BUDD, TIMOTHY ALAN

MUTATION ANALYSIS OF PROGRAM TEST DATA

Yale University PH.D. 1980

Mutation Analysis

of Program Test Data

A Dissertation

Presented to the Faculty of the Graduate School

of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy

by

Timothy Alan Budd

May, 1980

Acknowledgments

Credit must be given to my advisor, Richard Lipton, who originated the concept of mutation analysis and encouraged me to pursue it. The second greatest influence over this work came from Frederick Sayward, who for the four years I have known him has been a great friend as well as colleague. The third member of my committee is Alan Perlis, who I want to thank for his prompt reading and careful comments, as well as for many lively and enjoyable discussions during my four years at Yale.

Other researchers on mutation analysis who have helped formulate many of the concepts used in this thesis include Professor Richard DeMillo of the Georgia Institute of Technology and Robert Hess of Yale. In particular I want to thank Bob for running most of the experiments discussed in section 4.2.3.

I have had several useful conversations with other faculty members in the department of computer science at Yale, including Larry Snyder, Dana Angluin, Dave Barstow and Drew McDermott. Their comments and criticisms have been greatly appreciated. Dana in particular had several useful comments during the preparation of my thesis prospectus.

Mary-Claire van Leunen provided detailed editorial comments on

chapters one and five, and greatly improved their readability.

The staff of the department, including Rosemary Brown, Polly Bobroff, and Mary-Claire van Leunen, were always friendly and helped in solving the many major and minor difficulties involved in a task of this magnitude.

Research Support for this work was provided in part by the National Science Foundation, the Office of Naval Research, and the US Army Institute for Research in Management Information and Computer Science.

Finally I wish to thank my wife, Beth, who always had a hug when I needed it.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

TESTING AS AN INDUCTIVE PROCESS

Practicing programmers have traditionally increased their confidence in the correct functioning of a program by running it on a few test cases. If the test cases are well chosen such confidence is justified. If, as is more likely, the test cases are poorly chosen any sense of security can be misleading, since the test cases may reveal little or no information about whether the program is correct. It is extremely uncommon, however, for any attempt to be made at evaluating the effectivness or thoroughness of a set of test cases. A major reason for this omission is the lack of a generally agreed upon metric with which test cases can be measured. It is the aim of this thesis to analyze one such metric: mutation analysis.

Since program testing proceeds from the specific observations of a program on a small number of test inputs to a general assertion concerning the programs behavior, it is an inductive, rather than deductive, process. A programmer thinks his program is correct. Typically, he then collects confirming evidence for this belief by executing the program on various test cases. As long as no test case

conclusively contradicts his belief, each additional test case increases the programmer's feeling of confidence that the program is indeed correct. At some point when a sufficient number of test cases have been run (a point often equivalent to the programmer's becoming bored with testing) the process stops and the program is deemed correct.

Inductive arguments of this kind are typical in scientific research as well as everyday discussion [17, 45]. While in everyday situations these arguments may seem quite convincing, on a formal level there are some serious difficulties. Consider the assertion:

A1: All ravens are black.

Most people would agree that observing a black raven gives us evidence for A1 whereas observing a brown shoe does not. This is because the assertion seems to be somehow about ravens and not about shoes. Logically, however, A1 is equivalent to its contrapositive:

A2: No non-black object is a raven.

Since A1 is equivalent to A2 evidence for one must be evidence for the other. But A2 seems to be making an assertion about non-black objects just as much as A1 was making an assertion about ravens. Hence a brown shoe certainly seems to be evidence for A2. We seem to be reduced to conceding that by observing a brown shoe, a green vase, a white swan, or indeed any non-black object we are somehow increasing our belief in the blackness of ravens.

This paradox is obviously foolish, but as with most paradoxes the

reason it is foolish is not so obvious. It is not because the white swans or brown shoes we observe are neither blank nor ravens; Although an ornithologist might not dream of examining a brown shoe to test Al, he might think it necessary to examine some non-black birds that look very much like ravens, although they turn out to actually belong to some other species.

An initial response to this paradox might be that while the observation of a brown shoe may logically give some evidence for Al, the weight of the evidence it conveys is negligible. The notion that not all confirming observations are alike, that some are more important than others, was absent from our description of the inductive paradigm. While seemingly quite trivial, it forces us to consider the much less obvious question of how one tells the black ravens from the brown shoes, that is, how one separates the important test cases from those that are totally uninteresting. This is the problem that mutation analysis addresses. That is, mutation analysis is a method for evaluating the effectiveness of a set of test cases for a given assertion.

The goal of testing should not be merely to force acceptance by the sheer numbers of confirming examples, but rather to provide reasons why the particular assertion (program) being proposed is to be believed. In this respect the observation of a single black raven gives more evidence for Al than the observation of five thousand brown shoes.

The method used in mutation analysis to evaluate test cases is to construct a number of alternative assertions that are semantically close to the original, and then to ask if the test data distinguishes the two assertions (in the sense of being confirming evidence for the original and contradicting evidence for the alternative). Borrowing a name from biology, we say these alternatives are mutants of the original assertion.

For example, we may admit that observing a brown shoe does give evidence for A1, but it also gives evidence for the following two assertions

A3: All ravens are yellow.

A4: All birds are black.

Since on the weight of this evidence alone we therefore do not have any more confidence in A1 than in A3 or A4, our faith in the ability of this test data to confirm A1 is severely weakened.

In order to differentiate A1 from A3, we require a test observation that is a raven and not yellow. Such an observation will confirm A1 (if the raven is indeed black) and contradict A3, thereby showing how the two assertions differ. Similarly, to differentiate A1 from A4 we require a bird that is not black, and presumably therefore not a raven. This supports the intuitive feeling that the observation of a black raven or a white swan gives greater evidence for A1 than the observation of a brown shoe.

A more serious problem arises when we attempt to differentiate A1 from the assertion

A5: All members of *corvus corax* are black.

A5 is equivalent to A1, but that fact is evident only to those who happen to know the genus to which ravens belong. The problem of recognizing those mutants that are equivalent to the original assertion is the major impediment to the mutation analysis method. In practice however, as will be demonstrated in chapter four, this is not as serious a problem as might at first be imagined.

1.1 Testing computer programs

We are given a program P and a set of test cases that the programmer believes sufficient. As before, we now ask whether these test cases increase our confidence only in this particular program, and not some closely related but distinct programs, the so-called mutants of P. We generate a set of mutants, forming each by altering the program P in a simple way. These mutants are then run on the test data to see whether they receive the same answers as P. If a large number of mutants produce the same results we assert, as in the case of the ravens, that this test data gives us no more reason to believe the original program is correct than that any of the mutants are correct. A test set that fails to differentiate P from a large number of mutants is like the observation of a brown shoe, which gave us no more evidence that all ravens are black than that they are all yellow. On the other hand, if a majority of the mutants are eliminated, then the test data closely fits P, and our confidence in the effectiveness

of the test cases is increased.

At first blush this might seem an amusing methodology, useful for catching misspelled variable names and incorrect operators, but perhaps not worth all the fuss. To see that this is not the case note that there has been observed in many problem domains a strong coupling between simple and complex errors [11, 22, 29]. Test data that eliminates all non-equivalent mutants can be thought of as insuring the absence of simple errors (i.e. those errors that exactly match the mutation). This coupling of errors implies that while one is directly meeting the mutation goal, one is peripherally achieving a much stronger goal, that of differentiating P from a larger class of complex potential errors. This phenomenon has been called the coupling effect. The name coupling effect comes not only from the hypothesis that complex errors are coupled to simple ones, but also from the observation that different sections of one program are often strongly coupled together. Sayward has elsewhere given a statistical argument for belief in this coupling [65]. Chapters two and four will contain various theoretical arguments and empirical studies that address this belief. What follows will be an intuitive justification.

Programs are highly interrelated objects. Actions taken in one location will usually have a major effect on the actions taken in another section. A correct program is one that is consistent, one in which every action is neatly tailored to fit every other action, much like a solid wall of bricks. An incorrect program is like a badly

formed brick wall: One brick is out of place, and it pushes another out of place, and that one still another, and so on until the entire wall is in jumbles. Mutation analysis uniformly exercises every part of the program, pushing it to its boundary conditions. Even if the original cause of an error cannot be discovered, its effects often can be. In this manner complex errors are uncovered by simple means.

What about those programs that are internally consistent, but consistently wrong? Fortunately we know by numerous studies [24, 80, 86, 89] that these degenerate cases are rare. More specifically, we have an assumption that states:

> A competent programmer, after giving the task sufficient thought and pursuing the normal process of programming and debugging, has probably written a program that is either correct or "almost" correct, in that it differs from a correct program in simple ways [22].

The name competent programmer hypothesis has been given to this assumption, since one way to interpret it is to say that incompetent programmers (those who write vastly incorrect programs) are quickly discovered by running their programs on almost any test data. Most programmers are neither malicious nor incompetent, and they can be expected to produce a product that is at least approximately correct. Our task therefore is reduced to validating a program that is probably not correct, but is very close.

This is the framework for mutation analysis. We have a program P written by a competent programmer. We are given a set of test cases

which differentiate P from a small set of mutants derived from P. The coupling effect asserts that this test set in fact differentiates P from a much larger class of programs, throughout this thesis denoted ϕ. The competent programmer hypothesis asserts that with high probability either P is correct or some program in ϕ is the correct program. The task placed before the programmer is to try to generate test cases that distinguish all non-equivalent mutants. The coupling effect asserts that if the program is indeed incorrect, then the programmer cannot succeed at this task. That is, there is at least one non-equivalent mutant that cannot be eliminated without generating a test case that is incorrectly handled. Conversely, if the programmer can find test data that differentiates P from all non-equivalent members of ϕ, this means that the only member of ϕ that receives the correct answer on all these test cases is P itself. We can therefore conclude with high likelihood that P is correct.

There are two very different directions in which one could proceed from here. The first would be to give very precise definitions to the terms "close,""mutants," "competent programmer hypothesis," "coupling effect," which have been used up to now quite informally. Using these precise terms, one could then formally prove that a theorem similar to the coupling effect holds. Several studies of this nature are developed in chapter two. Unfortunately it becomes quickly apparent that the range of programs one can deal with in this manner is severely restricted. This is due not only to the difficulty

in dealing with complex data and control structures, but also to a plethora of formally undecidable properties [18, 31, 40, 52].

The other direction one can take is to let the more intuitive definitions of our basic terms suffice and to ask whether in some empirical or statistical sense the coupling effect holds. Studies of this nature are developed in chapter four. Given that these studies deal with the type of programs programmers write every day, they are in some sense of greater importance to the practitioner of software testing than the theoretical studies, even though the results derived are not formal but are only empirical. This schism between theoretical and empirical studies appears to be unreconcilable, and this dichotomy is reflected in the arrangement of this thesis. Although distinctly different, both types of studies are important in advancing our understanding of the process of program testing.

1.2 Error seeding, circuit theory and other metrics

It has been observed that, in its application, the mutation analysis method bears a certain superficial resemblance to the error seeding method of Mills [69] (also called 'bebugging' by Glib [36]). To motivate this method, Mills cites Feller's well known text on statistics [26]:

> Suppose that 1000 fish are caught in a lake and marked by red spots and released. After a while a new catch of 1000 fish is made, and it is found that 100 among them have red spots... We assume naturally that the two catches may be considered as random samples from the population of all fish in the lake... These figures would justify a bet that the true number of fish lies somewhere between 8500 and 12,000. [pages 43-44]

In error seeding the red fish correspond to artificially inserted errors. Given that a certain number of errors are discovered during testing, the ratio of artificial to natural errors that have been uncovered should (in theory) give some indication of the number of natural errors that remain in the program.

There are several points that can be raised here. First, note carefully the range of numbers given in the text from Feller; few software projects are large enough to contain 10,000 errors. In the more typical range for software errors the loss of significance will almost certainly render any estimates meaningless. Second, note the assumption that the fish (errors) are uniformly distributed. In software this is almost certainly not the case, since long sections of code are often simple minded, and interspersed with short bursts of complex, error prone computation. Since the natural errors are unknown we also have no assurances that a seeded error will not interact with a natural error in such a way that the effects of the natural error are canceled by those of the seeded error, so that the seeded error actually disguises the natural one.

Finally, note the assumption that the seeded errors (red fish) are identical, in terms of their distribution and appearance, with the more general population. While we have, for large classes of programs and programmers, some statistical ideas concerning the nature of those errors most often produced [24, 86], in any single program the errors are likely to be sporadic, nonuniform, and highly unpredictable.

Furthermore it is not clear how this statistical information can be used to generate artificial errors with the desired properties; if a large number of very subtle errors are caused by the programmer's using a wrong variable name this does not imply the converse, that a randomly changed variable name will result in a subtle error. In fact the experiments to be discussed in chapter four would lead us to believe that most of these errors would be quite obvious. At least one study in error seeding [23, 47] reached exactly this same conclusion, namely that the seeded errors were much easier to detect than the natural ones. It would seem that generating artificial errors with the same features and likelihood of detection as natural ones would be an intractable problem.

In mutation analysis we need assume nothing about statistical distribution of the errors in a program. Mutants are not examples of potential errors; they merely question whether the test data for a program is sensitive to changes in the program's structure. Furthermore, the failure to differentiate the program from a specific mutant points directly and unambiguously to a weakness in the test data. In error seeding, even assuming all the seeded errors are discovered, no such information is available.

It is mentioned by Girard and Rault [35] that the error seeding technique can be used to "assess the 'detecting power' of test cases generated randomly." This is similar to the goal of mutation analysis, but it does not avoid the pitfalls of error seeding described here.

Mutation analysis is much more closely related to the classical method for detecting logical faults in digital circuits. Many physical faults, such as short or open circuits, manifest themselves on a logical level as circuit lines being stuck either in the high or the low position. The single fault testing model [8] therefore considers each mutant formed by assuming a single input line is stuck and constructing test data to detect this condition. The number of test inputs required to achieve this goal is usually minuscule in relation to the exponential number of inputs required to test the circuit exhaustively. Furthermore there is an assumption analogous to the coupling effect that states that in practice single fault test sets are relatively good for the detection of multiple faults. For the most part, this version of the coupling effect is statistical, or based on previous experience [1]. However a paper by Ostapko [72] studies a device known as a programmable logic array (PLA) and shows that for these objects a very strong coupling effect can be formally proved.

Where mutation analysis and single fault testing differ is in their goals. Mutation analysis is concerned with evaluating test data and validating the initial design of a computer program. Single fault test sets check circuit deterioration at intermittent times during the life of a device. Otherwise the analogies between the two methods are close.

Prior to mutation analysis there were few attempts to formulate

metrics to measure the effectiveness of test cases. A method that gained a certain popularity was merely to count the number of distinct statements executed by a collection of test cases and express this as a percentage of the total number of statements in the program [68, 82]. This counting technique was later strengthened to the requirement that every predicate should take on its entire range of possible outcomes during a test [58]. There exist now several commercial systems that provide this type of information [41, 74, 82, 83]. There are even some systems that attempt automatically to generate test cases that satisfy these requirements [20, 76]. Although easily computed, the figures derived by merely counting either statements or branches are now generally viewed as providing too little information for an accurate judgment to be made concerning program correctness [34, 47, 52]. As will be shown in chapter three, mutation analysis subsumes the goals of these and several other testing techniques.

1.3 Literature and related work

The term "paradoxes of induction" and the example of the ravens given in section one was first used by Hempel [45]. In response to Hempel's paradox various solutions have been proposed in the literature. One solution, based on Bayesian principles, seems to be widely accepted [85]. This approach states that since non-black objects are evidently much more common than ravens, a raven is much more likely to provide us with a counterexample than a non-black

object; hence the observation of a raven is more significant. On a practical level Shortliffe [81] uses this bayesian approach in an automated system designed to assist physicians in making decisions on the basis of inconclusive evidence, an induction problem using past experience as the test observations. In this case the problem is deciding on an appropriate therapy for patients with infections.

As defined in the first section, the goal of testing is to differentiate a program from a small set of similar programs. Howden [51] briefly studied the notion of testing programs in the context of a small class of alternatives (which he termed the model set). His framework was much more highly structured than that of the present work, and his idea still seemed to be that test cases provided direct evidence for the program's being correct, rather than giving reasons for ruling out alternatives in the model. In a later work [55] he again approached the testing problem from the point of view of eliminating alternative programs. This work, although developed independently and presented in a different manner, is in many respects quite similar to the ideas developed in this chapter.

Reduced to its simplest form, the coupling effect asserts that while one is guarding against simple errors, complex errors are also detected. It has already been mentioned that a similar type of observation is made with respect to logical devices [8, 72]. Fosdick and Osterweil [29] have noted that the detection of data flow anomalies (which are one form of simple error) is often a powerful

tool in detecting other types of errors.

Richard Hamlet has also explored the notion of testing as a process of differentiating the given program from a set of alternatives [42], and in fact has constructed a system to perform this analysis [40, 41]. His framework is similar to mutation analysis, but lacks the coupling effect; hence his system must examine significantly more alternatives than the mutation analysis system. Hamlet's system will be described in more detail in section 3.5.

One way to view the goal of testing is that one is attempting to find test data that characterizes a given program. If it is possible to construct such data, then it should be possible to go the other way, that is, to automatically construct the program given characteristic examples of its input/output behavior. This problem of automatic programming has been extensively studied [4, 78, 84], but researchers seem to devote little attention to the question of when such characterizing data exists. Thus there seems to be little of help here for work on program testing.

Finally, there is a large body of literature concerned with characterizing the types of sequences that are recognizable by machines of different complexity classes. This work typically deals with more abstract types of machines, such as Turing machines or total recursive programs. A second feature of this work is that it deals with identification in the limit, that is, it assumes the presentation

of an infinite number of inputs. Although using both computers and induction, these two assumptions prevent us from directly applying the results developed in this manner to the problem of program testing. Examples of this work are: Blum and Blum [5], Angluin [2], and Kugel [62].

CHAPTER 2

THEORETICAL STUDIES

In chapter one a metric for evaluating test cases was proposed which, if applicable, makes an assertion which is valid only in a statistical sense concerning the likelihood of the program being correct. This chapter will demonstrate that in some abstract models of computation one can use test data which satisfies the mutation analysis metric to formally prove the correctness of programs.

To do so we must first give slightly more formal definitions to the basic concepts introduced in chapter one: Assume we have a large class of programs $\mathcal{P}$ which is our universe of discourse (examples might be finite state acceptors, partial recursive functions, deterministic push down automata). We are given a specific program P which is a member of $\mathcal{P}$. We will use {P} to denote the function computed by P, but use P(X) (rather than the more verbose {P}(x)) to denote the result of evaluating the function {P} on the input X. (The {P} notation is due to Kleene [61]).

