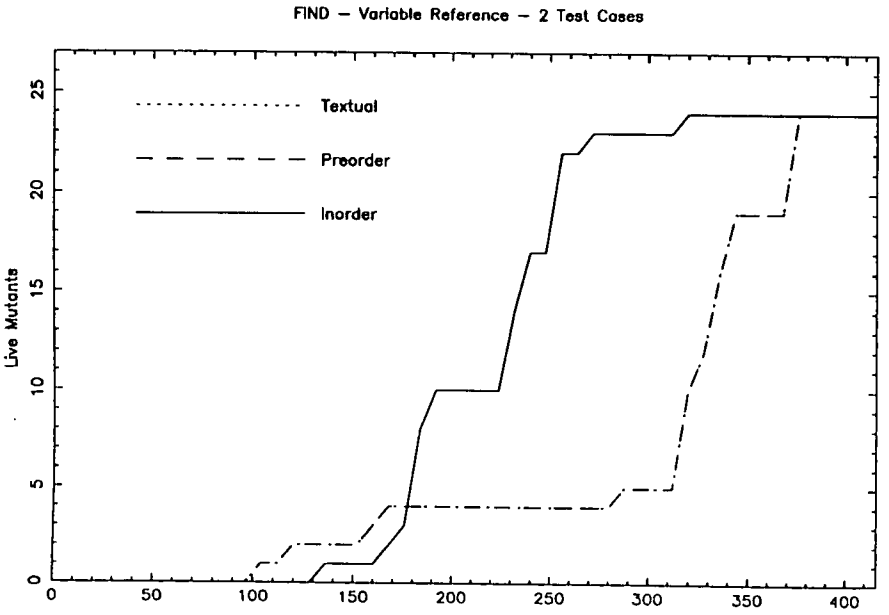

TRITTP – Logical Operator – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
2
4
6
8
10
12

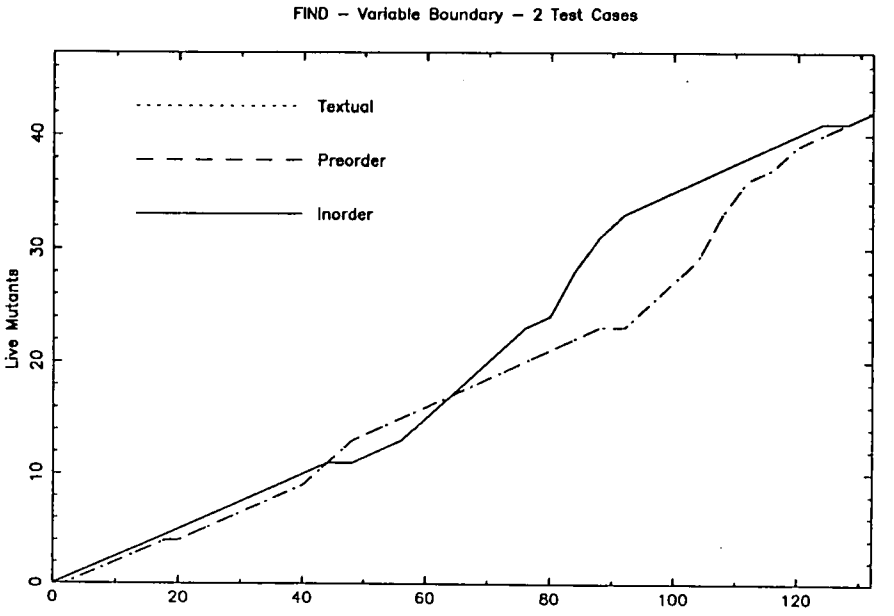

TRITYP – Variable Reference – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
0
5
10
15
20
25
0
20
40
60
80
100

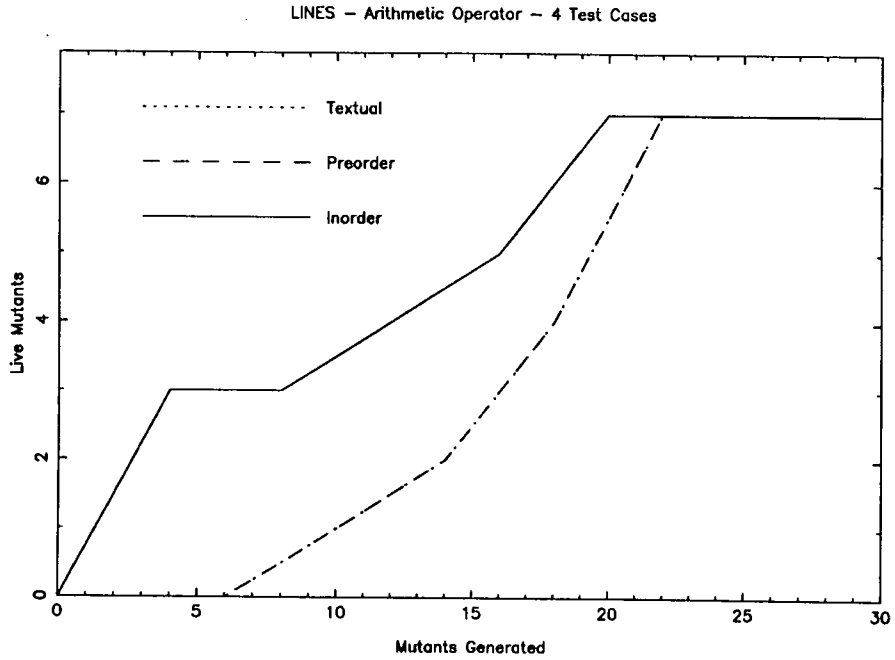

TRITYP – Variable Boundary – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
0
10
20
30
40
0
20
40
60
80
100