To ask if P is correct is equivalent to asking whether P realizes an intended (but unknown) function F. We must of course have some

limited knowledge of F for the problem to be feasible. There seem to be two general forms that this knowledge could take:

1. We have F in some inefficient or non procedural language and the question is to decide the equivalence of the program P and the representation of F.

2. The only knowledge we have of F is the ability to compute F(x) for any finite number of values.

Option one seems to be too strongly tied to the form of the representation of F, therefore this thesis will only consider option two.

We will formalize the competent programmer hypothesis by having a subset of $\mathcal{P}$, denoted Φ, of programs "close" to P and assuming that P is a member of Φ and furthermore there is some member of Φ (though perhaps not P) which realizes F. The mutants of P (denoted ω) are a subset of Φ. In each example cited in this chapter there will be an obvious recursive procedure which generates the mutants from the program P. Both the sets Φ and ω may depend upon P.

The purpose of testing is to find a finite set of test inputs, which we shall denote D, which possess the property that if P executes correctly on these inputs then P is "correct", that is, realizes F. But we must also do this using only the limited knowledge of F which has been outlined. In order to remove any extraneous details from the problem, we define the following two notions:
Given a finite set of test cases D and a set of programs S, we say that D <u>differentiates</u> P from S if for any program Q in S, if for each

value x in D P(x) = Q(x) this implies {P}={Q}.

(formally, $\forall Q \in S$ ($\forall x \in D$ P(x) = Q(x)) => {P} = {Q})

We say that a finite set of test cases D is <u>adequate</u> if it differentiates P from Φ.

Given the formalization of the competent programmer hypothesis which we are here using, an adequate set is certainly a sufficient condition for correctness. One can argue that it is also a necessary condition as follows: Consider testing to be a game between the person proposing the test inputs and an adversary. Instead of a single correct function F, the adversary keeps a set of "possible" correct functions. Initially this set contains all the functions realized by programs in Φ. Each time a request is made for the value of F at a specific point (and remember this is the only knowledge we have of F) the adversary chooses one of the functions in this set, returns its value, then eliminates all functions in the set which do not have the same value at that point. However D is constructed, if it is not adequate then by definition there are at least two non equivalent functions in the adversary's set. Both are realized by programs in Φ, and both these programs give the same answers on D. Only one can be equivalent to P. P being proposed as "correct" the adversary can then produce the other program as the true function F.

Since it is both a necessary and sufficient condition, this notion of adequate seems to formally capture the intuitive concept of

correctness.

The coupling effect one would like to demonstrate is that any finite set of test cases D which differentiates P from ω will also differentiate P from Φ, or to rephrase, any set D which differentiates P from ω is adequate.

First a number of easy examples will be presented which give a feeling for the type of results we are interested in, and which set the stage for later developments. For example, the existence of such a set D may depend upon a careful choice of the set Φ. Remember that we allowed the set Φ to depend on P. For example if P is a finite automaton and we let Φ be the set of all finite automata then we know that no matter what our set ω is an adequate set of test inputs cannot exist. On the other hand if we let Φ be the set of finite automata with at most one more state than P then not only does D exist but there is a recursive procedure to generate it [19].

It is possible to show examples where the question of whether an adequate set exists is formally undecidable: Assume we have some fixed Godel encoding of all partial recursive functions [77]. Let $P_i(x)$ be zero except in the case where the partial recursive function i converges given input i in exactly x steps, when it takes on the value one. Each program therefore is recursive and is nonzero on at most one input.

Lemma 1: (a) If P_i is the constant function zero then there does not exist any adequate set. (b) If P_i is not the

constant function zero, then any set which contains the input which causes the program to be nonzero is adequate, and no other set is adequate.

Proof:(a) Given any finite set D, we can find a recursive program which halts in some number of steps not contained in D, and therefore D fails to differentiate these two programs. (b) Let X be the input which causes P_i to be nonzero. If P_j is nonzero, then $P_i=P_j$. Otherwise it is obvious that $P_i \neq P_j$. If a set D does not contain X, then the same argument used in part (a) shows that D cannot be adequate.

Note that for any P_i, there exists an input X for which P_i returns a nonzero value if and only if the recursive function i halts.

Theorem 2: If for an arbitrary program P_i one can decide whether there exists an adequate set, one can tell whether the partial recursive function i will halt on input i.

Since the latter problem is undecidable [77], so is the question of whether an adequate set exists.

2.1 General remarks on adequate test sets

In this section we will only be interested in situations where an adequate set is known to exist. Given this assumption, there are two further questions we might ask.

1. Does there exist a recursive procedure to recognize such a set?
2. Does there exist a recursive procedure to generate such a set?

It is possible to make some very general observations, for

example for any finite set of programs there always exists an adequate set D, hence the major question is whether it can be generated or recognized. If one assumes all programs are recursive one can prove the following two general theorems. Assume, without loss of generality, that the inputs to the programs consist of a single positive integer.

Theorem 3: There exists a procedure to generate an adequate set if and only if there exists a procedure to recognize such a set.

proof: Notice that any set which contains an adequate set is itself adequate. If we have a procedure to recognize an adequate set then we can construct a procedure which generates such a set merely by repeatedly asking if the set composed of the numbers 1 through N is adequate for larger values of N. Since some set must eventually be adequate, this process must eventually halt.

On the other hand if we have a procedure to generate an adequate set then to recognize another set as adequate one merely asks if the second set divides ϕ into the same two groups as the generated set does.□

Theorem 4: There exists a procedure to generate a set D which differentiates P from a set of programs S if and only if the equivalence problem for P and each program in S is decidable.

proof: If we can generate such a set D then to decide the equivalence of P and any element in S is merely a matter of running both programs on each member of D and checking that the two answers

agree.

On the other hand, if one can decide the equivalence of P and any program in S then divide the set S into two parts: those equivalent to P are ignored. For those programs not equivalent to P dovetail all inputs until we find a value on which the two programs disagree. Since we know the programs are not equivalent this procedure must eventually halt with a set of test cases D which is by definition adequate.□

Theorem 4 formalizes the comment made in chapter one that the problem of deciding the equivalence of a program and its mutants is a major obstacle in theoretical studies. It might seem that the results of theorem 4 are discouraging, since the equivalence question for most commonly studied language classes are undecidable [50]. There seem to be two possible methods to circumvent this impasse:

1. Carefully define Φ so that the equivalence problem is decidable.
2. Merely assume that we have a procedure to decide equivalence, thus avoiding the question.

To pursue the first solution would lead us far into language theoretic issues which are totally removed from the problems of testing. Therefore in all the examples in this thesis I have chosen the second solution.

2.2 Decision Tables

Decision Tables* are a highly structured way of describing decision alternatives. Such tables are chiefly used in business and data processing applications [70, 75], although they can also be used to organize test data selection predicates [37].

To form a decision table we need a set of conditions, a set of actions, and a table composed of two parts. Entries in the upper part are from the set {YES, NO, DON'T CARE} (denoted Y, N, and *); entries in the lower table are either DO or DON'T DO (denoted X or 0). Each column in the matrix is called a rule. An example with four rules is shown in figure 2-1.

	1	2	3	4
condition 1	Y	Y	N	*
condition 2	N	*	Y	Y
condition 3	*	Y	Y	N
condition 4	N	Y	*	*
action 1	X	X	0	X
action 2	X	0	0	0
action 3	0	0	X	X

Figure 2-1: A typical decision table program

To execute the program on some input the conditions are first simultaneously evaluated, forming a vector of YES, NO entries. This vector is then compared to every rule. If the vector matches any rule the indicated actions are performed. It is assumed that either the actions are commutative or there is a given order of their

*This work was originally published in the proceedings of the 1978 Johns Hopkins Conference on Information Sciences and Systems [12]. The presentation given here has been greatly simplified and expanded.

application. If for each feasible input there is at least one rule that can be satisfied we say the decision table is complete. We say it is consistent if there is at most one rule [70]. We will assume the program under test is consistent. We can also assume it is complete, since an incomplete decision table can always be turned into a complete one by adding additional actions which merely return an error flag and additional rules which are satisfied by the previously unmatched inputs.

We will also assume that no two rules specify exactly the same set of actions. We do this with little loss of generality since two such rules can be combined into a single rule with at most the addition of one new condition.

Given a decision table program P let ϕ be the set of all consistent programs having the same conditions and actions as P. This means members of ϕ differ from P only in the table portions, or by having a different number of rules.

The mutants (ω) of P will be those members of ϕ which are formed by taking a single * entry and changing it into a Y and a N entry, respectively. If P is consistent then all the mutants will be consistent. Some of these mutants may be equivalent to P. The mutant which changes position j in rule i from a * to a Y is equivalent to P only if it is impossible for any input to satisfy rule i and not satisfy condition j.

There are two mutants for every * entry in P. This means there are no more than 2nm mutants, where n and m are the dimensions of the table. Each mutant requires at most a single test case to differentiate it from P. Even though there are potentially 2^n different inputs, an adequate mutation set need only have at most 2nm inputs.

We will make the following assumptions:

1. The decision table P is both consistent and complete, and all members of Φ are consistent.
2. Given any program in Φ, if we are given an example of input/output behavior we can determine which rule was applied to produce the output from the given input. In particular this implies that no two rules specify exactly the same set of actions.
3. There exists at least one input which satisfies each rule in P.
4. It is possible to decide the equivalence of P and any member of ω.

The results given here will demonstrate that any set of inputs which differentiates P from ω in fact differentiates P from Φ. Assume we have such a set D. Assume that each rule in P is satisfied at least once by some member of D, adding test inputs if necessary to meet this condition. We can initially fail to meet this condition only if there are some rules which do not contain *'s. Note that we could have guaranteed the satisfaction of every rule with mutants if we also mutated the action matrix, as was done in the original paper [12]. The present exposition seems to be simplified without

loss of generality by the elimination of this step.

Given any program Q in Φ, if for each x in D P(x) = Q(x) then we will say Q tests equal to P. Since each rule in P has a unique set of actions by a simple counting argument we know that if Q tests equal to P then for each rule in P there is a corresponding rule in Q with exactly the same actions. Using this fact, the following theorem can be demonstrated:

> **Theorem 5:** If D differentiates P from ω and Q tests equal to P, then for each rule in P the set of inputs satisfying the corresponding rule in Q is strictly larger than that of P.

proof: First note that it is not possible for a rule to have a Y entry in P and for the corresponding rule in Q to have an N, or vice versa. If this were so no data which satisfied the rule in P could satisfy the rule in Q.

Now consider each * entry in P. There are two cases. If the change which replaces this * by a Y (the same argument holds for N) is equivalent, this means the conjunction of the other conditions implies a YES in this position. In this case it doesn't matter whether the corresponding rule in Q has a Y or a * (and these are the only two possibilities) this change cannot contribute to decreasing the size of the set of inputs accepted by the rule in Q.

On the other hand if this change is not equivalent, D contains points which while satisfying the rule both satisfy and fail to

satisfy this particular condition. Both these must be accepted by the same rule in Q. Therefore Q must also have a * in this position.

The only remaining possibility is that some rule in P has a Y (or N) and the corresponding position in Q has a *. This strictly increases the size of the set of inputs accepted by this rule, giving the result.□

Theorem 6: If test data (D) executes every rule and differentiates P from ω, then D differentiates P from ϕ.

proof: Let Pi be the set of inputs accepted by rule i in P. Since P is consistent, the Pi are disjoint. Since P is complete, they cover the entire space of inputs. Each corresponding rule in Q must accept at least the set accepted by the rule in P. Since Q is consistent, it can satisfy no more. □

2.2.1 Extensions and restrictions

Notice the assumption that P is complete is not used in the proof of theorem 5. This might lead one to suspect that this restriction could be removed. To see that this is not the case, consider the three decision tables each consisting of a single rule, shown in figure 2-2.

	P	M	Q
condition one	Y	Y	*
condition two	*	Y	Y

Figure 2-2: Example showing completeness is necessary in theorem 6

Assume the satisfaction of condition one implies the satisfaction of condition two. This means the program P is equivalent to its

mutant M, therefore the only test case we will generate will be one which merely executes the rule (say YY). Now consider the program Q. Theorem 5 states that any input which satisfies the rule in P must satisfy the rule in Q, and indeed this is the case. The converse, however, is not true, as the input NY shows.

Testing an inconsistent program is rather like looking for missing path errors (see section 4.2.3), in that for both cases the testing method must attempt to guess at something which should be, but is not, contained in the program. Given the great deal of uncertainty involved in this, it should not be surprising that the difficulties involved in testing these programs are considerable.

We can, however, replace the assumption the P is complete with two weaker hypotheses: 1) that all the conditions are independent, that is, that all 2^n possible inputs are feasible. (Notice this implies that none of the mutants constructed in the last section can be equivalent). and 2) that no program in Φ can have more rules than are in P.

We can create a new type of mutant, by replacing each Y or N entry with a *, as long as by doing so we do not create an inconsistent program. Notice that if P is complete, all such mutants create inconsistent programs.

Theorem 7: If test data (D) executes every rule and differentiates P from ω, then D differentiates P from Φ.

proof: Theorem 5 still applies, and in fact can be strengthened, since there are no equivalent mutants of the first type. This means that for every * entry in P, the corresponding position in Q must also have a *. The only possible way to have an inequivalent program Q is for there to be some input X which satisfies rule j in Q but no rule in P. Let rule i be the rule in P with the same actions as rule j (a simple counting argument shows such a rule must exist). The only possible way for X to satisfy rule j and not rule i is for there to be a * entry in rule j and a Y (or N) in rule i which X fails. This means that the mutant which effects this same transformation cannot produce an inconsistent program (if rule i conflicted with some other rule in P it would have to conflict with the corresponding rule in Q). But therefore there must be some input which differentiates it from P. This means there is some input which is is rejected by i but satisfies the mutated rule. But this input must also satisfy rule j, contradicting the fact that Q tests equal to P.□

Notice that if the test sets used in theorem 6 and 7 exist they are very small. As noted earlier the number of test cases is linear in the size of the program, even though the number of inputs may be exponential.

Recall that theorem 4 implied we could form an adequate mutation set only if we could decide equivalence of P and each of its mutants. Obviously there are some cases where this is true, for example when all the conditions are independent and therefore none of the mutants

are equivalent. We can easily find examples where this is not true. This is the case whenever we have two conditions where the question of whether the first condition implies the second is undecidable.

condition 1	Y
condition 2	*

Figure 2-3: An example where equivalence is undecidable if the conditions interact

We can replace the * in the condition 2 row with a Y if and only if condition 1 always implies condition 2. In this fashion using any classic undecidable problem [50] we can construct a program with the property that the equivalence question for it and one of its mutants is undecidable.

An assumption made in proving theorem 6 was that each rule had a distinct set of actions. We avoided the question of what happens if this is not the case by using the device of combining two such rules into one and adding a new condition entry. But this is simply moving part of the table into the condition entries, which we then proceed to assume are correct. To see that there are errors which might not be detected in this manner consider the two decision tables shown in figure 2-4. The two programs are not equivalent (they process the input NNYN differently) yet they agree on the set of test inputs {NNYY, NYYN, YYNN, YNNY, NNNN, NYNY, YYYY, YNYN}, which is sufficient to eliminate all the mutants of program 1.

It is not clear whether the restriction to rules having distinct actions can be replaced with a weaker assumption, or if there is any

PROGRAM 1				PROGRAM 2			
N	Y	N	Y	*	*	*	*
*	*	*	*	N	Y	N	Y
Y	N	N	Y	*	*	*	*
*	*	*	*	Y	N	N	Y
X	X	0	0	X	X	0	0
0	0	X	X	0	0	X	X

Figure 2-4: A case not covered by the mutation test

test method which can be used to demonstrate correctness in this case, other then trying all 2^n possible inputs.

2.3 LISP programs

This section will consider programs written in the subset of LISP containing the functions CAR, CDR and CONS and the predicate ATOM*. A similar class of programs has been studied previously [44, 78, 84].

Associated with each S-Expression X we can construct a binary tree, which represents the structure of X. Call this tree the projection of X.

We will define a relation <= as follows. Given two S-expressions X and Y we will say X <= Y if the projection of X is equal to the intersections of the projections of X and Y.

We will make the convention that all S-Expressions (from now on we will use the less clumsy expression point) have unique atoms. Certainly if two programs agree on all such points they must agree on

---

*An earlier version of this section appeared in a paper presented at a workshop on software testing and test documentation [13]

all inputs, hence we can do this without loss of generality.

2.3.1 Straight line programs

We will call a LISP program a selector program if it is composed of just CAR and CDR. We will inductively define a straight line program as a selector program or a program formed by the CONS of two other straight line programs.

Theorem 8: If two selector programs return identical values on any input for which they are both defined, they must compute identical values on all points.

proof: The only power a selector program has is to choose a subtree out of its input and return it. One can view this as simply selecting a position in the complete CAR/CDR tree and returning the subtree rooted at that position. Since there is a unique path from the root to this position, there is a unique predicate which selects it out. Since atoms are unique by merely observing the output one can infer the subtree which was selected. □

We will say a straight line program P(X) is well formed if for every occurrence of the construction CONS(A,B) it is the case that A and B do not share an immediate parent in X. The intuitive idea of the definition should be clear: a program is well formed if it is not doing any more work than it needs to. Notice that being well formed is an observable property of programs, independent of testing.

We can define a measure of the complexity of a straight line program as follows:

1)The CONS-depth of a selector program is zero.

2)The CONS-depth of a straight line program
P(X) = CONS(P1(X),P2(X))

is
1+MAX(CONS-depth(P1(X)),CONS-depth(P2(X))).

Lemma 9: If two well formed selector programs compute identically on any point for which they are both defined, then they must have the same CONS-depth.

proof: Assume we have two programs P1 and P2 and a point X such that P1(X) = P2(X) yet the CONS-depth(P1) < CONS-depth(P2). This implies that there is at least one subtree in the projection of P2 which was produced by CONSing two straight line programs while the same subtree in P1(X) was produced by a selector. But then the objects P2 CONSed must have an immediate ancestor in X, contradicting the fact that P2 was well formed.□

Theorem 10: If two well formed straight line programs agree on any point X for which they are both defined, then they must agree on all points.

proof: The proof will be by induction on the CONS-depth. By lemma 9 any two programs which agree at X must have the same CONS-depth. By theorem 8 the theorem is true for programs of CONS-depth zero. Hence we assume it is true for programs of CONS-depth n and show the case for n+1.

If program P1 has CONS-depth n+1 then it must be of the form CONS(P11,P12) where P11 and P12 have CONS-depth no greater than n. Assume we have two programs P1 and P2 in this fashion. Then for all Y:

P1(Y) = P2(Y) IFF
CONS(P11(Y),P12(Y)) = CONS(P21(Y),P22(Y)) IFF
P11(Y) = P21(Y) and P12(Y) = P22(Y)

Hence by the induction hypothesis P1 and P2 must agree for all Y.

□

One can easily generalize theorem 10 to the case where we have multiple inputs. Recall that each atom is unique, therefore given a vector of inputs we can form them into a list and the resulting structure will be a single input with unique atoms. Similarly a program with multiple arguments can be replaced by a program with a single argument by assuming the inputs are delivered in the form of a list, and replacing references to argument names with selector functions accessing the appropriate positions in this list. Using this construction one can verify that if theorem 10 did not hold in the case of multiple arguments, we could construct two programs with a single argument for which it did not hold, giving a contradiction.

To summarize this section, for any well formed straight line program, any unique atomic point for which the function is defined is adequate to differentiate the program from all other well formed straight line programs.

2.3.2 Recursive programs

The type of programs studied in this section can be described as follows:

The input to the program will consist of selector variables,

denoted $x_1, \ldots x_m$ and <u>constructor variables</u>, denoted $y_1, \ldots y_p$. A <u>program</u> will consist of <u>program body</u> and a <u>recurser</u>. A <u>program body</u> consists of n statements, each statement composed of a <u>predicate</u> of the form ATOM($t(x_i)$) where t is a selector function and x_i a selector variable, and a straight line <u>output function</u> over the selector and constructor variables. A <u>recurser</u> is divided into two parts. The <u>constructor</u> part is composed of p assignment statements for each of the p constructor variables where y_i is assigned a straight line function over the selector variables and y_i. The <u>selector</u> part is composed of m assignment statements for the m selector variables where x_i is assigned a selector function of itself.

The example in figure 2-5 should give a more intuitive picture of this class of programs.

Program $P(x_1,\ldots x_m,y_1,\ldots y_p)$ =
IF $p_1(x_{i1})$ THEN $f_1(x_1,\ldots x_m,y_1,\ldots y_p)$
ELSE IF $p_2(x_{i2})$ THEN $f_2(x_1,\ldots x_m,y_1,\ldots y_p)$
...
ELSE IF $p_n(x_{in})$ THEN $f_n(x_1,\ldots x_m,y_1,\ldots y_p)$
ELSE
$y_1 := g_1(y_1,x_1,\ldots x_m)$
...
$y_p := g_p(y_p,x_1,\ldots x_m)$
$x_1 := h_1(x_1)$
...
$x_m := h_m(x_m)$
$P(x_1,\ldots x_m,y_1,\ldots y_p)$

Figure 2-5: An example recursive program scheme

Given such a program, execution proceeds as follows: Each predicate is evaluated in turn. If any predicate is undefined so is the result of the execution, otherwise if any predicate is TRUE the result of execution is the associated output function. Otherwise if no predicate evaluates TRUE then the assignment statements in the recurser and constructor are performed and execution continues with these new values.

the following restrictions we be assumed:

1. All the recursion selector and recursion constructor functions must be non trivial (i.e. of depth one at least, so that $x_i := x_i$ is ruled out).
2. Every selector variable must be tested by at least one predicate.
3. There is at least one output function which is not a constant.
4. (freedom) for each 1<=k<=n and m>=0 there exists at least one input which causes the program to recurse m times before exiting with output function k.
5. Each constructor variable appears totally in at least one output function.

Let ϕ be the set of all programs with the same number of selector and constructor variables as P, the same number of predicates, and output functions no deeper than some fixed limit olimit. Our goal is to construct a set of test cases (D) which differentiates P from all members of ϕ. The mutants of P (ω) will be defined as they become important to the proof. The argument will proceed in several small steps:

In section 2.3.2.1 basic definitions are given and some tools which will be used in later sections are derived. Section 2.3.2.2 shows how to use testing to bound the depth of the selector functions. In section 2.3.2.3 we narrow the form of the selector functions still further, finally in section 2.3.2.4 showing they must exactly match P. Section 2.3.2.5 deals with the points tested by the predicates, and in section 2.3.2.6 the main theorem is given. Section 2.3.3 concludes with some comments on the difficulty of proving a program correct in this manner.

2.3.2.1 Definitions and tools

Capital letters from the end of the alphabet (X, Y and Z) will be used to represent vectors of inputs. Hence we can refer to P(X) rather than $P(x_1,\ldots,x_m,y_1,\ldots,y_p)$. Similarly we will abbreviate the simultaneous application of constructor functions by C(X) and recursion selectors by R(X).

Letters from the start of the alphabet will be used to represent positions in a variable, where a position is defined by a finite CAR-CDR path from the root. When no confusion can arise we will frequently refer to "position a in X" whereby we mean position a in some x_i or y_i in X. We will sometimes refer to position b relative to position a, by which We mean to follow the path to a and starting from that point follow the path to b.

The depth of a position will be the number of CARs or CDRs

necessary to reach the position starting from the root. Similarly the depth of a straight line function will be the deepest position it references, relative to its inputs. Let maxd be the maximum depth of any of the selector, constructor, recurser or output functions in P.

The size of an input X will be the maximum depth of any of the atoms in X.

We can extend the definition of <= to the space of inputs by saying X<=Y if and only if all the selector variables in X are smaller than their respective variables in Y, and similarly the constructor variables.

We will say Y is X "pruned" at position a if Y is the largest input less than or equal to X in which a is atomic. This process can be viewed as simply taking the subtree in X rooted at a and replacing it by a unique atom.

If a position (relative to the original input) is tested by some predicate we will say that the position in question has been touched. Call the n positions touched by the predicates of P without going into recursion the primary positions of P.

The assumption of freedom asserts only the existence of inputs X which will cause the program to recurse a specific number of times and exit by a specific output function. Our first lemma shows that this can be made constructive.

Lemma 11: Given m>=0 and 1<=i<=n one can construct an input X so that P(X) is defined and when given X as an input P recurses m times before exiting by output function i.

proof: Consider m+p infinite trees corresponding to the m+p input variables. Mark in BLUE every position which is touched by a predicate function and found to be non-atomic in order for P to recurse m times and reach the predicate i. Then mark in RED the point touched by predicate i after recursing m times.

The assumption of freedom implies that no blue vertex can appear in the infinite subtree rooted at the red vertex, and that the red vertex can not also be marked blue.

Now mark in YELLOW all points which are used by constructor functions in recursing m times, and each position used by output function i after recursing m times. The assumption of freedom again tells us that no yellow vertex can appear in the infinite subtree rooted at the red vertex. The red vertex may, however, also be colored yellow, as may the blue vertices.

It is a simple matter to then construct an input X so that

1. all BLUE vertices are interior to X (non atomic),
2. The RED vertex is atomic, and
3. all YELLOW vertices are contained in X (they may be atomic)

□

Notice that the procedure given in the proof of lemma 11 allows

one to find the smallest X such that the indicated conditions hold. If a is the position in question we will call this point the minimal a point. Freedom implies no point can be twice touched, hence the minimal a point is a well defined concept.

Given an input X such that P(X) is defined, let $F_X(Z)$ be the straight line function such that $F_X(X) = P(X)$. Note that by theorem 10, F_X is defined by this single point.

Lemma 12: For any X for which P(X) is defined, one can construct an input Y with the properties that P(Y) is defined, $Y >= X$ and $F_X \neq F_Y$.

proof: Let m and i be the constants such that on input X, P recurses m times before exiting by output function i. Let the predicate p_i test variable x_j.

There are two cases. First assume f is not a constant function. Now it is possible that the position which would be tested by P_i after recursing m+1 times is an interior position in X, but since X is bounded there must be a smallest $k > m$ such that the predicate $p_i(R^k(x_j))$ is either true or undefined. Using lemma 11 we can find an input Z which causes P to recurse k times before exiting by output function i. Let Y be the union of X and Z. Since $Y >= Z$, P must recurse at least as much on Y as it did on Z. Since the final point tested is still atomic P(Y) will recurse k times before exiting by output function i. Since $f_i(R^m(X),C^m(Y)) \neq f_i(R^k(X),C^k(Y))$ we have that $F_X \neq F_Y$.

The second case arises when f_i is a constant function. By assumption 3 there is at least one output function which is not a constant function. Let f_i be this function. Let the predicate p_i test variable x_j. The same argument as before goes through with the exception that is may happen by chance the P(Y) = P(X) (i.e. P(Y) returns the constant value). In this case increment k by 1 and perform the same process and it cannot happen again that P(Y) = P(X). □

Lemma 13: If P touches a location a, then one can construct two inputs X and Y with the properties that P(X) and P(Y) are defined, furthermore for any Q in Φ, if P(X) = Q(X) and P(Y) = Q(Y) then Q must touch a.

proof: Let Z be the minimal a point. Using lemma 12 we can construct an input X such that P(X) is defined, X >= Z and $F_X \neq F_Z$. Let Y be X pruned at a.

First notice that P(Y) is defined and $F_Y = F_Z$. To see this note that every point which was tested by P in computing P(Z) and found to be non atomic is also non atomic in Y. Position a is atomic in both, and if the output function was defined on Z then it must be defined on Y which is strictly larger.

Suppose given input Y a program Q recurses m times before exiting by output function i, but does not touch position a. Since X is strictly larger than Y, on X Q must recurse at least as much and at least reach predicate i. Let the position in Y which was touched by predicate i and found to be atomic be b. Since position b is not the

same as position a, position b is also atomic in X. Therefore given input X Q will recurse m times and exit by output function i. But this implies by theorem 10 that $F_X = F_Y$, a contradiction. □

2.3.2.2 Bounding the depth of the recursion and predicate functions

Our first set of test inputs use the procedure given in lemma 13 to demonstrate that each of the n primary positions in P are indeed touched.

Next, for <u>each</u> selector variable, use the procedure given in lemma 13 to show that the first n+1 positions (by depth) must be touched. Let d be the maximum size of these m(n+1) positions. (We can assume d is at least 3 and is larger than both 2*maxd and olimit.)

> Lemma 14: If Q is a program in Φ which correctly processes these 2m(n+1) points, then the recursion selectors of Q have depth d or less.

proof: Study each selector variable separately. At least one of the n+1 points touched in that variable must have been touched after Q had recursed at least once. If the recursion selector had depth greater than d the program could not possibly have touched the point in question. □

> Lemma 15: If Q is a program in Φ which correctly processes these 2m(n+1) points, then none of the selector programs associated with the predicates of Q can have a depth greater than d.

proof: At least one of the inputs causes Q to recurse at least once, hence all the predicates must have evaluated FALSE and therefore were defined. If any of the predicates did have a depth greater than

d, they would have been undefined on this input. □

Since d>olimit we also have that d is a bound on the output functions of Q.

We are now in a position to make a comment concerning the size of the points computed by the procedure given in lemma 13. Let m be the maximum depth of the "relative root" (the current variable positions relative to the original variable tree) at the time position a is touched. We know the minimal a tree is no larger than m+maxd. This being the case to find an atomic or undefined point (as in the procedure associated with lemma 12) we will at worst have to recurse to a point m+maxd deep, but no more than m+maxd+d deep. Hence neither of the two points constructed in lemma 13 need be any larger than m+2*maxd+d. This fact will be of use in proving lemma 18.

2.3.2.3 Narrowing the form of the recursion selectors

Say a selector function f <u>factors</u> a selector function g if g is equivalent to f composed with itself some number of times. For example CADR factors CADADADR. We will say that f is a <u>simple factor</u> of g if f factors g and no function factors f, other than f itself.

Denote by s_i i=1,..,m the simple factors of r_i, the recursion selector functions. That is, for each variable i there is a constant m_i so that the recursion selector r_i is s_i composed with itself m_i times. Let q be the greatest common divisor of all the m's. Hence the recursion selectors of P can be written as S^q for some recursion

selector S.

We now construct a second set of data points in the following fashion: For each selector variable x_i, let a be the first position touched with depth greater than $2d^2$ in x_i. Using lemma 13 generate two points which demonstrate that position a must be touched. Let D_0 be the set containing all the $(2n + 2m(n+1) + 2m)$ points computed so far.

Lemma 16: If Q is a program in Φ which computes correctly on D_0 then recursion selector i of Q must be a power of s_i.

proof: Assume the recursion selector of x_i in Q is not a power of s_i. Recall that the depth of the selector cannot be any greater than d. Once it has recursed past the depth d it will be in a totally different subtree from the path taken by the recursion selector of P.

Since d>3 it is required that Q touch a point which has depth at least 3d. Q must therefore touch this point prior to recursing to the depth d. By lemma 14 this is impossible. □

We can, in fact, prove a slightly stronger result.

Lemma 17: If Q is a program in Φ which computes correctly on D_0 then there exists a constant r such that the recursion selectors of Q are exactly S^r.

proof: We know by lemma 16 that the recursion selectors of Q must be powers of s_i. For each selector construct the ratio of the power of s_i in Q to that in P. Lemma 17 is equivalent to saying that all these ratios are the same. Assume they are different and let x_i be

the variable with the smallest ratio and x_j the variable with the largest.

Let X and Y be the two inputs which demonstrate that a position a of depth greater than $2d^2$ in x_i is touched. Both P and Q must recurse at least 2d times on these inputs. In comparison to what P is doing, x_j is gaining at least one level every time Q recurses. By the time x_i is within range to touch a, X_j will have gone 2d levels too far. Since 2d>d+2*maxd, x_j will have run off the end of its input, hence Q cannot have received the correct answer on X and Y. □

Lemma 13 gave us a method to demonstrate a position is touched. We now give the opposite: a way to demonstrate a position is not touched.

> **Lemma 18:** If Q is a program in Φ which computes correctly on all the test points so far constructed, then for any position a not touched by P one can construct two inputs X and Y so that if P(X)=Q(X) and P(Y)=Q(Y) then Q does not touch a.

proof: Let position a be in variable x_i. Let m be the smallest number such that after recursing m times recursion selector i is deeper than the depth of a. Let h be the maximum depth of any recursion selectors at this point. Let X be the complete tree of depth h+2d pruned at a.

There are two cases: If P(X) is not defined, assume Q touches a. The relative roots of Q can not be deeper than h+d at the time a is touched. Hence the minimal a point is not any deeper than h+2d.

Since X is strictly larger than the minimal a point Q(X) must be defined, which contradicts the fact that P(X)=Q(X).

The second case arises if P(X) is defined. Using lemma 12, construct on input Z>=X such that $F_X \neq F_Z$. Let Y be Z pruned at a. Assume Q touches a. Since Y>=X, Q(Y) must be defined, so assume P(Y) is defined. By construction $F_Y=F_Z \neq F_X$. But since Q touched a, then $F_X=F_Y$, which is a contradiction. □

2.3.2.4 the recursion selectors must be the same as P

If Q is a program in Φ which executes correctly on D_0 then from lemma 17 we know the recursion selectors of Q must be S^r for some constant r. From lemma 14 we know the depth of S is no larger than d, hence there are at most d/(depth of S) possible alternatives. For each possible r (not equal to q, that is the value in P), construct a mutant program P' which is equal to P in all respects but the mutant selectors, which are S^r.

In this section we will consider test cases as pairs of inputs, generated using the procedure given in lemma 18, which return either the values YES, saying they were generated by the same straight line program, or NO, saying they weren't. Other than this we will not be concerned with the output of the mutants.

If each mutant touches a point which P does not, then construct two points (using lemma 18) to demonstrate this. If any mutant touches only points which P itself touches, then we will say P cannot

be shown correct by this testing method. This is the first example of testing by using mutant programs. Call this set of test inputs D_1.

Lemma 19: If Q is a program in Φ which executes correctly on D_0 and D_1 then the recursion selectors of Q must be exactly S^q.

proof: Assume not, and that the recursion selectors are S^r for some constant $r \neq q$. No matter what the primary positions of Q are, we know it must touch at some point the primary positions of P. It therefore must always touch the primary positions of P relative to the position it has recursed to. But therefore it must at least touch the points which the mutant associated with r does. □

2.3.2.5 Testing the Primary Positions of P

Consider each primary position separately. Assume that in some program Q in Φ the position is not primary, but that it is touched after having recursed m times. Let b be the position of a relative to S^{qm}. This means in Q that b is primary. Now b cannot even be touched (let alone be primary) in P because of the assumption of freedom. Using the procedure given in lemma 18, construct two points which demonstrate that b is not touched. Taken together, these test points insure that the primary positions of P must be primary in any other program.

Notice carefully that we need to make no other assumptions about the other primary positions in Q, that is, we can treat each of them independently. We therefore have at most n(d/(depth of S^q)) mutant programs, hence at most twice this number of test points. Call this

test set D_2.

Lemma 20: If Q is a program in Φ which executes correctly on D_0, D_1, and D_2 then the primary positions of Q are exactly those of P.

□

Notice that by theorem 10 this also gives us the following

Lemma 21: The output functions of Q are exactly those of P.

2.3.2.6 Main Theorem

Once we have the other elements fixed, the recursion constructors are almost given to us. Remember one of the assumptions made in the beginning was that each of the constructor variables appears in its entirety in at least one of the output functions. All we need do is to construct P data points so that data point i causes the program P to recurse once and exit using an output function which contains constructor variable i. Call this set D_3. Using theorem 10 we then have

Lemma 22: The recursion constructors of Q must be exactly those of P.

□

The only remaining source of variation is the order in which the primary positions are tested. The only solution we have been able to find here (short of making more severe restrictions on Φ) is to try all possibilities. (If all the output functions are unique, this step is unnecessary, since D_0 suffices to show the order.) There are n! of

these, some of which may be equivalent to the original program. Let D_4 be a set of data points which differentiates P from all non equivalent members of this set.

Putting all of this together gives us the main theorem:

Theorem 23: Given a program P in ϕ, there exists a finite set of test cases which can be effectively constructed which have the property that for any program Q in ϕ, if Q tests equal to P on these points, then P is equivalent to Q.

Proof: This follows directly from lemmas 14, 19, 20, and 22.

Corollary: Either P is correct or no program in ϕ realizes the intended function.

Corollary: If the competent programmer hypothesis holds then P is correct.

2.3.3 Discussion

Theorem 23 and the restrictions which were made in section 2.3.2 in order to prove it demonstrate one very critical problem with this approach. These restrictions were not made because they were in any sense natural or reasonable, but because they were necessary to the proof. While we cannot rule out the possibility that these restrictions could be removed or that a simpler proof could be found, this does show that any such result is likely to be difficult to discover. This severely weakens our hope that a theorem such as 23 could be proved for a more natural class of programs. This being the case, if one wants to analyze the type of programs which programmers

are more accustomed to using the empirical approach of chapter four must be used.

Note that although the class of programs studied here is small, it is not vacuous. Several examples previously studied by other authors [44, 78, 84] can be expressed in the form used here.