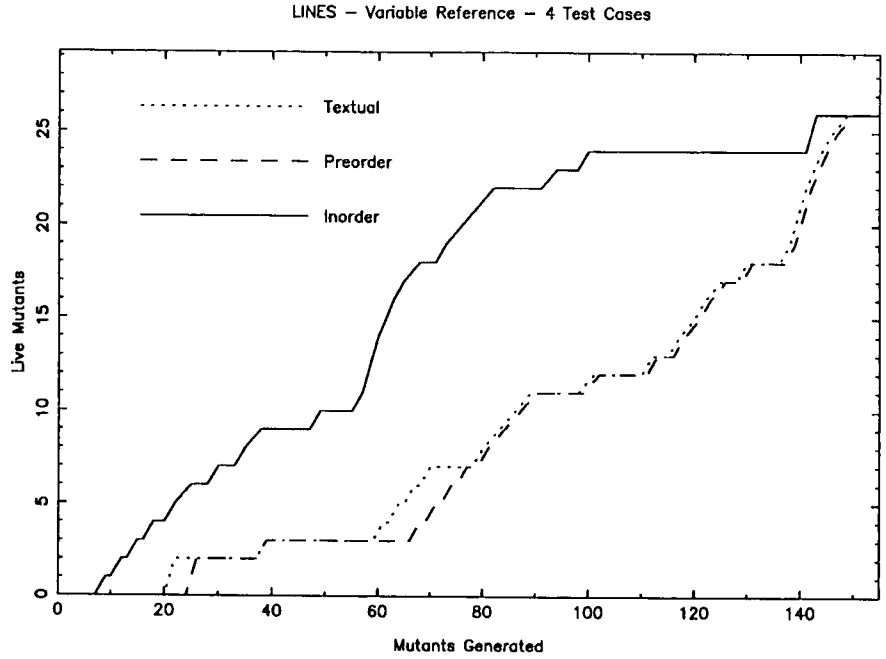

FIND – Relational Operator – 2 Test Cases
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
2
4
6
0
5
10
15
20
25
30
35
40
45

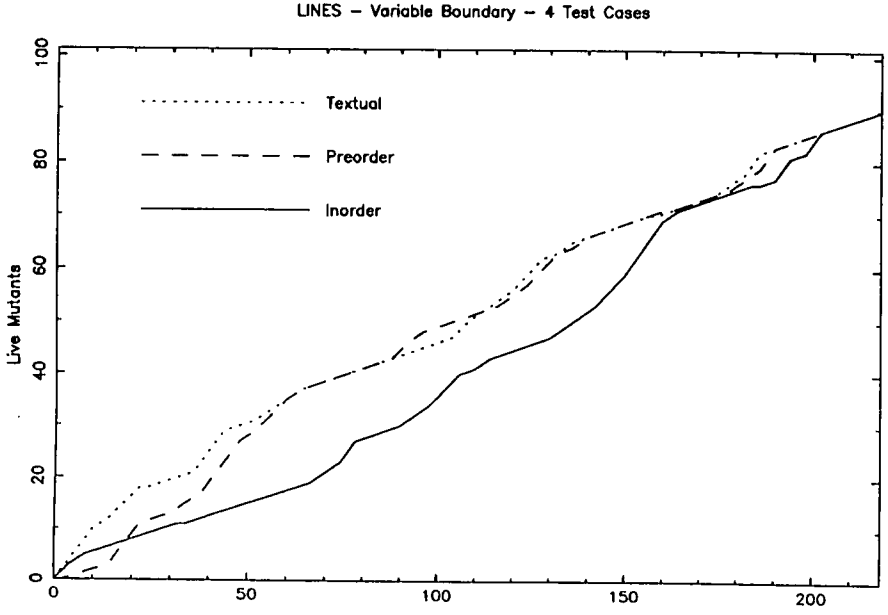

FIND – Assignment Operator – 2 Test Cases
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
5
10
15
0
20
40
60
80
100
120