While it is known that the equivalence problem for linear recursive schemata is decidable [64], it is not clear what relationship this has to the present work. For one thing the programs studied in the section on LISP are partial, not total as is assumed in the schemata results. Secondly while theorem 10 gives us some knowledge about how an output was derived from a given input, we cannot a priori decide, for example, how many times the program recursed before providing this output. Finally note that, as opposed to the finite case (theorem 4), it is not clear at all that merely having a decidable equivalence property is sufficient to show the existence of a set which differentiates P from the infinite set ϕ.

To see that ϕ is infinite, we point out that even with the assumed bound on the depth of the output functions, we did not bound the number of CONS functions they could contain, hence there are an infinite number of programs in the set ϕ. This is true even after we have bounded the depth of the recursion selectors and the predicate selectors in lemma 15.

The most important aspect of this result is not the proof (which,

in fact has rather limited applicability) but the method of the proof. Once we have fixed the recursion selectors via test set D_0, the remainder of the arguments are proved by constructing a small set of alternative programs (the mutants) and showing that test data designed to distinguish these from the original actually will distinguish P from a much larger class of programs. In all we constructed d(1/(depth of S) + n/(depth of S^q)) + p + n! mutants, and proved that test data which distinguishes P from this set of mutants actually distinguishes P from the infinite set of programs in ϕ.

Finally note that although the proof of the result given here is rather long and tedious, the end result is a procedure which is entirely mechanical for proving correctness. The user of such a procedure need have no knowledge of the proof which was used to validate the method, much like the user of a timesharing system need have no knowledge of how the operating system is implemented. This is the direction I feel research in testing should follow: finding mechanical methods which may be difficult to verify, but once verified give an easy procedure for developing good test data.

CHAPTER 3

A FORTRAN MUTATION TESTING SYSTEM

During the summer of 1977 a system to perform mutation analysis on FORTRAN programs was designed and constructed. This first system was called PIMS, for PIlot Mutation System [11] and was implemented on the PDP10 at Yale University.

The PIMS system allowed the user to test single ANSI standard FORTRAN subroutines. The language FORTRAN was chosen for the initial implementation because it was (and still is) widely studied in the testing literature, it has a fairly simple semantics with few language constructs, and it lends itself to considering large programs on a module by module basis. The last consideration was important because of the restriction to testing only single subroutines.

A second important consideration in favor of FORTRAN was the fact that FORTRAN programs possess the important property that small syntactic changes usually produce only small semantic changes. A higher level language, for instance APL, does not possess this property. In APL even small syntactic changes can produce radically different programs, hence we would expect very few of the syntax based

mutants to produce useful results for APL. On the other hand, much lower level languages, for example assembly code, possess this property to an even larger extent than FORTRAN. But mutating the assembly code produces other problems; What may be a simple mutation on the FORTRAN level may produce large changes in the assembly code, for example changing an AND operator to an OR operator where these are implemented with control structures. An even more important consideration is psychological; A mutant described in FORTRAN has a quickly assimilable meaning, whereas a mutant described in assembly language might require a significant amount of analysis on the part of the human tester to decipher its consequences. Therefore in many respects, FORTRAN was a good choice for an experimental testing system.

Both the PIMS system and the later EXPER system operated by parsing the program into an internal form, which was interpreted, and producing the mutants at the internal form level. This method was used rather than parsing the program into assembly code for several reasons: If the mutants were produced at the source level, it would be necessary to recompile the program each time a mutant was to be executed. On the other hand if the mutants were produced at the assembly language level we would be restricted by the particular structure of the PDP10 machine architecture. In any case the program would have to be parsed anyway in order to construct the mutants, so that half the interpretive system was already necessary. In addition

an interpreter based system allowed us greater control over program execution, which was important since many of the mutants were unstable, often resulting in infinite loops, zero divides or floating point interrupts.

Several experiments were conducted on PIMS during the following year, and it quickly became apparent that the restriction to a single subroutine was becoming a bottleneck. Accordingly in the spring of 1978 a second mutation system, which came to be called EXPER (for EXPERimental mutation system), was designed and built. Details of EXPER are described in [14, 15]. The chief language limitations in this system were the absence of input/output statements (all communication is through parameter and common variables), the lack of statement functions, and the lack of complex data types. The EXPER system was used for all the studies described in chapter four.

3.1 Mutant operators in the EXPER system

The most important source of variation in the design of a system to implement mutation analysis is the choice of those programs which are to be considered mutants of the original program. The choice of mutants is, in effect, what characterizes the system, and two systems which produce different mutants from the same source program may exhibit radically different behavior. This section will be devoted to describing the mutants produced by the EXPER system.

Since the mutants are produced automatically they must be

produced by some fixed algorithm. In designing EXPER, therefore, we chose to view the construction of the mutants as the application of a series of mutant operators to the original program. A mutant operator is a procedure which takes as input a program and produces as output a set of mutant descriptions. A mutant description is a short (3 computer words in the PDP20 implementation) encoding of the type of mutation and its location in the original program. See [14] for more details on the actual implementation.

We can divide the mutant operators into three groups, depending upon whether they affect operands, operators or statements as a whole.

3.1.1 Source operand mutant operators

The first set of mutant operators alter the basic data objects which are being manipulated by the program. We can consider three different types of basic data objects: constants, scalar variables and array references. There are accordingly nine mutant operators which can be described by the form

replace each x with each distinct occurrence of y

where x and y range over constants, scalars and array references.

Another operator takes each constant (even those which appear in DATA statements) and alters it slightly. Slightly means for integers plus or minus 1, for reals plus or minus 10% (or if the value is zero .01), for logicals the logical complement, and for characters the first character in a string constant is replaced by the character's neighbors in the underlying ordinal scheme (e.g. 'cat' is replaced by

'bat' and 'dat'). Of course it may be possible for this type of mutant to duplicate one produced by the first group, and a certain attempt is made to avoid this redundancy.

A third type of source operand mutation takes the array name in each occurrence of an array expression and changes it to all other array names of the same dimensionality.

3.1.2 Operator mutations

In order to extend the error detection power of the mutations, the EXPER system adds several new operators to the usual FORTRAN repertoire. These new operators cannot appear in the original program, but are produced in mutants.

The first two are binary operators which can take the place of either arithmetic or logical operators. They are called leftop and rightop and their semantics is to evaluate both operands (this is an artifact of the stack type architecture of the interpreter) and to return either the left or right hand argument, ignoring the other argument.

A second pair of new operators are also binary and can take the place of relational or logical operators. They are called trueop and falseop and their function is to evaluate both operands, and regardless of their values return TRUE or FALSE, respectively.

There are also several unary operators which have been created. Twiddle (denoted ++ or --) is an operator which returns its argument

plus or minus one (if the argument is integer) or plus or minus .01% or .01 (which ever is greater, if the argument is real). -ABS returns the negation of the absolute value of its argument. ZPUSH(X) returns X if nonzero, otherwise a TRAP error occurs and the mutant program is eliminated. The purpose of the last operator is to force the expression X to be zero.

Having described these new operators, the procedures which construct the mutants can be characterized as follows: Arithmetic operator mutations are formed by taking each arithmetic operator and replacing it with the other members of the set {+, -, /, *, **, leftop, rightop}. Relational operator mutations are formed by replacing each relational operator with other members of the set {.LE., .LT., .EQ., .NE., .GT., .GE., trueop, falseop}. Logical operator mutations are formed by replacing each logical operator with other members of the set {.OR., .AND., leftop, rightop, trueop, falseop}. Unary operator removals are made by deleting each unary operator. Unary operator insertions are formed by inserting the unary operators {-, .NOT., ++, --, ABS, -ABS, ZPUSH} wherever they would be syntactically correct.

Again, there is some possibility of creating redundant mutations (for example changing A - 1 to A * 1, which is the same as deleting the - 1 clause altogether) or unnecessary mutants (such as ZPUSH(A*B), since the effect will be achieved by placing the ZPUSH around the A and B separately). Some effort is made to avoid these.

3.1.3 Statement mutations

A sequence of unlabelled, non-IF statements is called a basic block [39]. Statements in a basic block have the property that if any one of the statements is executed they all must be executed. One type of statement mutation operator replaces the initial statement of each basic block with a TRAP statement. The semantics of the TRAP statement are that if it is ever executed it immediately causes the mutant to abort. On the other hand if such a mutant survives it implies that the basic block has never been executed. In this manner mutants insure that every statement is executed at least once.

Just because a statement can be reached does not mean that it is performing a necessary part of the program being executed. A second mutant operator replaces each statement with a CONTINUE statement, effectively deleting the statement.

A third operator changes the labels on GOTO statements and arithmetic IF statements to other labels in the program.

The final mutant operator has two parts. The first part changes the ending statement label on DO loops to other labels which lie between the do loop head and the end of the program. The second part changes the DO statement into a FOR statement, where a FOR statement differs from a DO statement in that if the ending value is smaller than the starting value the loop is not executed, in contrast to FORTRAN where loops are always executed at least once.

3.2 A consideration of the power of the mutant operators

One way to assess the capabilities of the mutant operators just outlined is to select a set of programs with known errors and ask which of the errors would be caught by this method. This information can then be compared with other testing methodologies which have been proposed in the literature, or used to direct the search for new mutant operators. Several studies of this nature will be reported on in the next chapter.

A second way to evaluate this particular choice is to ask if the operators force data to be constructed which achieves the goals of other testing methods. In this section a number of testing methods will be examined, and we will see that in many cases mutation analysis does subsume their goals.

3.2.1 Trivial errors

If one of the mutants considered is indeed the correct program then of course the error will be discovered when an attempt is made to eliminate that particular mutant. Alternatively if the errors in the original program act in a reasonably independent manner and each error is individually captured by a single mutation then the errors will almost certainly be detected.

Given the vast folklore about large systems failing for extremely trivial reasons [67], the ability to detect such simple errors in indeed a good starting place. However many errors do not correspond

exactly to the generated mutations, and multiple errors may interact in subtle fashions. This being the case, any realistic testing method must demonstrate a much more powerful error detecting capability.

3.2.2 Statement analysis

Many programming errors manifest themselves by sections of code being "dead", that is unexecutable, when they shouldn't be. Also many bugs are of such a serious nature that any data which executes the particular statement in error will cause the program to give incorrect results. These errors may persist for weeks or even years if the error occurs in a rarely executed section of code.

Accordingly a reasonable first goal for a set of test cases is that every statement in the program is to be executed at least once [58].

Various authors have presented methods to achieve this goal [41, 82]. Usually these methods involve the insertion of counters into the straight line segments of code. When all counters register non-zero values every statement in the program has been executed at least once.

In mutation analysis we take a different approach with the same objective. If a statement is never executed then obviously any change we produce in it will not cause the altered program to produce test answers differing from the original. However as a means of directing the programmers attention to these errors in a more direct and

unambiguous fashion the TRAP mutants described in section 3.1.3 are generated. Obviously these mutations are extremely unstable, since any data which executes the replaced statement will cause the mutant to produce an incorrect result, and hence to be eliminated. The reverse, however, is also true. That is, if any of these mutants survive, then the statement which the mutation altered has never been executed. Hence an accounting of the survival of this class of mutations gives important information about which sections of code have and have not been executed.

A statement can be executed and still not serve any useful purpose. In order to investigate this possibility we generate another type of mutant which replaces every statement with a CONTINUE statement (a convenient FORTRAN statement with no semantic meaning). The survival or elimination of these mutations gives more information than merely whether the statement is executed or not, it indicates whether or not the statement is performing anything useful. If a statement can be deleted with no effect then at best it indicates a waste of machine time and at worst it is probably indicative of much more serious errors.

Merely being able to execute every statement in the program is no guarantee that the code is correct [37, 52] Problems such as coincidental correctness or predicate errors may pass undetected even if the statement in error is executed repeatedly. In subsequent sections we will see how mutation analysis deals with these problems.

3.2.3 Branch analysis

Some authors have pointed out [58] that an improvement over statement analysis can be achieved by insuring that every flowchart branch is executed at least once. For example the following program segment

```
A
IF (condition)
  THEN B
C
```

has two branches corresponding to the two flows A-B-C and A-C. All three statements A,B and C can be executed by a single test case. It is not true, however, that in this instance all branches have been executed.

The requirement that every branch be taken is equivalent to requiring that every predicate expression evaluate both TRUE and FALSE. It is this formalization which is used in mutation analysis. (Sources of other branches are arithmetic IF statements and GOTO statements. Simple GOTO statements are covered by the statement analysis mutants described in the last section. Arithmetic IF and computed GOTO statements are covered by mutating the label portion of the statement).

Among the mutants generated are ones which replace each relational expression and each logical expression by the logical constants TRUE and FALSE. Of course, like the statement analysis mutations these are very unstable and easily eliminated by almost any data. But if they survive they point directly and unambiguously to a

weakness in the test data which might shield a potential error.

By mutating each relation or logical expression independently we actually achieve a stronger goal than that usually achieved by branch analysis.

Consider the compound predicate

(A <= B AND C <= D)

The usual branch analysis method would only require two test cases to test this predicate. If the test points were (A<B,C<D) and (A<B,C>D) this would have the effect of only testing the second clause, and not the first. This is because branch analysis fails to take into account the "hidden paths" [22], implicit in compound predicates.

In testing all the hidden paths mutation analysis would require at least four points to test this predicate. The four points correspond to the branches (A > B,C > D), (A > B, C <= D), (A <= B,C > D) and (A <= B, C <= D). (Predicate testing, as described in section 3.2.5 would require us to construct,in addition, several more points.)

As an example of how errors can be detected in this manner consider the program shown in figure 3-1, taken from an article by Geller [33]. The program is intended to derive number of days between two given days in a given year. The predicate which determines whether a year is a leap year or not is, however, incorrect in this version. Notice that if a year is divisible by 400 (year REM 400 = 0) it is necessarily also divisible by 100. Hence the logical expression

```
PROCEDURE calendar (INTEGER VALUE
                        dayl, monthl, day2, month2, year);
  BEGIN
    INTEGER days;
    IF month2 = monthl THEN days = day2 - dayl
      COMMENT if the dates are in the same month, we can
      compute the number of days between them immediately;
    ELSE
      BEGIN
        INTEGER ARRAY daysin (1 .. 12);
        daysin(1) := 31; daysin(3) := 31;  daysin(4) := 30;
        daysin(5) := 31; daysin(6) := 30;  daysin(7) := 31;
        daysin(8) := 31; daysin(9) := 30;  daysin(10):= 31;
        daysin(11):= 30; daysin(12):= 31;
        COMMENT set daysin(2) according to whether or not
            year is a leap year ;
        IF ((year REM 4) = 0) OR
           ((year REM 100) = 0 AND (year REM 400) = 0)
          THEN daysin(2) := 28
          ELSE daysin(2) := 29;
        days := day2 + (daysin(monthl) - dayl);
        COMMENT this gives the correct number of days -
                    days in complete intervening months);
        FOR i := monthl + 1 UNTIL month2 -1 DO
          days := daysin(i) + days;
        COMMENT add in the days in
                complete intervening months;
      END;
    WRITE(days)
  END;
```

Figure 3-1: Program exhibiting an error caught by branch analysis

formed by the conjunction of these two conditions is equivalent to just the second term alone. Alternatively, the expression year REM 100 = 0 can be replaced by the logical constant TRUE and the resulting mutant will be equivalent to the original. Since this is obviously not what the programmer had in mind the error is discovered.

3.2.4 Data flow analysis

During execution a program may access a variable in one of three ways [29]. A variable is <u>defined</u> if the result of a statement is to assign a value to the variable. A variable is <u>referenced</u> if the

statement requires the value of the variable to be accessed. Finally a variable is undefined if the semantics of the language do not explicitly give any other value to the variable. Examples of the latter are the values of local variables on invocation or procedure return, or DO loop indices in FORTRAN on normal do loop termination.

Fosdick and Osterweil [29] have defined three types of data flow anomalies which are often indicative of program errors. These anomalies are consecutive accesses to a variable of the forms:

1. undefined and then referenced
2. defined and then undefined
3. defined and then defined again

The first is almost always indicative of an error, even if it occurs only on a single path between the place where the variable becomes undefined and the reference place. The second and third, however, may not be indications of errors unless they occur on every path between the two statements.

Although the first type of anomaly is not attacked by mutations per se it is attacked by the mutation system, which is a large interpretive system for automatically generating and testing mutants. Whenever the value of a variable becomes undefined it is given a special marking. Before every variable reference a check is performed to see if the variable has a value. If the variable does not an error is reported to the user, who can take corrective action.

The second and third types of anomalies are attacked more directly. If a variable is defined and not used then usually the statement can be eliminated with no obvious change (by the CONTINUE insertion mutations described in the last section). This may not be the case if, for example, in the course of defining the variable a function with side effects is invoked. In this case the definition can likely be mutated in any number of different ways which, while preserving the side effect, obviously result in the variable being given different values. An attempt to remove these mutations will almost certainly result in the anomaly being discovered.

3.2.5 Predicate testing

Howden [52] has defined two broad categories of program errors under the names domain errors and computation errors. The notions are not precise and it is difficult with many errors to decide which category they belong in. Informally, however, a domain error occurs when a specific input follows the wrong path due to an error in a control statement. A computation error occurs when an input follows the correct path but because of an error in computation statements the wrong function is computed for one or more of the output variables.

Following Howden's study, several researchers examined the question of whether certain testing methodologies might reliably uncover errors in these or other classification schemes. A number of authors [6, 20] observed that points on or near a predicate border are most sensitive to domain errors. A testing method, called the domain

strategy [87], formalized this in that it is guaranteed to catch a limited class of these types of errors. The original reference should be consulted for a more complete presentation of the several technical restrictions and applications of their method, but we can here give an informal description of how it works.

If a program contains N input variables (including parameters, array elements and I/O variables) then a predicate can be described by a surface in the N dimensional input space. Often the predicate is linear, in which case the surface is an N dimensional hyperplane. Let us consider a simple two dimensional case where we have input variables I and J and the predicate in question is

```
I+2*J <= -3
```

The domain strategy would tell us that in order to test his predicate we need three test points, two on the line I+2*J=-3 and one a small distance e from the line. Call the two points on the line A and B and the point off the line C.

Assuming a correct outcome from these tests what have we discovered? We know the line of the predicate must cut the sections of the triangle AC and BC. Since e is quite small the chances of the predicate being one of these alternatives but not the original line is also small. Hence, although we don't have complete confidence that the predicate is correct, we do have a much larger degree of confidence than we could otherwise have attained.

To see how mutation analysis deals with the same problem we first observe that it really is not necessary to have both A and C be on the predicate line. If A is on the line and B and C are on opposite sides of the line the same result follows. We now described how mutations cause points with these properties to be generated.