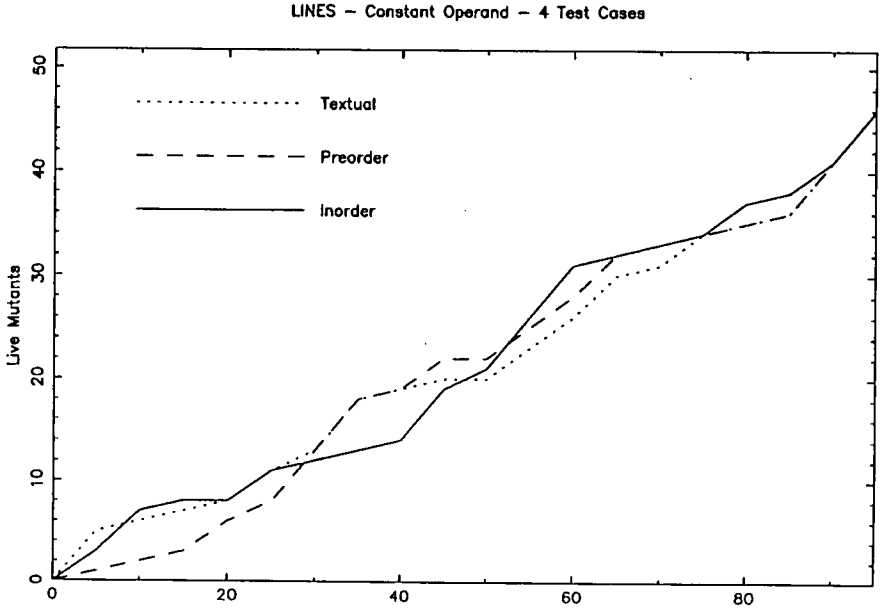

FIND – Variable Reference – 2 Test Cases
Textual
Preorder
Inorder
Live Mutants
0
5
10
15
20
25
0
50
100
150
200
250
300
350
400

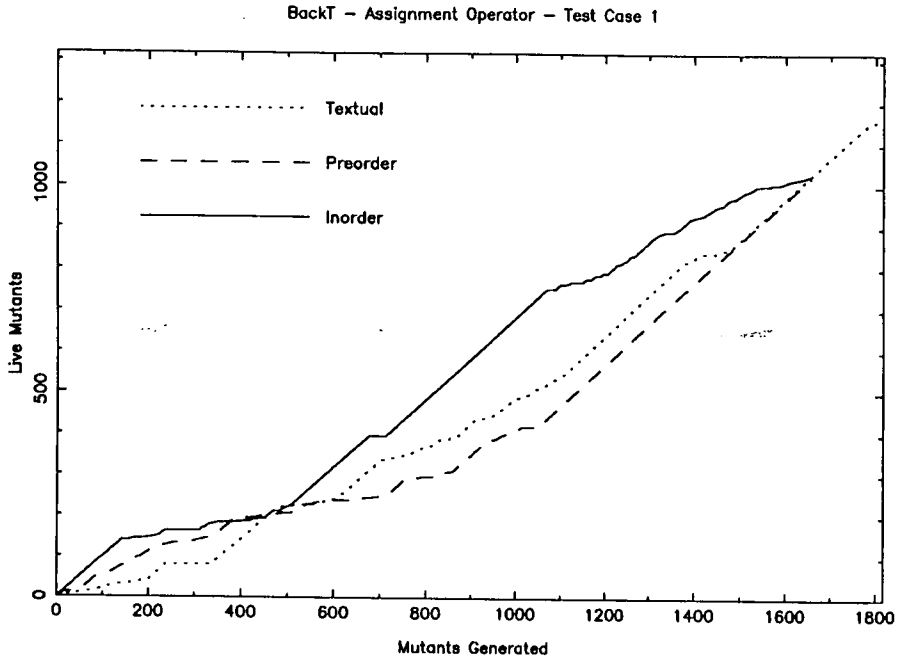

FIND – Variable Boundary – 2 Test Cases
Textual
Preorder
Inorder
Live Mutants
0
10
20
30
40
0
20
40
60
80
100
120

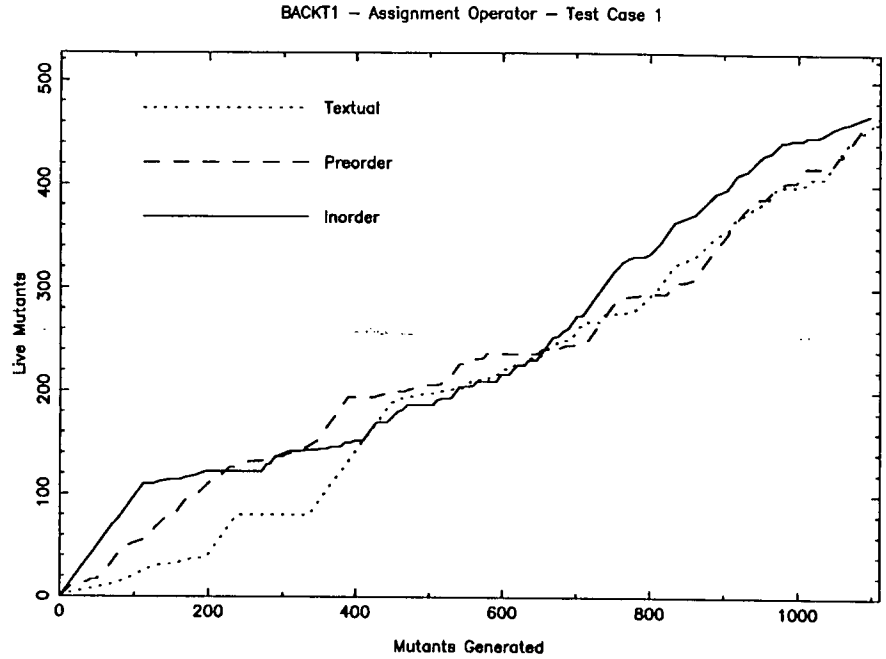

LINES – Arithmetic Operator – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
2
4
6
0
5
10
15
20
25
30