As an intuitive aid one can think of mutation analysis as posing certain alternatives to the predicate in question, and requiring the tester to supply reasons, in the form of test data, why the alternative predicate could not be used just as well in place of the original. These alternatives are constructed in various ways.

A number of the alternatives are generated by changing relational operators. Changing an inequality operator to a strict inequality operator, or vice versa, generates a mutant which can only be eliminated by a test point which exactly satisfies the predicate. For example changing I+2*J <= -3 to I+2*J < -3 requires the tester to exhibit a point for which I+2*J = -3, hence which satisfies the first predicate but not the second.

A second class of alternatives involves the introduction of the unary operator "twiddle" (denoted ++ or --), whose semantics were described in section 3.1.3. Graphically, the effect of introducing twiddle is to move the proposed constraint a small distance parallel to the original line. In order to eliminate these mutants a data point must be found which satisfies one constraint but not the other,

hence is very close to the original constraint line.

Finally a third class of alternatives are constructed by changing each data reference into all other syntactically correct data references, and each operator into all other syntactically correct operators. The effects of these are related to the phenomenon of spoilers, which are described in section 3.2.9.

The total effect achieved by so many alternatives is to cause the programmer to generate a large number of test inputs which are very closely tied to the particular form of the program. Hence by a process similar to that of White et al [87] we increase inductively our confidence that the predicate is indeed correct.

```
READ I,J;

IF I <= J + 1
  THEN K = I + J - 1;
  ELSE K = 2*I + 1;

IF K >= I + 1
  THEN L = I + 1;
  ELSE L = J - 1;

IF I = 5
  THEN M = 2*L + K;
  ELSE M = L + 2*K -1;

WRITE M;
```

Figure 3-2: Example program from White [87]

In order to more fully illustrate the construction of these alternatives and demonstrate their utility we will examine a small example. The program in figure 3-2 was taken from the paper describing this method. No specifications were given, but the program

can be compared against a presumably "correct" version. It is a good example because it only involves two input variables, hence the alternatives can be easily illustrated in a graphical manner.

As one can see the program has three predicates: I <= J+1, K >= I+1 and I = 5. Consider only the effects of changing the first.

The EXPER system looks at 41 distinct alternatives for the predicate I <= J+1 [16]. In fact 45 choices are tried, however some of the choices are redundant, for example ++I <= J+1 and I <= --J + 1. These redundancies are created because the mutants are formed in an entirely mechanical way. It is our feeling that the processing time lost because of redundant mutations is much less than the time which would be required to eliminate them by preprocessing the alternatives, hence the presence of these redundancies is of little concern.

In the paper from which the example program was taken the authors hypothesize that the program contains the following four errors.

1. The predicate K >= I+1 should be K >= I+2.
2. The predicate I=5 should be I=5-J.
3. The statement L=J-1 should be L=I-2.
4. The statement K=I+J-1 should read

```
THEN IF (2*J < -5*I -40)
        THEN K = 3;
        ELSE K=I+J-1;
```

It can be shown that the attempt to eliminate the alternative K >= I+2 must necessarily end with the discovery of the first error.

Note that this is not trivially the case since errors 1 and 4 can interact in a subtle fashion.

3.2.6 Error sensitive test cases (ESTCA)

Another testing methodology directed specifically at detecting a certain type of error is the Error Sensitive Test Cases method proposed by Kenneth Foster [30]. The method is a procedure for deriving test cases specifically directed to detecting the following types of errors:

1. Omitted or out of sequence operations or conditions.
2. References to the wrong variables.
3. Substitution of relations or conjunctions in simple or compound conditions (LT for LE, OR for AND).
4. Missing or incorrectly placed parentheses, incorrect grouping of variables in an arithmetic expression.
5. Substitution of arithmetic operations (* for **).
6. Incorrect constant values as factors in arithmetic computations or as limit counters.
7. Arithmetic operations on variables in the wrong sign form (load or store positive, negative, with complement sign, and so on).

The method used in detecting these errors is quite similar to mutation analysis. That is, test data is derived which would be incorrectly processed were the program to contain an example of this type of error. This is precisely the same thing as requiring that all non equivalent mutants be eliminated.

The major difficulty with Fosters method is its complexity. In order to include all the special cases which can arise in conjunction

with these errors the procedure is, by necessity, quite detailed and complex. Note that if, for example, we want to insure that a specific + operator might not inadvertently be appearing where a * operator should it is not simply sufficient to find test data where the two expressions locally disagree, but we must continue to analyze the two programs until they halt to insure that different responses are produced. This distinction between local and global testing is discussed in more detail in section 5.1.

The ESTCA method was designed to allow a human programmer to construct test cases, but in so doing the amount of information which must be maintained and processed is exceedingly large. The danger is quite clear that one can lose sight of the forest for all the trees. This criticism is only partially alleviated by proposing to generate the test cases automatically (a proposal which in itself raises many more unanswered questions).

On the other hand the experience with the mutation system suggests that the vast majority of changes of the type analyzed by ESTCA are likely to be detected by even the most rudimentary test cases. It is only a small number of changes which are subtle enough to require detailed investigation, and it is only these few changes which should require the human testers attention. Unfortunately discovering, a priori, which of these changes are important is an exceedingly difficult problem.

3.2.7 Domain pushing

One very important mutation which was mentioned in the section on predicate errors concerns the introduction of unary operators into the program. These unary operators are introduced wherever they are syntactically correct according to the rules of FORTRAN expression construction. In addition to the operators ++ and -- already discussed, the remaining unary operators are - (arithmetic negation) and a class of non FORTRAN operators ABS (absolute value), -ABS (negative absolute value) and ZPUSH (zero push). Only the actions of the last three will be described in this section.

Consider the statement

```
A = B + C
```

in order to eliminate the mutants

```
A = ABS(B) + C
A = B + ABS(C)
A = ABS(B + C)
```

we must generate a set of test points where B is negative (so that B+C will differ from ABS(B)+C), C is negative and the sum B+C is negative. Similarly negative absolute value insertion forces the test data to be positive, and ZPUSH forces it to be zero. We use the term domain pushing for this process, meaning the mutations push the tester into producing test cases where the domains satisfy the given requirements.

Compound this process by every position where an absolute value sign can be inserted and one can observe a scattering effect, where the tester is forced to include test cases acting in various conditions in a multitude of domains. Very often in the presence of

an error this scattering effect will cause a test input to be generated which will demonstrate the error.

Notice that if it is impossible for B to be negative then this example produces an equivalent mutation, that is the altered program is equivalent to the original. In this case the proliferation of these alternative can either be a nuisance or an important documentation aid, depending upon their frequency and the testers point of view. The topic of equivalent mutants will be examined more fully in section 4.5.

Recall again that one of the errors the program in section 3.2.5 was presumed to contain altered the statement L = I-2 to L = J-1. One effect of this error is that any test input in the area I > J+1 and I <= 0 will produce erroneous results. But this is precisely the area which the mutant K = 2*ABS(I) + 1 directs us to. This means that this error could not have gone undiscovered using mutation analysis.

This process of pushing the programmer into producing data satisfying some criterion is often also accomplished by other mutations. Consider the program in figure 3-3, which is based on a program by Naur [71], and is one of the programs studied in section 4.2.2.

Consider the mutant which replaces the first statement FILL:=0 with the statement FILL:=1. The effect of this mutation is to force a test case to be defined in which the first word is less than MAXPOS

```
alarm := FALSE
bufpos := 0;
fill := 0;
REPEAT
  incharacter(cw);
  IF cw = BL or cw = NL
  THEN
       IF fill + bufpos <= maxpos
       THEN BEGIN
            outcharacter(BL);
            END
       ELSE BEGIN
            outcharacter(NL);
            fill := 0 end;
            FOR k := 1 STEP 1 UNTIL bufpos DO
                 outcharacter(buffer[k]);
            fill := fill + bufpos;
            bufpos := 0 END
     ELSE
       IF bufpos = maxpos
       THEN alarm := TRUE;
       ELSE BEGIN
          bufpos := bufpos + 1;
          buffer[bufpos] := cw END
 UNTIL alarm OR cw = ET
```

Figure 3-3: Example program from Naur [71]

characters long (since the effect of the mutant must be manifest before FILL is redefined). This test case detects one of the five errors in the program [37]. The interesting observation is that the effect of this mutation is several statements distant from the statement in which the mutation takes place, again illustrating one aspect of the coupling effect.

3.2.8 Special values testing

Another form of testing which has been introduced by Howden [53], is called special values testing. Special values testing is defined by a number of rules , for example

1. Every subexpression should be tested on at least one test case which forces the expression to be zero, together with one which forces it be be greater than zero, and one which

forces it to be less than zero.

2. No two variables should always have equal values.
3. A variable should assume more than one value during each test case.
4. Every subexpression should take on more than one value across all test cases.

That the first rule is enforced by the zero push and absolute value mutations has already been discussed in the last section on domain pushing.

That the second rule is important is undeniable. If two variables are always given the same value then they are not acting as "free variables" and a reference to one can be universally replaced with a reference to the second. In fact this is exactly what happens in this case, and the existence of these mutations enforces the goals of the distinct values rule.

A similar argument could be made concerning the importance of the third rule, however in languages which do not have the ability to make labelled constants (such as FORTRAN), variables are often used in this fashion. This rule is not enforced by mutation analysis.

The fourth rule is enforced for some special cases, for example predicates (section 3.2.3), and by substituting constants for scalars and array expressions. A more general method of enforcing this rule could be envisioned and is indeed implemented in a system similar to mutation analysis (see section 3.5). However there is at least some

doubt whether the more general capabilities would justify the increased costs involved in enforcing them.

3.2.9 Coincidental correctness

We say the result of evaluating a given test point is coincidentally correct if the result matches the intended value in spite of the fact that the function used to compute the value is incorrect. For example if all our test data results in a variable I having the values 2 or 0, then the computation J = I*2 could be coincidentally correct if what was intended was J = I**2.

The problem of coincidental correctness is really central to program testing. Every programmer has encountered statements which were incorrect, but which produced the correct response for a surprising large number of inputs. Yet with the exception of mutation analysis no testing methodology in the authors knowledge deals directly with this problem. Some researches even go so far as to state that the problems of coincidental correctness are intractable [87].

In mutation analysis coincidental correctness is attacked by the use of spoilers. Spoilers implicitly remove from consideration data points for which the results could obviously be coincidentally correct, in a sense "spoiling" those data points. For example by explicitly making the mutation J = I*2 => J = I**2 we spoil those test cases for which I = 0 or I = 2, and require that at least one test

case have an alternative value.

Often the fact that two expressions are coincidentally the same over the input data is an indication of program error or poor testing. For example the sorting program shown in figure 3-4, taken from a paper by Wirth [88], will perform correctly for a large number of input values. If, however, the statements following the IF statement are never executed for some loop iteration it is possible for R3 to be incorrectly set, and an incorrectly sorted array may be produced.

```
Sort(R4)
For R1 = 0 by 4 to N begin
  R0 := a(R1)
  for R2 = R1 + 4 by 4 to N begin
    if a(R2) > R0 then begin
      R0 := a(R2)
      R3 := R2
    end
  end
  R2 := a(R1)
  a(R1) := R0
  a(R3) := R2
end
```

Figure 3-4: A program exhibiting a coincidental correctness error

By constructing the mutant which replaces the statement a(R1):=R0 with a(R1):=a(R3) we point out that there are two ways of defining R0, only one of which is used in the test data. Therefore the error is uncovered.

3.2.10 Missing path errors

As identified by Howden [52], we can say a program contains a missing path error if a predicate is required which does not appear in the program under test, causing some data to computed by the same

function when really different functions are called for. These missing predicates can, however, be the result of two different problems, so we might consider the following definitions: A program contains a specificational missing path error if two cases which are treated differently in the specifications are incorrectly combined into a single function in the program. On the other hand a program contains a computational missing path error if within the domain of a single specification a path is missing which is required only because of the nature of the algorithm or data involved.

An example of the first type is error number four from the example in section 3.2.5. Although this error might result from a specification, there is nothing in the code itself which would give any hint that the data in the range 2*J<-5*I-40 is to be handled any differently than given in the test program.

For an example of the second class of error consider the subroutine shown in figure 3-5, which is one of the programs studied in section 4.2.1. The inputs are a sorted table of numbers and an element which may or may not be in the table. The only specification is that upon return X(LOW) <= A <= X(HIGH), and HIGH <= LOW + 1. The problem arizes if the program is presented with a table of only one entry, in which case the program loops forever.

Nothing in the specifications state that a table with only one entry is to behave any differently from a table with multiple entries,

```
      SUBROUTINE BIN(X,N,A,LOW,HIGH)
      INTEGER X(N),N,A,LOW,HIGH
      INTEGER MID
      LOW = 1
      HIGH = N
6     IF(HIGH - LOW - 1) 7,12,7
12    STOP
7     MID = (LOW + HIGH) / 2
      IF (A - X(MID)) 9,10,10
9     HIGH = MID
      GOTO 6
10    LOW = MID
      GOTO 6
      END
```

Figure 3-5: Program exhibiting a missing path error

it is only because of the algorithm used that this must be treated as a special case.

Problems of the second type are usually caused by the necessity to treat certain values, for example negative numbers, differently from others. This being the case the process of data pushing and spoiling described in sections 3.2.7 and 3.2.9 will often lead to the detection of these errors. So it is in this case where an attempt to remove either of the following mutants will cause us to generate a test case with a single element.

```
IF (HIGH - LOW - 1) 12,12,7
  MID = (LOW + HIGH) - 2
```

Since mutation analysis, like most other testing methodologies, deals only with the program under test (as opposed to dealing with the specifications), the problems of detecting specificational missing path errors are much more difficult. Since mutation analysis causes the tester to generate a number of data points which exercise the program in a multiplicity of ways our chances of stumbling into the

area where the program misbehaves are high, but are by no means certain.

So it is with the missing path error from the example in section 3.2.5. It is possible to generate test data which passes our test criterion but which fails to detect the missing path error. We view this not, however, as a failure of mutation analysis but as a fundamental limitation in the testing process. In our view the only way that this type of error can be eliminated is to start with a core of test cases generated from the specifications, independent of the program implementation. This core of test cases can then be augmented to achieve goals such as those presented by mutation analysis. Some methods of generating test data from specifications have been discussed elsewhere [37, 73].

3.3 A discussion concerning the number of mutants generated by EXPER

The most commonly made criticism of mutation analysis is that it requires the execution of an inordinately large number of alternative programs. This section will analyze the number of mutants generated by a typical program. The question of whether this number is practical will be more fully addressed in section 4.4.

What are some ways we can measure the size of a program? The easiest metric is just the number of executable statements, which we will denote by N. This number can, however, be deceptive. For example a single assignment statement can be simple (A = B) or

extremely complex

```
A=SQRT(B+SIN(ABS(B**5))/COS(B*B))
```

Accordingly another measure we might use is the number of references made to constants, scalar variables, and arrays (X) or the number of these references which are distinct (Y).

Another element we might wish to measure is the complexity of the control structure. McCabe has defined one commonly used measure of complexity [66], which we will denote V. Finally Halstead [28] has defined a general metric of program size, which he calls Effort (E).

N	M	X	Y	E	V
12	2580	103	21	32033	1
13	317	27	8	4071	5
17	386	32	8	6928	4
17	634	45	9	15246	7
24	2716	72	40	17565	7
26	646	40	11	16270	9
33	859	55	13	41819	12
33	23382	407	53	249701	1
56	3657	129	23	138939	9
66	2425	115	15	170492	10
67	5230	158	28	189585	15
71	2888	135	16	166715	11
98	8457	227	32	365825	22
112	16380	237	68	320331	26
277	34657	545	63	3024488	122
514	120000	1138	93	19267409	113

Table 3-1: Number of mutants generated versus program size

Table 3-1 gives, for 16 typical programs, a table of N, M, X, Y, E and V. Notice that the number of mutants is not particularly tied to the number of statements (the two programs with 33 statements are a good example of this). If we use a correlation coefficient [27] as a measure of relationship we can correlate the number of mutants with each of the other columns (plus the product of X and Y) and obtain the

statistics shown in table 3-2. As one can see the number of mutants seems to be most highly correlated with the product of X and Y.

N	X	Y	XY	E	V
.950	.978	.826	.999	.975	.798

Table 3-2: Correlation coefficients for mutants

This high relationship is undoubtedly related to the "replace every member of X with a member of Y" nature of the data mutations, since data mutations account for, on the average, 82% of all mutants generated (operator mutations making 5% and statement mutants 13%). We therefore computed the correlation coefficients for each of the major categories of mutants separately, obtaining the statistics shown in table 3-3.

	DATA	OPERATOR	STATE
N	.946	.953	.940
M	.999	.953	.977
X	.980	.993	.921
Y	.836	.874	.722
XY	.999	.961	.970
E	.970	.880	.999
V	.795	.880	.764

Table 3-3: Correlation coefficients by mutant type

We find that, as we suspected, the data mutants are most highly related to XY. However, the operator mutants correlated more highly with just X, and the statement mutants with E. Combining these figures together we found that the number of mutants can be approximated by the equation

$$M = 79 + .766\ XY + 4\ X + .0008\ E$$

This equation is, however, correlated only marginally better than the simple predictor XY (an increase of one in the fifth decimal place).

3.4 A sampling experiment

It has been observed that there is a great deal of redundancy built into the mutants, in that several mutants, although acting differently, will often all have the same effect. Although it appears that much of this redundancy is unpredictable, we can still make use of it by randomly generating only a small percentage of the mutants. (Often these redundant mutants could be detected algorithmically, but would require a significant amount of analysis. In designing EXPER there was a conscious choice made to allow redundant mutants rather than expend the time to detect them).

One experiment, designed by Sayward, was intended to measure the degree of this redundancy. In this experiment three programs where studied by three different subjects. Each subject analyzed each program generating 10% of the mutants for one, 25% for the second and 50% for the third. Each program was studied at each percentage level. After adequate test data had been constructed which eliminated all the nonequivalent mutants at the lower percentage level, the test data was used to execute all the mutants. The number of nonequivalent mutants not caught could then be expressed as a percentage of the total number of mutants. The results were as shown in table 3-4.

PERCENT GENERATED	10	25	50
PROGRAM A	0.63%	0.37%	0.12%
PROGRAM B	0.80%	0.27%	0.28%
PROGRAM C	0.82%	0.28%	0.27%

Table 3-4: Results of a sampling experiment

Even with test data generated using only 10% of the mutants no

more than 1% of the nonequivalent mutants were overlooked. The resulting percentages were much lower than expected. In order to investigate this further, all the programs analysed in the reliability study reported in section 4.2.3 were first analyzed using only 10% of the mutants. In each case it was found that test cases so developed eliminated over 99% of all nonequivalent mutants. This suggests very strongly that 1) more effort could be expended to delete and eliminate redundant mutants, and 2) generating even a small percentage of the mutants is a useful heuristic for evaluating and constructing test cases in practice.

3.5 A discussion of a similar system

Independently of the work on the mutation system at Yale, a system with several similar capabilities was being constructed at the University of Maryland by Richard Hamlet [40, 41]. Although there are similarities in goals, there are several major differences between EXPER and the Maryland systems. The mutation system was designed to be highly interactive and iterative, so that the programmer can enter a few test cases, observe their effect, and then enter more test cases. In contrast, Hamlet's system is based around a batch compiler; Test cases are typed as source statements along with the program and are executed by the complier, so that errors are reported much as syntax error would be.

Since it was an experimental system, Hamlet's system was purposely a rather limited implementation. In particular the only

data types in the language which he processed (a version of ALGOL) were integers, and the routines to be tested were limited to integer functions with a single integer argument. Because test cases were entered alongside the source statements, it was necessary that the description of a test case be succinct, for this reason the restriction to single integers was significant.

Hamlet's system operated by keeping a history of the values of expressions as they occured in the execution of the program on the test cases. In this respect the system is similar to the testing and debugging system created by Fairley [25]. After all the test cases have been executed, this history is then analyzed to check for several conditions, including a) every statement has been executed, b) every variable has taken on multiple values during its existence (this is looking at a single variable across all the locations it is altered in the program), c) every expression has taken on multiple values across all test cases (this is looking at a single location in the program across all test cases). In addition, for each expression in the program the system does an exhaustive substitution of simpler expression, to insure that the full complexity of the expression is required.

Some of these features are directly implemented in mutation analysis, for example the ability to detect that every statement is executed. Some are partially implemented, for example the branch analysis mutants (section 3.2.3) insure that at least branch

expressions vary across all their values. It is significant that Hamlet singles out branch expressions as being the most important example of how errors are detected in this manner.

A slightly more general method to achieve Hamlet's goal (c) could be envisioned for mutation analysis as follows: A special array exactly as large as the number of subexpressions computed in the program is kept, with two additional tag bits for each entry in this array. Initially all tag bits are off, indicating the array is uninitialized. As each subexpression is encountered in turn the value at that point is recorded in the array and the first tag bit is set. Subsequently when the subexpression is again encountered if the second tag bit is still off the current value of the expression is compared against the recorded value. If they differ the second tag bit is set. Otherwise no change is made.

In this fashion by counting those expressions in which the second tag bit is OFF and the first ON one can infer which subexpression have not altered their value over the test case executions, and hence one can construct mutations to reveal this. However, in view of Hamlet's comments on the number of errors caught in this fashion, one might decide this was not worth the effort.

An approximation to Hamlet's substitution of simpler expressions is achieved by the use of the operators leftop and rightop (described in section 3.1.2). Mutation analysis, however, makes the basic

assumption that the analysis of a small number of carefully chosen alternatives will actually achieve as much as an exhaustive analysis of all possible alternatives (the coupling effect). For this reason the mutation system examines far fewer alternatives. In one example program Hamlet's system generated around 8,000 alternatives, whereas the mutation system only considered about 350.

CHAPTER 4

EMPIRICAL STUDIES

It is highly unlikely that a concise theorem, of the type developed in chapter two, can ever be proved for a large set of natural errors in any realistic programming language like FORTRAN. This is not only due to the presence of a large number of formally undecidable problems [18, 31, 40, 52], but is also supported by much simpler arguments.

1. To be tractable, a testing method cannot be exhaustive, in the sense of trying every possible state vector at every statement in the program. Assume we have a finite set of test inputs for a program containing the predicate C. As was done in section 2, consider testing to be a game played against an adversary. The adversary can look at the set of values of the state vectors at the point C was evaluated, and divide them into two sets: those where C evaluated TRUE and those where C evaluated FALSE. All the adversary need do to find a program which is close to the original (differs from it in only one place) and which is likely not equivalent to the original, but which agrees on the proposed test data, is to find a condition D

which is independent of C but which also evaluates TRUE on the first set. The adversary can then reveal that in the "correct" program the predicate is (C AND D).

The two programs are likely not equivalent, but are indeed very similar. Since FORTRAN is so expressive it is highly likely such a condition can be found. In our inductive model of the world we have no more reason to believe in the one than in the other. An example of this type of error is encountered in the NAUR program described in appendix C.

2. One might suspect that predicates cause trouble, but even in the case of straight line programs we can have difficulty. Consider the two programs shown in figure 4-1 below. The programs are very similar to each other (they differ in only two places), and one might competently be considered an approximation to the other. Yet they are not equivalent. The programs compute the same answers on the set {0 0 0 0, 1 3 4 8, -1 -3 -4 -8}, furthermore these three inputs eliminate all mutants in the EXPER FORTRAN mutation system. Although this proves nothing, (other than being our first example of where EXPER can fail), it shows that even in the very restricted class of straight line FORTRAN programs with only addition and subtraction tremendous difficulties can be encountered.

What then saves us from a morass of intractability?

The saving grace is that programmers are usually not adversaries,

```
FUNCTION E(A,B,C,D)          FUNCTION E(A,B,C,D)
F = B+C                      F = B+A
G = A-F                      G = A-F
E = G+D                      E = G+D-B
END                          END
```

Figure 4-1: Two programs which are close but not equivalent

hiding inscrutable mistakes in obscure locations. Most errors a programmer makes are simple in form, well understood and classifiable. Furthermore we are willing to live with something less than assurances of total correctness. If we observe that the program works correctly for a large number of well designed inputs and that the program has been reasonably well exercised then we are usually willing to believe it is correct (always keeping in mind the small possibility we may be wrong). The fact that these goals are not perfect but are attainable is why testing has been and will continue to be established practice among programmers.

Given a testing tool, like the EXPER system, which we know is not perfect, experimental studies are of great importance in establishing how well we can expect the tool to work in practice. People are quite willing to live with non perfection, for example compilers that on rare occasions fail, as long as the number of failures are small in relation to the number of successes. Similarly the lack of perfection in the mutation system should not dishearten us, but rather should encourage us to examine how it works on the types of errors commonly

encountered.

Notice that, as we saw in chapter one, the mutation methodology exists quite independently of the EXPER mutation system. Experimental studies must be of a specific system, in this case the EXPER system. These only roughly approximate an evaluation of the method in general. It is possible that an alternative system could be constructed using the mutation idea but with quite a different set of mutant operators. Such a system could perform in a radically different manner from EXPER, with a result that the empirical results reported here could be similarly changed.

The selection of mutant operators described in the last chapter is also not static. Several of the operators mentioned in chapter 3 were not present in the original implementation of EXPER, and were added as experience suggested they might be useful. There is no evidence that this trend has halted. Further experience with the system and its error detection capabilities will undoubtedly suggest new operators which might significantly alter the statistics reported here for experimental studies.

4.1 An example of the coupling effect

This section will illustrate a representative case of coupling in a FORTRAN program. The program is adapted from the IBM scientific subroutines package [59], a collection of statistical and scientific programs in common use. The error was artificially inserted in a

study by Gould and Drongowski [38]. The error occurs in the line which reads

```
                              40 INN = UBO(3)
```

but which should read

```
                              40 INN = UBO(2)
```

```
      SUBROUTINE TAB1(A,NV,NO,NINT,S,UBO,FREQ,PCT,STATS)
      INTEGER INTX
      REAL TEMP, SCNT, SINT
      INTEGER INN, J, IJ
      REAL VMAX, VMIN
      INTEGER I, NOVAR
      REAL WBO(3), STATS(5), PCT(NINT), FREQ(NINT)
      REAL UBO(3), S(NO)
      INTEGER NINT, NO, NV
      REAL A(600)
      NOVAR = 5
      DO 5 I=1, 3
 5    WBO(I) = UBO(I)
      VMIN = 0.1000000000E+11
      VMAX = -0.1000000000E+11
      IJ = NO * (NOVAR - 1)
      DO 30 J=1, NO
      IJ = IJ + 1
      IF(S(J)) 10,30,10
 10   IF(A(IJ) - VMIN) 15,20,20
 15   VMIN = A(IJ)
 20   IF(A(IJ) - VMAX) 30,30,25
 25   VMAX = A(IJ)
 30   CONTINUE
      STATS(4) = VMIN
      STATS(5) = VMAX
      IF(UBO(1) - UBO(3)) 40,35,40
 35   UBO(1) = VMIN
      UBO(3) = VMAX
 40   INN = UBO(3)
      DO 45 I=1, INN
      FREQ(I) = 0.0000
 45   PCT(I) = 0.0000
      DO 50 I=1, 3
 50   STATS(I) = 0.0000
      SINT = ABS((UBO(3) - UBO(1)) / (UBO(2) - 2.0000))
      SCNT = 0.0000
      IJ = NO * (NOVAR - 1)
      DO 75 J=1, NO
      IJ = IJ + 1
      IF(S(J)) 55,75,55
 55   SCNT = SCNT + 1.0000
      STATS(1) = STATS(1) + A(IJ)
      STATS(3) = STATS(3) + A(IJ) * A(IJ)
```

```
      TEMP = UBO(1) - SINT
      INTX = INN - 1
      DO 60 I=1, INTX
      TEMP = TEMP + SINT
      IF(A(IJ) - TEMP) 70,60,60
   60 CONTINUE
      IF(A(IJ) - TEMP) 75,65,65
   65 FREQ(INN) = FREQ(INN) + 1.0000
      GOTO 75
   70 FREQ(I) = FREQ(I) + 1.0000
   75 CONTINUE
      IF(SCNT) 79,105,79
   79 DO 80 I=1, INN
   80 PCT(I) = (FREQ(I) * 100.0000) / SCNT
      IF(SCNT - 1.0000) 85,85,90
   85 STATS(2) = STATS(1)
      STATS(3) = 0.0000
      GOTO 95
   90 STATS(2) = STATS(1) / SCNT
      STATS(3) = SQRT(ABS((STATS(3) - (STATS(1) * STATS(1))
     * / SCNT) / (SCNT - 1.0000)))
   95 DO 100 I=1, 3
  100 UBO(I) = WBO(I)
  105 RETURN
      END
```

There are a number of mutants which cause the programmer to generate test inputs which uncover this error. Consider, for example, the one which changes the statement

IF (A(IJ) - TEMP) 75,65,65

to

IF (A(IJ) - 1.000) 75,65,65

Control reaches this point only if A(IJ) is bigger than TEMP, so control always passes to 65. By tracing the flow of control we can discover that TEMP is equal to the value of the input parameter UBO(3) at this point. To eliminate this mutant then we must find a value where A(IJ) is less than one but larger than UBO(3). Therefore UBO(3) must be less than one. There is nothing in the specifications which

rules out UBO(3) being less than one, but the error causes UBO(3) to be assigned to the integer variable INN. All the feasible paths which go through the mutated statement also go through label 65, which references FREQ(INN). Since INN is less than or equal to zero, this is out of bounds, and the error is discovered.

4.2 Reliability studies

A method one might consider for evaluating a tool such as the program mutation system would be to use an experiment similar to those used in psychological studies, such as the double blind technique. Using this method one has a group of subjects which varying levels of programming and testing skills and a group of programs which have zero or more errors known only to the experimenter. Each subject reports on the errors detected in trying to pass the mutant test. Analysis of variance or similar statistical techniques can then be used to evaluate the results.

Unfortunately there are two serious difficulties which prevent one from using a technique such as the one just described. The first is the high cost of performing such controlled multi-subject experiments. The second, and more serious difficulty is the problem of factoring out those errors caught as a direct consequence of the method from errors caught by other means, such as merely reading the listing. (Holthouse et al [49] mention this difficulty and describe these errors as being caught by the 'peripheral vision' of the human tester.)

In order to remove this second difficulty in comparing various testing methods we need a uniform notion of when a method has discovered a particular error. Howden calls this idea reliability and defines it as follows:

> If the use of a program testing technique is guaranteed to always reveal the presence of a particular error in a program, then the technique is said to be reliable for the error. [53] (italics mine)

We have taken the word guarantee as it is used here to mean that the method itself, no matter who applies it or their level of programming expertise, must somehow insure that the error will be discovered.

Notice that there is a certain artificiality introduced concerning whether test data which shows the program is incorrect actually shows the presence of a particular error. As noted by Gannon [32], in the presence of such test data a possible, and indeed likely, outcome is that the wrong cause will be diagnosed and an incorrect fix applied. We have tried to avoid a discussion of this problem by making the definition that a set of test inputs (D) reveals the error E if upon removing or correcting E the test inputs D are correctly processed.

In order to achieve this extremely stringent requirement, we devised the following experimental method: The goal of the method is to achieve one of two possible outcomes. Either

1. a set of test cases is developed which are processed

correctly by the erroneous program and which eliminate all non equivalent mutants (in which case we say the error is not reliably caught by mutation analysis), or

2. a set of mutants is discovered with the property that the only inputs which cause an observable difference in the erroneous program and the mutant programs also cause an observable difference in the erroneous program and the correct program. In this case we say that mutation analysis would reliably uncover the error, since any data which eliminated one of these mutants would discover the error.

The method used to achieve these goals was as follows: An experienced programmer would be given the incorrect program and would have total knowledge of the location and nature of the errors it contained. Choosing a mutant he would attempt to find a test case for which both the correct and erroneous program agreed but which differentiated the erroneous program from the mutant program. The construction of test data in this fashion puts mutation analysis in the worst possible light, in that the tester is forced to act as an adversary and find the least meaningful sets of inputs. Often this involves a detailed analysis of the effects of a certain error. For example, one of the programs in the second study computes statistics for vectors of inputs; For a vector of three numbers, the correct answer is produced only if $5X^2-5XY+5Y^2-5YZ+5Z^2-5ZX=9$. While it is possible that another test generation method would only construct inputs which satisfied this constraint, it seems extremely unlikely. (Because of the often significant amount of work which must be done to find this absolutely worst case inputs, Professor Sayward has coined the slightly more picturesque term Beat the System Experiments for

this type of study.)

Having found test data to eliminate this mutant, the tester would then execute all the mutants on this test case (probably eliminating a large number of other mutants in the process). If it was not possible to find such a test case another mutant was chosen. This process was iterated until one of the two goals given above was achieved. This is a worst case type of analysis, in that any other method of generating test data must necessarily find at least the errors found in this fashion, and very likely many of the others also.

This type of experiment is actually an extension of the reliability studies performed by Hamlet [42] and Howden [52]. These earlier studies, however, were directed at comparing two or more competing methodologies, and deriving statistical information of the form "on the following samples of programs method A discovered X% of the errors and method B discovered Y%." In the following experiments we were much less concerned with the number of errors caught and much more concerned with the type of errors missed. Furthermore this information was not used to compare two methods but was designed to evaluate the mutation analysis system (EXPER) and to direct the search for new mutant operators which would improve the system.

For example, several of the programs studied revealed that a significant number of errors in FORTRAN were caused by programmers treating the DO statement as if it were an ALGOL FOR statement,

forgetting that no matter what the limits are, a DO statement will always (perhaps erroneously) execute the loop body at least once. The way we chose to detect these errors was to introduce a mutant which changed a DO statement into a FOR statement, bringing this fact to the programmers attention and forcing him to derive data which indicated he had knowledge of this potential pitfall.

Using this methodology, several experiments were conducted measuring the reliability of mutation analysis. Both the first two studies were based on data from previous studies by Howden [54, 56].

4.2.1 An experiment using program fragments from Kernighan and Plauger

This study was based on 12 program fragments from the "common blunders" chapter of Kernighan and Plaugers book The elements of programming style [60]. A description of each of the errors is contained in appendix A.

William Howden had previously studied these program fragments in an attempt to compare the error detection capabilities of symbolic evaluation and path testing. In symbolic evaluation a program is executed symbolically rather than with numerical values and the user is given the symbolic output which can, presumably, be checked against a specification for correct output. The second method studied by Howden, path testing, involves finding data which executes every feasible path (up to iterations of loops) and which iterates each loop at least twice. For each such path test data was generated randomly.

By generating the test data randomly Howden's study was not, in the strict sense of the definition given in the last section, a reliability study. A reliability study is, however, an example of worst case analysis, in that any other method must necessarily give better results. Hence the figures reported by Howden can be considered as bounding what one would expect from a reliability study.

Howden found that symbolic evaluation would detect 13 of the 22 errors (15 if a more graphic method of presenting the symbolic output were used). Path testing would detect only 9. Combining the two method one would detect 16 of the 22 errors. Using the definition of reliability described in the last section we were able to demonstrate that mutation analysis would necessarily discover 20 of the 22 errors. These results are given in table 4-1, and further information on the particular errors and their discovery can be found in appendix A.

SYMBOLIC EVALUATION	13/22
PATH TESTING	9/22
BOTH COMBINED	16/22
MUTATION ANALYSIS	20/22

Table 4-1: Howden's data combined with that for Mutation Analysis

In [54] Howden describes some of the errors not caught by the two methods he studied. For example in one case there is the computation

```
J = MARKS(I)-1/10 +1
```

where a pair of parenthesis have been erroneously omitted around the formula MARKS(I)-1. This is one of the errors which would not be detected by symbolic evaluation unless a special two dimensional output was used. To see how this error is caught by mutation analysis

note that in FORTRAN the expression 1/10 is equal to zero. By making the mutation which replaces 1/10 with 0/10 the error is quickly revealed.

Another error not revealed by either symbolic evaluation or path testing involved the omission of an absolute value operator from an expression. One of the mutants generated is, however, exactly the correction needed to repair this error. Hence the error is again easily discovered.

There are two errors which are not caught in this experiment. The first involves two adjacent statements which should be interchanged. Note that we could have chosen to make a mutant operator which interchanged statements, in which case this error would have been caught. Because in so many cases one can interchange statements with no effect on the program we chose not to make this operator. This illustrates the fact that a slightly different set of operators could radically alter the results reported in these experiments.

The second error involves strict equality being used with real variables when a fuzzy equality which avoids round off problems should be used. It is much more difficult to find a mutant operator which could conceivably discover this error.

4.2.2 An experiment using four programs from a study by Howden

The four programs analyzed in this study were taken from the report on an experiment conducted by Howden for the National Bureau of Standards [56]. All the programs had previously appeared in the literature, and are described in appendix B. In this study Howden considered a number of different test data generation methods to determine which would reliably uncover errors in six different programs.