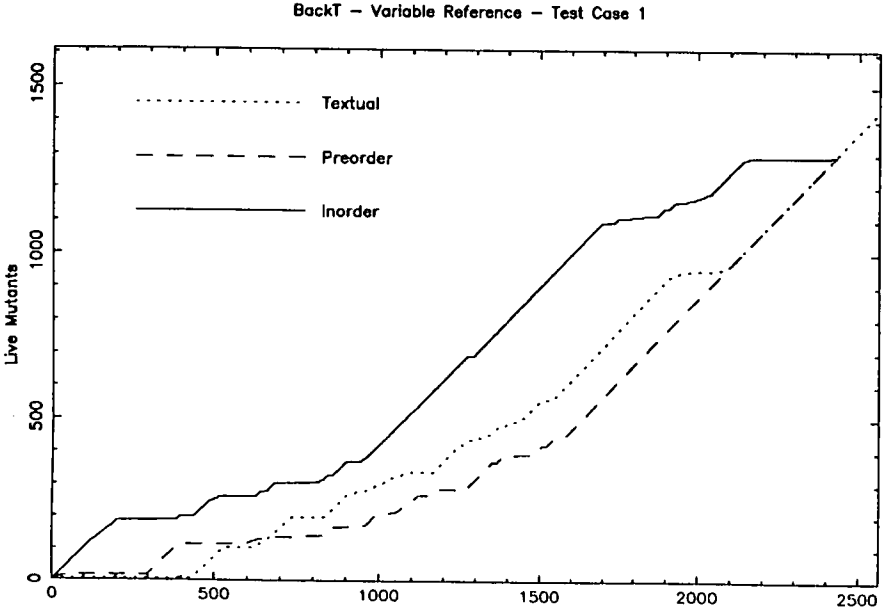

LINES – Variable Reference – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
5
10
15
20
25
0
20
40
60
80
100
120
140

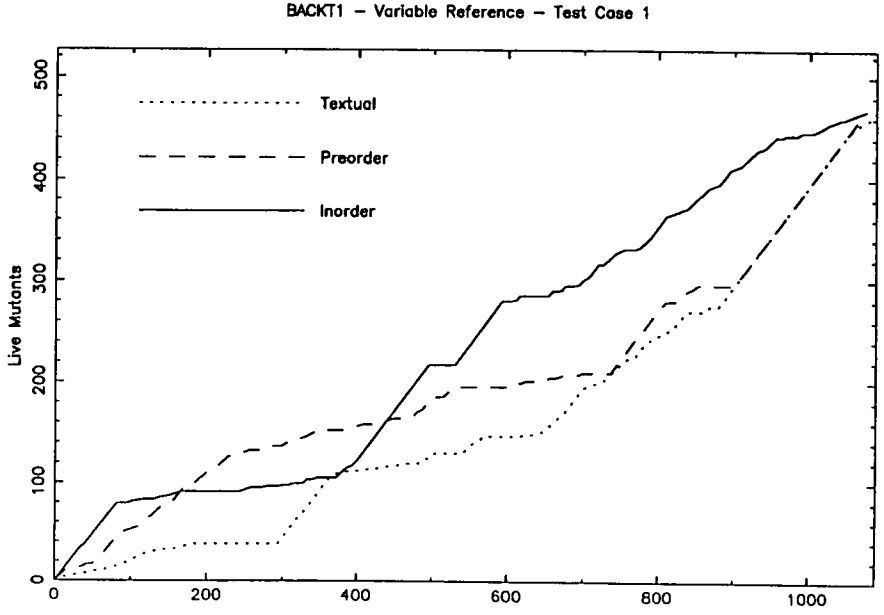

LINES – Variable Boundary – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
0
20
40
60
80
100
0
50
100
150
200