It was our desire in undertaking this study that the data presented by Howden would serve as a useful benchmark by which the capabilities of mutation analysis could be evaluated. Unfortunately, two of the programs in Howden's study were written in COBOL and PL/1 and depended heavily on Fixed Decimal, Picture type data, or ON conditions. The fact that these issues do not arise in FORTRAN and cannot be easily simulated meant that these two programs had to be excluded from this study. Even more disturbing was the fact that the two programs excluded accounted for 23 of the 28 errors (or 82%) considered by Howden. (The COBOL program contained 3 errors, all of which are caught by branch analysis. Since mutation analysis subsumes branch analysis these would all have been caught by mutation analysis. Because of the presence of ON conditions the other PL/1 program could not be evaluated.)

This left us with four programs containing a total of five errors. Although this sample was much too small for us to draw any

definitive conclusions, it is hoped that because of the number of methods studied by Howden some idea of the relative strengths of the methods can be gathered.

Howden analysed six different testing methods: Path testing is a technique which requires that each executable path through the program be executed at least once. (This definition seems to differ from the definition used by Howden in the earlier study.) This technique is not practical since a program may have an infinite number of paths, but it does give an upper bound on the reliability of techniques that require testing of some subset of the set of all paths. Branch Testing requires that each branch be tested at least once for all its possible outcomes. Structured testing assumes that the program consists of a hierarchical structure of small functional modules. Each path through a functional module which executes loops less than 2 times is tested at least once. Special values testing is a collection of rules which experience indicates are important for finding good test data. Examples of such rules are that each expression should, if possible, evaluate to zero, that different elementary items in an input data structure have distinct values, plus rules specific to the program under test. Anomaly Analysis does not execute the program but rather looks at the code for suspicious looking constructs. Finally, Specification Requirements constructs test cases only from the specifications, and not from the code itself.

The single error in this study which mutation analysis failed to

PATH ANALYSIS	4/5
BRANCH ANALYSIS	0/5
STRUCTURED TESTING	3/5
SPECIAL VALUES	4/5
ANOMALY ANALYSIS	0/5
SPECIFICATION REQUIREMENTS	3/5
MUTATION ANALYSIS	4/5

Table 4-2: Number of errors caught versus testing method

detect (which is described in appendix B), can be characterized as a missing path error. For methods which, like mutation analysis, are based on an examination of the code, these are indeed the most difficult type of errors to detect. It is interesting to note that those methods which construct test data from a description of the program and not the code itself (specification requirements and special values testing) do well at discovering these errors. This would seem to imply that a combined method would be most desirable, where an initial core of test cases would be constructed just from the specifications, and then this core could be expanded to correct weaknesses as demonstrated by mutation analysis. Such a combined testing strategy might prove very effective.

Appendix B contains more detailed information on each of the programs and how the errors are detected.

4.2.3 Further reliability studies

Mutation analysis is unique in that by an appropriate choice of new mutant operators the method can in practice be significantly improved. This notion of reliably uncovering errors gives us important information which can be used to help direct the search for these new operators. To understand this, note that in studying a

given program and a given set of errors, information is obtained no matter what the outcome:

a) If the errors are reliably found, insight is gained into why the method works. This is because the result is an actual set of mutants which force the discovery of the error. Some connection between mutants and errors can then be formulated.

b) If the method fails to reliably find the error, then the weakness so shown can be used to direct the search for new mutant operators.

It is in this manner that many of the mutant operators described in the last chapter have been discovered and added to the EXPER system.

It is precisely because such a useful store of information can be discovered that we have continued to run these reliability experiments on other programs.* To date thirteen programs have been analyzed. These programs contained a total of 30 errors. Of these 30 mutation analysis, as characterized by the EXPER system, would discover 25. Further information on the programs is contained in appendix C.

---

*Several of the programs studied here were actually analyzed, under my direction, by Mr. Robert Hess. I am extremely grateful for his assistance.

It is difficult to construct a classification scheme for error types which is neither so specific that each error forms its own type nor so general that important patterns cannot be detected. If the classification is based on logical mistakes it is often hard to relate errors to mistakes in the code. On the other hand it seems difficult to base a scheme just on mistakes in the code, since often a single logical mistake will be responsible for changes in several locations in the program. Goodenough and Gerhart [37] and Howden [52] among others have attempted to construct a generally applicable system. Neither of these systems give a sufficiently intuitive picture of the errors in any particular class. Therefore we have chosen to group the errors in these thirteen programs into the following categories:

Missing Path Errors: These are errors where a whole sequence of computations which should be performed in special circumstances are omitted.

Incorrect Predicate Errors: These are errors which arise when all important paths are contained in the program, but a predicate which determines which path to follow is incorrect.

Incorrect Computation Statement: These are errors which arise from a computation statement which is incorrect in some respect.

Missing Computation Statement

Missing Clause in Predicate: This is a special case of an

incorrect predicate error, but since it is so hard to detect we give it special treatment.

The 30 errors in these 13 programs range from simple to extremely subtle errors. Due to the worst case nature of reliability studies the fact that 5 errors are not discovered does not mean that these errors would always remain undiscovered if mutation analysis were used, merely that we cannot guarantee the discovery. Table 4-3 gives the number of errors detected by error type. Of these 30 errors, only 11 would be caught using branch analysis.

	NUMBER	CAUGHT
MISSING PATH ERROR	7	6
INCORRECT PREDICATE ERROR	4	3
INCORRECT COMPUTATION STATEMENT	15	14
MISSING COMPUTATION STATEMENT	3	2
MISSING CLAUSE IN PREDICATE	1	0

Table 4-3: Number of errors detected versus error type

One can notice that in three of these categories the errors are caused by the lack of certain constructs in the program. Since the testing method is being asked to guess at something which is not in the program, we should be surprised that it does as well as indicated. None the less, missing path errors and missing clauses in predicates are probably the most difficult errors for any testing method to discover.

Table 4-4 shows the number of errors which are detected broken down by operator type. Two figures are given for each mutant operator type; The first is the total number of errors detected by mutants of that type, and the second represents the number of errors identified

by this operator and only this operator. From this table we can gain some idea of the relative strengths of the mutant operators.

	TOTAL	UNIQUE
Constant replacement	5	1
Scalar Variable Replacement	4	0
Scalar Variable for Constant Replacement	5	1
Constant for Scalar Variable Replacement	5	0
Source Constant Replacement	3	1
Array Reference for Constant Replacement	2	0
Array Reference for Scalar Variable Replacement	3	0
Comparable Array Name Replacement	2	0
Constant for Array Reference Replacement	3	0
Scalar Variable for Array Reference Replacement	2	0
Array Reference for Array Reference Replacement	1	0
Data Statement Alteration	0	0
Unary Operator Insertion	4	0
Arithmetic Operator Replacement	4	1
Relational Operator Replacement	1	1
Logical Connector Replacement	0	0
Absolute Value Insertion	5	2
Statement Analysis	6	5
Statement Deletion	0	0
Return Statement Replacement	2	0
GOTO Label Replacement	4	2
DO Statement End Replacement	2	1

Table 4-4: Errors detected versus mutant classification

The low figures for logical connector replacement and data statement alteration are due more to the lack of these constructs in the programs being tested than to any fundamental weakness in these operations. The high figure for the number of errors caught only by statement analysis is due to the fact that the other mutant operators were enabled only after all statement analysis mutants were eliminated (i.e. all statements had been executed).

The observation we can make is that we again see a strong redundancy in the mutant operators, particularly in the operand mutations. This redundancy is less noticeable, although still present

in some degree, in the operator and statement mutations. This remark might lead us to conclude that in sampling mutants (see section 3.4) we should weight the different mutant types; Perhaps generating all the statement and operator mutants and only sampling the operand mutations.

We can envision using this data to construct a procedure which would minimize the expenditure of machine and human resources in the discovery of errors; However there is an important point to be noted in this regard, which is that while logically two mutants may have the same error detecting power, psychologically they may be vastly different. For example mutants which in effect say "this statement has never been executed", "this statement can be deleted", or "this relational or logical operator can be replaced with the constant TRUE" pinpoint an error much more directly and forcefully than one which says "This expression can be incremented by one and the same result will be produced." This psychological argument would seem to imply that the first mutant types to be enabled should be statement analysis, statement deletion, goto label replacement (used in branch analysis), and the arithmetic, relational and logical operator replacements.

The choice of which mutant operators to apply next seems to be a trade off between human and machine resources. Operand mutations seem to have a much more immediately assimilable meaning than do either the remaining statement or operator mutations. On the other hand, operand

mutations typically comprise over 80% of all mutants generated. If we want to minimize machine resources (i.e. the number of mutants executed before an error is found), we would therefore enable first all the operator and statement mutants, and only once they have been analyzed consider the operand mutants. On the other hand if we want to minimize human resources (i.e. the amount of time the tester must spend analyzing mutants and constructing test cases), then we would do just the opposite. It seems difficult to decide without further information which order would be best in practice, as each individual situation would dictate its own solution.

4.3 Testing large systems

In an attempt to discover if the testing of large hierarchies of programs presented any serious difficulties not encountered in testing small single modules, several parts of the EXPER system were tested using the system on itself. These parts consisted of two large groups of subroutines from the parser, each approximately 1,000 statements in length (not counting comment cards). Because of time and space limitations not all the subroutines could be tested, hence only the most central and critical routines were selected (in all about half the total number of statements were tested).

The experiment was quite rewarding as over a dozen errors were discovered in the EXPER system. The large number of errors was surprising since at the time the experiment was conducted the system had been in operation for over a year and a half. Examples of the

type of errors discovered were: passing the wrong number of parameters to procedures, or passing incorrect parameters, declaring variables the wrong type, referencing out of bounds array indices, getting stuck in infinite loops on error conditions, dead code which could not be executed, and a confusion over whether columns 72 to 80 could contain valid FORTRAN statements.

The difficulties encountered in this experiment were much more a consequence of managing a large network of subroutines, and not particularly related to mutation analysis. Most of these difficulties have been noted previously by other authors [49]. Examples of the problems encountered are:

A) Defensive coding. This is responsible for sections of code marked "Hope this never happens but if it does do the following." It seems clear that defensive coding encourages reliability, especially since about half the time the assumption that the code can never be executed proves to be wrong. But if the assumption is right then the unexecuted parts of the code can be mutated in any fashion with no effect.

B) Portions of code which, while executable, are difficult to reach because they require an inordinately large input space or too much CPU resources to duplicate.

C) Routines which are also used by other sections of code not being tested. In one case during this experiment a predicate was

discovered which could not be forced to be true. A fix was then applied which removed the offending statement. A month later it was discovered that the subroutine in question was used elsewhere and the now absent predicate was of critical importance.

D) Starting over again. Once testing has commenced the code is regarded as unalterable. If errors are discovered and the program must be changed, one is forced to start the entire testing process over again. This means that all the previously developed test cases must be rerun and all the previously eliminated mutants dealt with again.

If one were at attempt to construct a commercially viable mutation testing system, all of these problems would have to be dealt with. In spite of these difficulties this experiment did prove that mutation analysis could be applied to medium to large software systems. The difficulties involved in using mutation analysis seem no more severe than those involved in any other testing method.

4.4 The lifespan of an average mutant

An important observation to keep in mind when considering the cost of mutation analysis is that about 80% of all mutants die the first time they are encountered, no matter how good or bad the test data is. This means that at worst only about 20% of the mutants generated will require lengthy investigation.

The reason for this high attrition rate seems to be a striking

nonuniformity with regards to how mutants die in the various test cases. In the EXPER system there are eight ways a mutant can die: by computing the wrong answer, by receiving an arithmetic fault, by computing a subscript index out of bounds, by executing a trap statement, by referring to an undefined variable, by attempting to divide by zero, by running for too long, and by attempting to change a read only variable. We have observed that in the first few test cases a high percentage of mutants die by means other than getting the wrong answer. The situation is thereafter reversed, when almost every remaining mutant which dies does so because it computes an answer different from the original program.

We have also observed that in achieving the goal of all non equivalent mutants being eliminated, about twice as many mutant executions are performed as there are mutants generated. This figure includes equivalent mutants which survive all test cases. If we eliminate these from consideration, then the average mutant survives about 1.5 test cases before being eliminated.

The last few mutants to be eliminated are, however, extremely recalcitrant. It is these mutants which are probably the most difficult and the most important to remove since they give the greatest insight into the functioning of the program. Typically, the last 50% of test cases are used to eliminate the last 2-10% of the mutants.

4.5 The problem of equivalent mutants

In the first chapter it was noted that a major stumbling block to the application of mutation analysis is the problem posed by mutants which are equivalent to the original program. In chapter two we saw that this same problem was of some concern in the theoretical studies. In practice equivalent mutants are a nuisance, but for an entirely different reason. It is not that equivalent mutants are difficult to discover, but that they are so prevalent and simple minded that they get in the way of the more important aspects of testing.

Typically between 4 and 10% of the mutants generated are equivalent, with heavy clustering towards the 4. The equivalent mutants are not, however, distributed with the same ratios as all mutants. In fact, a very small number of mutant types account for a disproportionate number of equivalent mutants. The following table gives some typical figures. The first column gives the percentage of equivalent mutants which the equivalent mutants of the given type represent, and the second column gives the same percentage for all mutants.

	PERCENT OF EQUIVALENT	OF ALL
ABSOLUTE VALUE INSERTION	75	4.0
GOTO REPLACEMENT	12	0.7
RELATIONAL OPERATOR REPLACE	7.5	0.5
ALL OTHER MUTATIONS	5.5	0.5

Table 4-5: Percentage of equivalent mutants versus mutant type

In order to investigate how difficult it would be to construct an automatic system to eliminate these mutants we defined several levels

of difficulty. An automatic system could easily be constructed to remove mutants of the first three levels. Mutants of level four could, in principle, be eliminated but the costs might be prohibitive. Mutants of level 5 probably could not be eliminated algorithmically.

TYPE 1: These are mutants eliminable by
a) noting that if a parameter has a variable upper bound, the value of the upper bound variable must be strictly positive, and
b) Noticing the values on DO loop limits, for example if I=1,10 then for the extent of the loop I is positive and between 1 and 10.

TYPE 2: These are mutants eliminable by examining the statements in the immediate proximity of the mutated statement, in particular no further removed then the last multiple entry point (labelled statement or DO loop start).

TYPE 3: Eliminable by noting that if a variable is initialized to a non negative (strictly positive) value and always incremented then it will remain non negative (strictly positive).

TYPE 4: These are mutants which are eliminable in theory but would require a symbolic executor system to trace a large number of feasible paths.

TYPE 5: Finally, these are mutants which require a deep understanding of the algorithm, knowledge about number theory, or other real world knowledge generally beyond the scope of automatic

analysis.

By level of difficulty, equivalent mutants typically group as shown in table 4-6. As can be seen, generally well over 70% of the equivalent mutants can be detected by the most rudimentary automatic procedures. Most of the remaining 30% could, in principle, be eliminated automatically hence are probably easy for humans to recognize. Generally less than 3% of the equivalent mutants (0.14% of all mutants) require a deep understanding of the program or programming process to be eliminated.

LEVEL	PERCENT OF EQUIVALENT	OF ALL
1	31.1	2.3
2	2.8	0.13
3	40.8	2.0
4	22.9	1.4
5	2.4	0.14

Table 4-6: Percentages of mutants versus level number

Baldwin and Sayward [3] have discussed the use of traditional program optimization methods in the detection of equivalent mutants. While powerful, these methods do not seem to be directly applicable to absolute value insertion mutations, which table 4-5 shows are the most common form. A simpler method would probably suffice.

So as not to leave the impression that the problem of equivalent mutants is trivial, note that often those few mutants in the type 5 category are extremely subtle. During the course of the sampling experiment discussed in section 3.4 there were several extended discussions concerning whether certain mutants were or were not equivalent. There is even a program containing two mutant changes

which was published and asserted to be equivalent [65], however later investigation proved this not to be the case. The saving grace is that these examples are rare.

CHAPTER 5

DIRECTIONS FOR FUTURE RESEARCH

Mutation analysis is a recent innovation. Because it represents a new solution to some very general problems, the method has numerous aspects which I have not pursued in this thesis. In this chapter I will specifically mention five areas of possible future research:

1. The use of symbolic execution to generate test cases automatically.
2. Preprocessing or postprocessing the program to reduce the number of mutants generated.
3. Expanding the concept of test case to include more than just input/output behavior.
4. The analysis of new and different mutant operators.
5. The application of the mutation analysis paradigm to other problem domains.

5.1 Using symbolic execution to generate test cases

Symbolic execution is another testing method which has been extensively studied [6, 20, 54]. In this method, variables are treated as algebraic unknowns, and a specific path through a program is interpreted symbolically, producing an equation (or several equations) expressed in terms of these unknowns. The equations can then be solved by some automatic means to derive test data which

follows this path. In essence this is what a human tester must do, the difference being the machine is now used to derive the test data.

Symbolic execution is costly and there are problems connected with solving the resultant equations, but a greater shortcoming is the fact that the goal is very weak. In most systems that have been proposed the goal is a set of test cases that execute every statement. As I have discussed in section 3.2.2, a test set of this nature gives us very limited knowledge about whether the program is correct. This is most strikingly illustrated by the case of straight line code, where often a test case consisting of all zero inputs can execute every statement while telling us next to nothing about the program.

Some symbolic execution systems have slightly stronger goals, and the test cases they generate are slightly better. For example the ATTEST system [21] uses the symbolic information to preclude zero divide, index overflow or underflow, computed goto out of bounds, and variable dimension out of bounds. But problems such as predicate errors (section 3.2.5) and coincidental correctness (section 3.2.9) may still pass undetected.

Mutation analysis provides a goal for symbolic execution systems that is significantly stronger. Each mutant, in effect, presents a different goal for the symbolic execution system: that of finding test data to differentiate it from the original program. That is, the symbolic execution system can be used to generate test data that

eliminates mutants, a task usually left to the human tester. In the case of equivalent mutants such a task is impossible, but it might be possible to use the information obtained to prove equivalence. It can happen that a mutated statement is not locally equivalent to the original, and yet the program might be globally equivalent to the original. So a symbolic execution system would have to evaluate more then just local situations, tracing at least one program path from start to finish.

The use of symbolic evaluation in conjunction with mutation analysis could both increase the capability of symbolic evaluation and ease the problem of generating test cases for mutation analysis.

5.2 Reducing the number of mutants generated by EXPER

The sampling experiment described in section 3.4 suggests that there is a large amount of redundancy in the mutants generated by EXPER. An interesting question is whether this redundancy is algorithmic, that is, whether it might be possible to decide a priori which mutants are redundant and therefore unnecessary. We made a very limited attempt at this type of analysis with EXPER, writing a number of different rules concerning when not to generate mutants [14]. These rules, however, examine only the immediate neighborhood of the mutated section of code. It is possible that with more global information (such as might be obtained from the symbolic execution system described in the last section) a large number of mutants might be eliminated without any need for test cases. On the other hand, it

is equally possible that the cost of this analysis might far outweigh the cost of executing the redundant mutants.