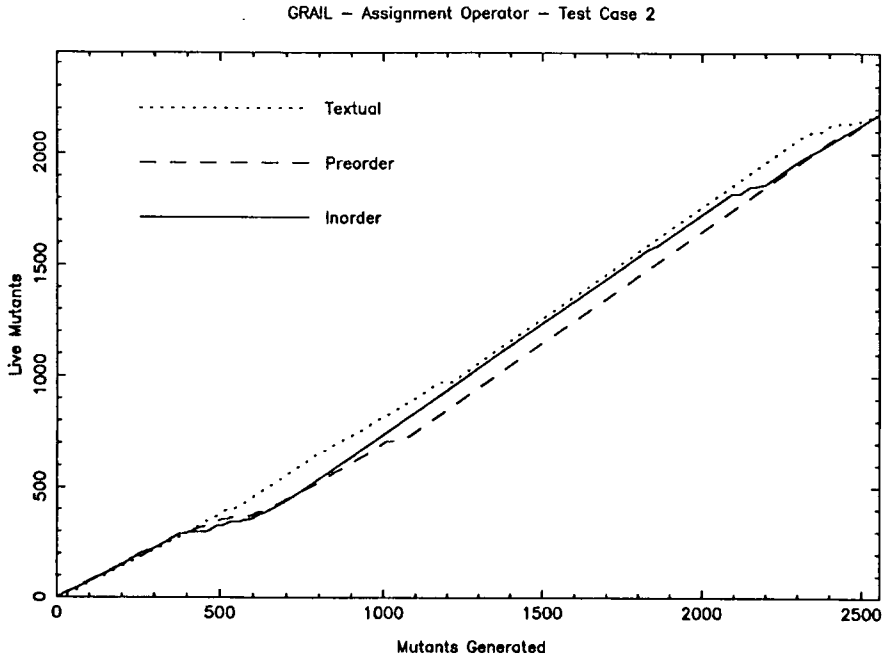

LINES – Constant Operand – 4 Test Cases
Textual
Preorder
Inorder
Live Mutants
0
10
20
30
40
50
0
20
40
60
80

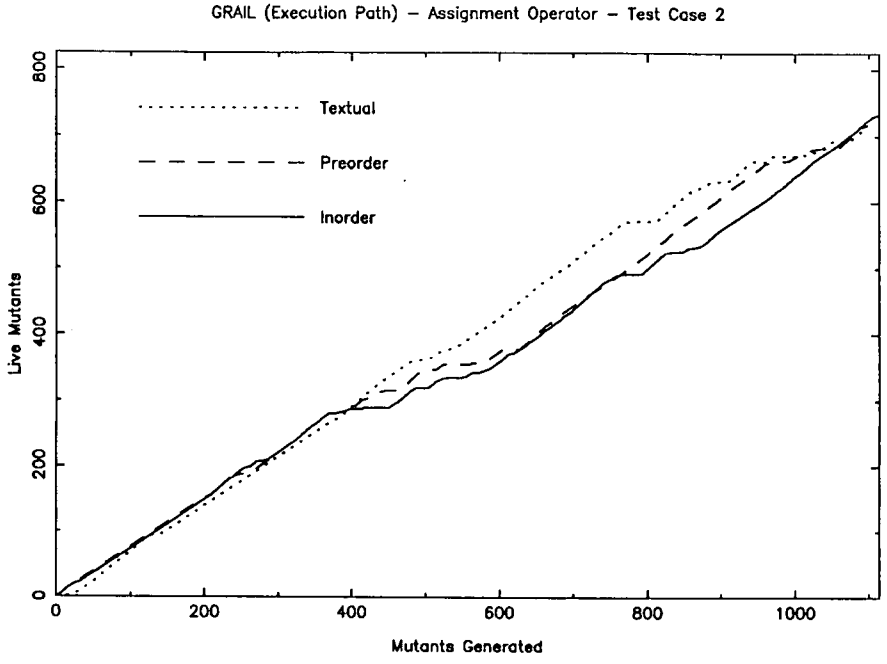

BackT – Assignment Operator – Test Case 1
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
500
1000
0
200
400
600
800
1000
1200
1400
1600
1800

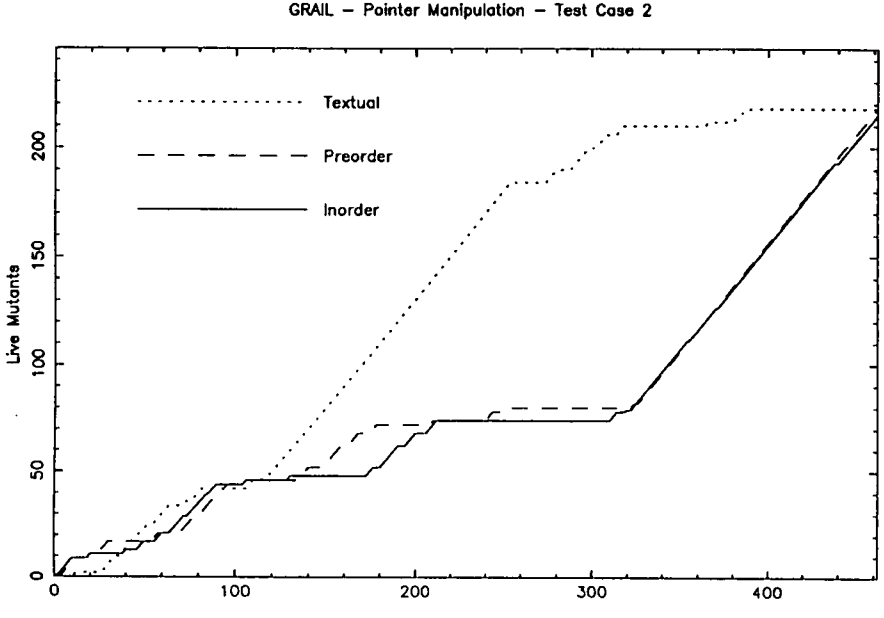

BACKT1 – Assignment Operator – Test Case 1
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
100
200
300
400
500
0
200
400
600
800
1000

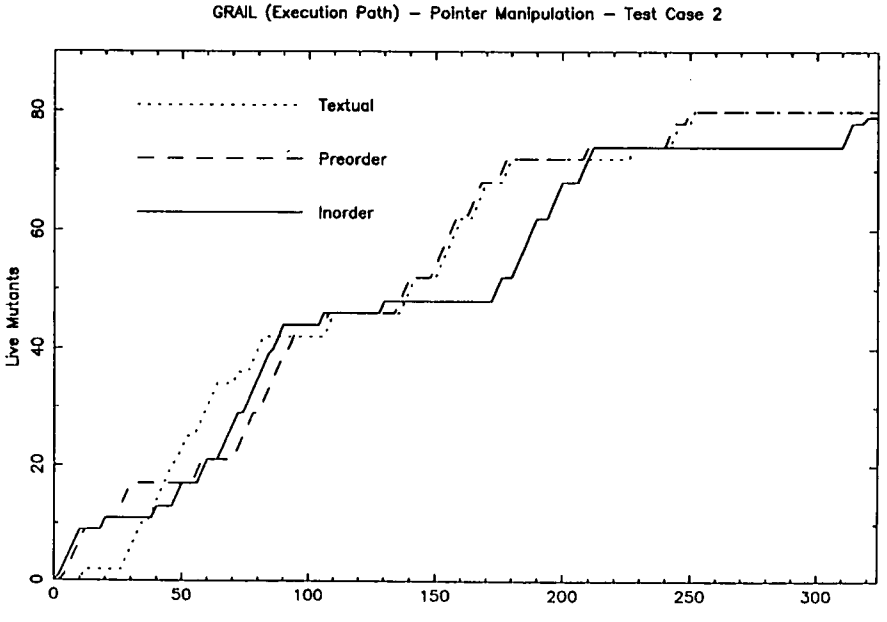

BackT – Variable Reference – Test Case 1
Textual
Preorder
Inorder
Live Mutants
0
500
1000
1500
0
500
1000
1500
2000
2500


BACKT1 – Variable Reference – Test Case 1
Textual
Preorder
Inorder
Live Mutants
0
100
200
300
400
500
0
200
400
600
800
1000


GRAIL – Assignment Operator – Test Case 2
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
500
1000
1500
2000
2500


GRAIL (Execution Path) – Assignment Operator – Test Case 2
Textual
Preorder
Inorder
Live Mutants
Mutants Generated
0
200
400
600
800
1000


GRAIL – Pointer Manipulation – Test Case 2
Textual
Preorder
Inorder
Live Mutants
0
50
100
150
200
300
400


GRAIL (Execution Path) – Pointer Manipulation – Test Case 2
Textual
Preorder
Inorder
Live Mutants
0
20
40
60
80
50
100
150
200
250
300

Bibliography

[1] *Glossary of Software Engineering Terminology.* IEEE/ ANSI, 1984.

[2] A. T. Acree. *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1980.

[3] A.T. Acree, T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Mutation Analysis. Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, April 1979.

[4] H. Agarwal, R. DeMillo, R. Hathaway, Wm. Hsu, W Hsu, E. Krauser, R.J. Martin, A. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Purdue University, West Lafayette, Indiana, March 1989.

[5] W.F. Appelbe, R. A. DeMillo, D.S. Guindi, K.N. King, and W.M. McCracken. Using mutation analysis for testing ADA programs. Technical Report SERC-TR-9-P, Purdue University, West Lafayette, Indiana 47907, 1989.

[6] V.R. Basili and R.W. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Trans. on Software Engineering*, SE-13(12), December 1987.

[7] Barry Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[8] F.P. Brooks. *The Mythical Man Month.* Addison Wesley, 1980.

[9] T. Budd and F. Sayward. Users Guide to the Pilot Mutation System. Technical Report 114, Department of Computer Science, Yale University, 1977.

[10] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven CT, 1980.

[11] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. The Design of a Prototype Mutation System for Program Testing. In *Proceedings NCC, AFIPS Conference Record*, pages 623–627, 1978.

[12] T. A. Budd, R. Hess, and F. G. Sayward. EXPER Implementor's Guide. Technical report, Department of Computer Science, Yale University, 1980.

[13] T. A. Budd and R.J. Lipton. Mutation Analysis of Decision Table Programs. In *Proceedings of the 1978 Conference on Information Sciences and Systems*, pages 346–349, 1978.

[14] T.A. Budd. A Heirarchy of Test Methods. Technical report, University of Arizona, 1983.

[15] B. Choi, A. Mathur, and B. Pattinson. PMothra: Scheduling Mutants for Execution on a Hypercube. In *Procs. ACM SIGSOFT, 3rd Symposium on Testing, Verification and Analysis*, 1989.

[16] B. Choi and A.P. Mathur. Use of Fifth Generation Computers for High Performance Reliable Software Testing (Final Report). Technical Report SERC-TR-72-P, Purdue University, West Lafayette, Indiana 47907, 1990.

[17] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[18] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the Mothra Software Testing Environment. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4), April 1978.

[20] R. A. DeMillo, F. G. Sayward, and R. J. Lipton. Program Mutation: A New Approach to Program Testing. In *Infotech International State of the Art Report: Program Testing*, pages 107–126. Infotech International, 1979.

[21] R. A. DeMillo and E. H. Spafford. The Mothra Software Testing Environment. In *Proceedings of the 11th Nasa Software Engineering Laboratory Workshop*, Goddard Space Center, December 1986.

[22] R.A. DeMillo. Test Adequacy and Program Mutation. In *Procs. 11th International Conference on Software Engineering*, pages 355–356. IEEE Computer Society Press, 1989.

[23] R.A. DeMillo and A.P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software. Technical Report SERC-TR-92-P, Purdue University, West Lafayette, Indiana 47907, 1991.

[24] I.M.M. Duncan and D.J.Robson. Parameterized Mutation Testing. *Journal of Software Testing, Verification and Reliability*, 1(4), January- March 1992.

[25] I.M.M. Duncan and D.J. Robson. An Exploratory Study of Common Coding Faults in C Programs. Computer Science Technical Report 5/91, University of Durham, 1991.

[26] J.W. Duran and S.C. Ntafos. An Evaluation of Random Testing. *IEEE Trans. on Software Engineering*, SE-10(4), July 1984.

[27] A.S.C. Ehrenberg. *Data Reduction.* Wiley-Interscience, 1975.

[28] N. E. Fenton, R.W. Whitty, and A.A. Kaposi. A Generalised Mathematical Theory of Structured Programming. *Theoretical Computer Science*, 36:145–171, 1985.

[29] K.A. Foster. Error Sensitive Test Cases Analysis (ESTCA). *IEEE Trans. on Software Engineering*, SE-6(3):258–64, May 1980.

[30] P.G. Frankl, S.N. Weiss, and E.J. Weyuker. ASSET: A System to Select and Evaluate Tests. In *Int. Conf. Procs. Software Tools*, pages 72–9, April 1985.

[31] J.M. Galvin. Mutation Analysis: A User's View. In *Proceedings 7th Annual Micro-Delcon*, ISBN 0 8186 -5545, pages 30–40. IEEE Computer Society Press, 1984.

[32] M. R. Girgis. *Studies of Program Test Coverage Criteria and the Development of an Automated Support System.* PhD thesis, Liverpool University, 1986.

[33] M. R. Girgis and M. R. Woodward. An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria. In *IEEE Computer Society Press*, pages 64–73, July 1986.

[34] M.R. Girgis and M.R. Woodward. An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis. In *Procs. 8th International Conference on Software Engineering*, 1985.

[35] R.G. Hamlet. Testing Programs with Finite Sets of Data. *Computer Journal*, 20(3), 1977.

[36] R.G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, SE-3(4), 1977.

[37] R.G. Hamlet. Unit Testing for Software Assurance. In *Proc. COMPASS89*, 42-48, 1989. IEEE.

[38] D. Hanson, R. Lipton, F. Sayward, and R. A. DeMillo. Program perturbations. Technical Report, Yale University, USA, 1976.

[39] J.Z.W. Hartmann. *Structural Testing Techniques for the Selection Revalidation of Software.* PhD thesis, Computer Science, University of Durham, Science Site, Durham, U.K., 1992.

[40] M.A. Hennell, D. Hedley, and I.J. Riddell. Assessing a Class of Software Tools. In *IEEE 7th Int. Conf. Procs. on Software Engineering*, pages 266–77, arch 1984.

[41] Bill Hetzel. *The Complete Guide to Software Testing.* QED Information Sciences, Inc, second edition, 1988.

[42] C.A.R. Hoare. Algorithms 65 : Find. *CACM*, 4(1), April 1961.

[43] M.A. Holthouse and M.J. Hatch. Experience with Automated Testing Analysis. *IEEE Computer*, 12(8):33–6, Aug 1979.

[44] J.R Horgan and A.P Mathur. Assessing Testing Tools in Research and Education. *IEEE Software*, pages 61 – 69, May 1992.

[45] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *Transactions on Software Engineering*, 8(2):371–379, July 1982.

[46] W.E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–215, September 1976.

[47] D. Ince. The Validation, Verification and Testing of Software. Technical Report 84/8, The Open University, Milton Keynes, England, December 1984.

[48] D. Ince and S. Hekmatpour. An Empirical Evaluation of Random Testing. *Computer Journal*, 29(4), 1986.

[49] S. C. Johnson. Yacc : Yet Another Compiler-Compiler. Technical Report 32, Computer Science, Bell Laboratories, Murray Hill, New Jersey, 1975.

[50] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[51] D. E. Knuth. *The Art of Computer Programming*, volume 1 : Fundamental Algorithms, chapter 2. Addison Wesley, 1969.

[52] D.E. Knuth. The Errors of Tex. *Software Practice and Experience*, 19(7):607–685, July 1989.

[53] A. Koenig. *C Traps and Pit Falls.* Addison-Wesley, 1989.

[54] D.I. Korchemnyi. Application of Mutation Analysis to Evaluation of Program Testing Quality. *Programming and Computer Software*, 17:177–183, 1991.

[55] E. Krauser, A. Mathur, and V. Rego. Mutant Unification : A New Method for Mutation Testing on SIMD Machines. In *Software Engineering and Its Applications, 3rd Int. Workshop.* EC2, Nanterre, December 1990.

[56] E. W. Krauser, A. P. Mathur, and V. Rego. High Performance Testing on SIMD Machines. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

[57] M. E. Lesk. Lex - A Lexical Analyzer Generator. Technical Report 39, Computer Science, Bell Laboratories, Murray Hill, New Jersey, 1975.

[58] R. J. Lipton and F. G. Sayward. The Status of Research on Program Mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.

[59] L.J.Morell. *A Theory of Error Based Testing.* PhD thesis, University of Maryland, College Park, MD, 1984.

[60] B. Marick. A Survey of Test Effectiveness and Cost Studies. UIUCDCS-R-90-1652, University of Illinois, 1990.

[61] B. Marick. Two Experiments in Software Testing. UIUCDCS-R-90-1644, University of Illinois, 1990.

[62] B. Marick. Experience With The Cost Of Different Coverage Goals For Testing. In *Ninth Annual Pacific NorthWest Software Quality Conference*, 1991.

[63] A.C. Marshall, D. Hedley, I.J. Riddell, and M.A. Hennell. Static Data-Flow Aided Weak Mutation Analysis. *Info. Software Technology*, 32(1), Jan/Feb 1990.

[64] A. Mathur. On the Relative Strengths of Data Flow and Mutation Based Test Adequacy Criteria. In *Ninth Annual Pacific NorthWest Software Quality Conference*, 1991.

[65] A.P. Mathur. Reducing the Cost of Mutation Testing : An Empirical Study. Technical Report SERC-TR-138-P, Purdue University, West Lafayette, Indiana 47907, 1993.

[66] J.A. McDermid, editor. *Software Engineer's Reference Book.* Butterworth Heinemann, 1992.

[67] L. J. Morell. Theoretical Insights into Fault-Based Testing. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

[68] G.J. Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. *Communications of the ACM*, pages 760 – 768, Sept 1978.

[69] G.J. Myers. *The Art of Software Testing.* Wiley and Sons, 1979.

[70] S.C. Ntafos. On Required Element Testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[71] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988.

[72] A. J. Offutt. The Coupling Effect: Fact or Fiction? In *Proceedings of the Third Software Testing, Analysis, and Verification Symposium*, Key West, Florida, December 1989. IEEE Computer Society Press.

[73] A.J. Offutt. Investigations of the Software Testing Coupling Effect. *ACM Trans. on Software Engineering and Methodology*, 1(1):5–20, January 1992.

[74] A.J. Offutt and S.D. Lee. How Strong is Weak Mutation. In *Proceedings of the Fourth Workshop on Software Testing, Verification and Analysis.* IEEE Computer Society Press, 1991.

[75] A.J. Offutt, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective Mutation. In *ICSE15*, pages 100–107, Baltimore, MD, May 1993. IEEE Computer Society Press.

[76] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 1982.

[77] H.A. Priestley and M.P. Ward. A Multipurpose Backtracking Algorithm. *Journal of Symbolic Computation*, To Appear.

[78] S. Rapps and E.J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. on Software Engineering*, SE-11(4):367–375, April 1985.

[79] D.J. Richardson and M.C. Thompson. The RELAY Model Of Error Detection and its Application. In *Proc. 2nd Workshop on Software Testing, Verification and Analysis*, pages 223–230, 1988.

[80] I.J. Riddell, M.A. Hennell, M.R. Woodward, and D. Hedley. Practical Aspects of Program Mutation. Technical report, University of Liverpool, 1982.

[81] M. Sahinoglu and E. H. Spafford. Sequential Statistical Procedures for Approving Test Sets Using Mutation-Based Software Testing. Technical Report SERC-TR-79-P, Purdue University, West Lafayette, Indiana 47907, 1990.

[82] I. Sommerville. *Software Engineering.* Addison Wesley, third edition, 1989.

[83] E.H. Spafford. Extending Mutation Testing to find Environmental Bugs. *Software - Practice and Experience*, 12(2):181–189, Feb 1990.

[84] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation Analysis Using Program Schemata. In *ISSTA*, pages 139–148, Cambridge MA, June 1993.

[85] J.M. Voas. PIE : A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering*, SE-18(8):717–727, August 1992.

[86] S.N. Weiss and V.N. Fleyshgakker. Improved Serial Algorithms for Mutation Analysis. In *Int. Symp. Software Testing and Analysis*, 1993.

[87] E.J. Weyuker. The Cost of Data Flow Testing. *IEEE Trans. on Software Engineering*, 16(2):121–128, February 1990.

[88] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

[89] M.R. Woodward. Concerning Ordered Mutation Testing of Relational Operators. *Journal of Software Testing, Verification and Reliability*, 1(3):35–40, 1991.

[90] M.R. Woodward. Errors in Algebraic Specifications and an Experimental Mutation Testing Tool. *Software Engineering Journal*, 8(4):211 – 224, July 1993.

[91] M.R. Woodward, D. Hedley, and M.A. Hennell. Experience with Path Analysis and Testing of Programs. *IEEE Transactions on Software Engineering*, SE-6(3):278–285, May 1980.

[92] D. Wu, M.A. Hennell, D.Hedley, and I.J. Riddell. A Practical Method for Software Quality Control via Program Mutation. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

UNIV DURHAM