5.3 Using more than input/output behavior in test cases

A testcase in EXPER is defined only by input/output behavior. But in many situations it is reasonable to assume that a user knows more than merely whether a given output is correct for a given input. For example he might be able to tell whether some intermediate values are correct, or he might be able to recognize a symbolic trace of the correct computation.

If we include this type of information the problems of testing may become significantly easier [4, 9, 57]. Recently Martin Brooks has analysed a testing procedure very similar to mutation analysis using program traces as an additional source of information [9].

5.4 New mutant operators

It is a certainty that the set of mutant operators described in chapter three is not perfect, and that the process of discovering new mutant operators will continue. One new direction is indicated by the recently created zero push operator (section 3.1.2). Other push mutants that could be envisioned are one push, blank push for characters, and an arbitrary constant push where the constant values are taken from the program. For analysis of numerical software we might want a big number push and a small number push, to insure that quantities are both larger than some fixed limit and smaller (in absolute value) than some quantity.

Section 3.5 mentioned a new operator of a type not currently implemented in EXPER, which measured the values of an expression at a specific point.

There is a danger in this game of making new mutant operators that we will significantly increase the cost of an analysis without significantly improving its capabilities. For this reason reliability studies of the type discussed in chapter four should be used in evaluating new operators, and a new operator should not be introduced unless it reliably detects at least one new error representing a class of errors committed in practice.

Finally this research might introduce a completely different type of mutant operator. One possibility is based on the observation that all the current mutant operators manipulate the code, but an equally important part of the program is the data. For example one might consider a mutant operator that would alter an input parameter by 10% of its value. This type of operator might not help in finding errors, but would be useful in evaluating the robustness of a software system.

5.5 Mutation analysis in other problem domains

In chapter one the notion of weighing inductive evidence by using mutation analysis was introduced in a framework quite divorced from computers and computer programs, and it is possible that mutation analysis might be used to analyze some very general logical theories.

On a practical level, one might ask whether mutation analysis on

other languages would be significantly different from the analysis of FORTRAN presented in this thesis. I doubt very much that ALGOL-like languages would present any new insights, but totally different languages like LISP, SNOBOL, APL and SETL might produce some surprises.

Various researchers are currently attempting to define a language that can be used to formally express the specifications of a program. Given such a language, one could conceive of a system that develops test cases using mutation analysis on the specifications, test cases that could then be used as a basis for generating more extensive data using mutation analysis of the program.

Another interesting direction would be to apply mutation analysis to a totally different form of testing, for example the testing of logical circuits. Assume we have a model of either the logical or physical components of, say, an LSI chip, and we can interpret the actions of this model on certain inputs. We could then consider mutants that altered the model in some way, perhaps related to design or fabrication defects, and search for test data that would detect these errors.

There is an interesting twist involved in the modelling of physical circuits: Because of the way circuits are made it is often not possible to insure that what is intended to be, for example, a .1 ohm resistor will not actually be .05 or .15 ohms. Hence one must

attempt to design devices that are impervious to such changes. Mutants that produce these changes are just the opposite of those we have been considering. In the paradigm which I have described up to now an uneliminated mutant indicates a potential error. Here, since all such changes should be transparent, if such a mutant is eliminated it indicates an error.

5.6 Summary

This thesis has examined many issues related to the problem of program testing, all unified by the mutation analysis paradigm introduced in chapter one. To provide some sort of summary, chapter one introduced into the usual inductive procedure a method for weighing the importance of test case observations. The method is quite general, and may have interesting applications quite unrelated to computer programming. Chapter one discusses how in the particular case of testing computer programs this method can be strengthened even further by observing the coupling effect and the competent programmer hypothesis.

Chapter two was devoted to showing that in some restricted domains the mutation analysis method can be used to formally prove the correctness of programs. In particular two examples, decision tables and linear recursive lisp programs, are studied in detail.

While the results of chapter two may have some limited applicability, the type of programs analyzed by these methods are

quite different from the type of program run on a typical day on an average computer. I have argued (chapter 4) that it is extremely unlikely that a concise clear theorem of the type developed in chapter two can be proved for a reasonable class of errors in any general (that is, Turing complete) programming language. In order to examine exactly what the capabilities of mutation analysis are in these more general settings, in chapter three I describe a system which applies mutation analysis to FORTRAN programs. Also in this chapter I show how the particular mutant operators this system uses can mimic several other testing methods.

Chapter four goes on to describe several experiments conducted with the aid of this system. I analyze the time and machine resources the system requires, difficulties involved in using it, its effectiveness in finding errors, and compare the method against other testing methodologies. I also show how the information obtained from these studies can be used to direct the search for new mutant operators, thereby improving the error detection capabilities of the system.

Mutation analysis is a tool. It does not immediately solve all the problems associated with testing, but it can be a significant help in the detection of errors and the testing of computer programs. It does provide something few other testing methods can, which is a quantitative estimate of test data adequacy.

The mutation analysis paradigm presented in this thesis is an example of an inductive, rather than the more widely studied deductive [43], means of increasing confidence in software. The field of inductive formalisms in computer testing is certainly not exhausted by this approach, and the need for discovery, comparison, and analysis of other test measurement methods should certainly provide an attractive research area for some time to come.

Appendix A

Errors from Kernighan and Plaugers chapter on Common Blunders

This appendix lists the 22 errors contained in the common blunders chapter of Kernighan and Plaugers book The Elements of Programming Style. All page numbers refer to the first edition.

There are six general ways in which errors are detected:

8 are caught merely as a consequence of the interpretation process,

5 are caught by spoiling coincidentally correct expressions,

2 are caught by the correct program being a mutant of the incorrect one,

2 are caught by domain pushing (inserting ABS statements),

2 are caught by predicate testing,

1 is caught by the branch analysis mutants.

The 22 errors are as follows:

1. Page 77. Sin routine, variable SUM is uninitialized. Caught by the interpreter.

2. Page 78. Sin routine, DABS operator needed, caught since this is a mutant.

3. Page 78. Sin routine, -1**(I/2) used instead of (-1)**(I/2). The exponent can be mutated to I/3 or I/1 or removed altogether with no noticeable effect.

4. Page 78. Sin routine, two statements interchanged. This error is not necessarily caught by mutation analysis.

5. Page 79. Current routine, uninitialized variable E.

Caught by the interpreter.

6. Page 79. Current routine, integer/real mismatch. Caught by the interpreter.

7. Page 80. Current routine, variable C not reset. Caught by branch analysis mutations, since when SC+CI .LE. TC (which must happen to eliminate all branch analysis mutants) the wrong answer will be produced.

8. Page 80. Current routine, fails to work when variable CI=0. Caught by zero push mutations.

9. Page 81. Expression NUM should be NUM(1). Gives a compiler error.

10. Page 81. Variables initialized with DATA statements are overridden. In EXPER variables in DATA statements default to read only unless otherwise marked.

11. Page 83. Program fails to work if exactly 46 transactions. Caught by changing the Greater than operator to Greater than or Equal.

12. Page 84. Greater than instead of Greater than or Equal needed. This is a mutation.

13. Page 84. Possible reference to undefined variable LOW(2). Caught by changing DO 12 I=2,N to DO 12 I=1,N.

14. Page 85. Possible error if B+C less than .01 . Caught by twiddling B+C by .01 .

15. Page 85. Loop exits out of both side and bottom. Caught by changing 60 to 61, forcing loop to go through 60 times.

16. Page 87. Search Program. Uninitialized Variables. Caught by interpreter.

17. Page 87. Search Program. Doesn't work for tables of one entry. Caught by changing (LOW+HIGH)/2 to (LOW+HIGH)-2.

18. Page 87. Search Program. Doesn't work when match is in A(1). Caught same as previous error.

19. Page 89. J=MARKS(I)-1/10 should be J=(MARKS(I)-1)/10 Caught by changing 1/10 to 0/10.

20. Page 90. Parenthesis missing around expression AN - 1.0. This will be caught by almost any data, in particular when an attempt is made to force (SUMSQ - (SUMX**2 / AN)) to be

zero.

21. Page 91. 10 times .1 is not 1. Any data will give wrong answer.

22. Page 93. Equality should be fuzzy. This error is not caught by mutation analysis.

Appendix B

Details of the four programs from a study by Howden

The first program is written in an ALGOL dialect and initially appeared in a paper by Henderson and Snowden [46]. It is intended to read and process a string of characters which represent a sequence of telegrams, where a telegram is any string terminated by the keywords "ZZZZ ZZZZ ". The program scans for words longer than a fixed limit, and isolates and prints each telegram along with a count of the number of words contained therein, plus an indication of the presence or absence of over length words. The program has also been studied in Ledgart [63] and Gerhart and Yelowitz [34]. The program contains the following loop which is intended to insure that blank characters are skipped and that following the loop the variable LETTER contains a non blank character.

```
WHILE input ≠ emptystring AND FIRST(input) = ' '
  DO input := REST(input);
IF input = emptystring THEN input = READ + ' ';
LETTER = FIRST(input);
```

The WHILE loop terminates either on an empty string or a non blank character. If it terminates on an empty string and the first character in the buffer loaded by the READ instruction is blank, LETTER can contain a blank character.

When this program is translated into FORTRAN and executed on the EXPER system the error is not necessarily caught. The reason for this

failure is not so much a failure of mutation testing as it is of FORTRAN. ALGOL treats strings as a basic type, whereas in FORTRAN they are simulated by arrays of integers. The fact that strings are basic to ALGOL means that if we were constructing a mutation system for ALGOL instead of FORTRAN we would have to consider a different set of mutant operators. A natural operator one would consider can be explained by noting that blanks play a role in string processing programs analogous to that played by zero in numbers. Hence we might hypothesize a blank push operator similar to the zero push operator in EXPER. If we had such an operator an attempt to force the expression FIRST(input) to blank would certainly reveal the error.

The second program, also written in ALGOL, appeared in a paper by Naur [71] and has also been studied widely [30, 34, 37]. The program is intended to read a string of characters consisting of words separated by blanks and/or newline characters, and to output as many words as possible with a blank between every pair of words. There is a fixed limit on the size of each output line, and no word can be broken between two lines.

There are two errors in this program, as studied by Howden. Each time a word is encountered which fits on the current line a blank is inserted to separate it from the preceding word. In the case of the first word in the file this causes an extra blank to be inserted. The second error occurs if the last word in the file is not followed by a blank or newline, in which case the word buffer area is not emptied

and the last word is not output.

Goodenough and Gerhart [37] consider the fact that the program does not suppress multiple blanks between words to be a third error. I have also taken this position.

If we attempt to eliminate mutants of the erroneous program, we find the following three mutants cannot be eliminated without causing the original program to fail:

1. The first FILL := 0 statement can be replaced with FILL := 1
2. FILL never has the value zero in the statement FILL := FILL + 1
3. BUFPOS is always greater or equal to one in the loop FOR k=1,BUFPOS (No data forces the execution of the hidden path in which the loop is never executed)

If the first mutant is to be eliminated its effects must be noticed before the FILL := 0 statement following the writing of the newline character. This mutant can only be eliminated if the first input character is a blank, newline or the start of a word of less than MAXPOS characters. If the first input character is a blank or newline an unnecessary blank will be output, revealing the multiple blanks error. If the first input character is the start of a word of less than MAXPOS characters, an unnecessary space will be output before the word and the initial blank error will be discovered.

FILL can have the value zero in the statement FILL := FILL+1 only in the case we have just output a newline character (which may be the

initial newline). In this case the space is redundant and the initial blank error is revealed.

The only way BUFPOS can equal zero in the FOR loop is in the event of two or more consecutive blank lines or newlines. This would reveal the multiple space error.

Hence both the multiple spaces and the initial blank error will be discovered. If we correct those two errors and perform the reliability experiment again we discover that it is possible to eliminate all mutants using test cases which end in "newline,end of text" or "blank, end of text". These test cases do not reveal the last word error, hence mutation analysis cannot guarantee the discovery of this error. Note, however, that if the test cases are constructed randomly it is extremely unlikely that they would all end in one of these two forms.

The third program appears in a paper by Wirth describing the language PL-360 [88]. It is intended to take a vector of N numbers and sort them into decreasing order. It was also studied by Gerhart and Yelowitz [34]. As the outer loop is incremented over the list of elements the inner loop is designed to find the maximum of the remaining elements, and set register R3 to the index of this maximum. If the position set in the outer loop is indeed the maximum, then R3 will have an incorrect value and the three assignment statements ending the loop will give erroneous results. A listing of this

program is given in section 3.2.9.

There are three mutants which cannot be eliminated without discovering this error. The first two change the statement R0 := A(R1) into R0 := A(R1)-1 and R0 := -ABS(A(R1)) respectively. The third mutant changes the statement A(R1) := R0 into A(R1) := A(R3).

The final program is written in FORTRAN, and computes the total, average, minimum, maximum, and standard deviation for each variable in an observation matrix. The program is adapted from the IBM scientific subroutines package [59]. It was analyzed and three artificial errors inserted in a study by Gould and Drongowski [38]. In Howden's study only one of those errors was discussed. The error occurs in a loop which computes standard deviations. The program has the statement

```
SD(I)=SQRT(ABS((SD(I)-(TOTAL(I)*TOTAL(I))/SCNT)/SCNT - 1
```

A pair of parenthesis have been inadvertently left off the final SCNT - 1 expression. Let X stand for the quantity

```
ABS(SD(I)-(TOTAL(I)*TOTAL(I))/SCNT)
```

The correct standard deviation is SQRT(X/(SCNT-1)). The only way this can be made zero is for X to be zero. But the program containing the error computes the standard deviation as SQRT(1-X/SCNT). If X is zero this quantity is 1, hence the standard deviation is wrong. Alternatively, if the incorrect expression is forced to be zero, the correct standard deviation should be greater than one. Hence by forcing the standard deviation in this line to be zero the error is easily revealed.

Appendix C

Programs analyzed in the third reliability study

The 13 programs studied in section 4.2.3 consisted of 3 of the 4 described in the preceding appendix (the telegraph program, the sorting program, and the statistics program) plus 10 other programs taken from the literature. These last 10 programs will be described here.

The first program appeared in an article by Geller in the Communications of the ACM [33]. A source listing of this program is given in section 3.2.3, where the single error in the program is analysed.

The second program computes the Euclidean greatest common divisor of a vector of integers. It appeared in an article by Bradley in the Communications of the ACM [7]. The program contains the following four errors: (1) If the last input number is the only non-zero number, and it is negative, then the greatest common divisor returned is negative. (2) If the greatest common divisor is not 1, then a loop index is used after the loop has completed normally, which is in error according to the FORTRAN standard. (3,4) There are two DO loops for which it is possible to construct data so that the upper limit is less than the lower limit, which causes the program to produce incorrect results since FORTRAN do loops always execute at least once.

None of the errors are caught using branch analysis. All are caught with mutation analysis.

The next three programs are adapted from the IBM Scientific Subroutines Package [59]. In each program three errors where artificially inserted in a study conducted by Gould and Drongowski [38].

The first program computes the first four moments of a vector of observations. One of the errors would be detected using branch analysis, the other two can be overlooked. All three errors would be discovered using mutation analysis.

The second program computes statistics from an observation table. Again one error would be discovered using branch analysis, but all three errors are discovered with mutation analysis.

The third program computes correlation coefficients. In addition to the three artificial errors inserted by Gould and Drongowski, the program contains a third error which is also present in the original program. This third error involves a variable which is saved and restored so that on returning from the subroutine it should have the same value as on entry. It is possible, however, for the value of this variable to change and not be restored. Two of the artificial errors and the naturally occuring error are detected with branch analysis. All four errors are detected using mutation analysis.

The next program takes three sides of a triangle and decides if it is isosceles, scalene or equilateral. It first appeared in a paper by Brown and Lipow [10]. In [65] a bug is described where two occurrences of the constant 2 are replaced with the variable K. This bug is very subtle, however it can be detected with the test case 6,3,3. Neither branch analysis or mutation analysis would force the discovery of this error.

The seventh program is the FIND program from an article by C. A. R. Hoare [48]. The bug has been studied by the group developing the SELECT symbolic execution system [6]. The bug is very subtle and neither branch testing nor mutation analysis would guarantee its discovery. It would appear that the failure to detect this bug is an artifact of the worst case nature of this analysis, since the error was easily discovered during some early experiments on the coupling effect [22].

The eighth program is the text editor by Naur also described in the last appendix. In this case, however, we used the version studied by Goodenough and Gerhart [37] containing five errors. A listing is given in section 3.2.9.

The ninth and tenth programs are an accounting program and a student scores program from a technical report by S. Sheppard et al [79], issued by the Office of Naval Research. The first program contains three errors and the second a single error. All errors were

detected using mutation analysis. Only two would be caught using branch analysis.

Bibliography

[1] Vinod K. Agarwal and Gerald M. Masson.
Recursive Coverage Projection of Test Sets.
IEEE Transactions on Computers (11):865-870, November, 1979.

[2] Dana Angluin.
On the Complexity of Minimum Inference of Regular Sets.
Information and Control 39(3):337-350, December, 1978.

[3] Douglas Baldwin and Frederick Sayward.
Heuristics for Determining Equivalence of Program Mutations.
Technical Report 161, Yale University, 1979.

[4] A. W. Biermann and R. Krishnaswamy.
Constructing Programs from Example Computations.
IEEE Transactions on Software Engineering SE-2(3):141-153, September, 1976.

[5] Lenore Blum and Manuel Blum.
Toward a Mathematical Theory of Inductive Inference.
Information and Control 28(2):125-155, June, 1975.

[6] R. S. Boyer, B. Elspas and K. N. Levitt.
SELECT - A formal system for testing and debugging programs by symbolic execution.
Sigplan Notices 10(6):234-245, June, 1975.

[7] Gordon H. Bradley.
Algorithm and Bound for the Greatest Common Divisor of n Integers.
Communications of the ACM 13(7):433-436, July, 1970.

[8] Melvin A. Breuer and Arthur D. Friedman.
Diagnosis & Reliable Design of Digital Systems.
Computer Science Press, Woodland Hills, CA, 1976.

[9] Martin Brooks.
Automatic Generation of Test Data for Recursive Programs Having Simple Errors.
PhD thesis, Stanford University, 1980(expected).

[10] J. R. Brown and M. Lipow.
Testing for Software Reliability.
In Proceedings 1975 International Conference on Reliable Software, pages 518-527. IEEE, 1975.
IEEE catalogue number 75 CHO 940-7CSR.

[11] Timothy A. Budd, Richard J. Lipton, Frederick G. Sayward and Richard A. DeMillo.
The Design of a prototype mutation system for program testing.
In Proceedings 1978 National Computer Conference, pages 623-627. AFIPS Press, Montvale, New Jersey, 1978.

[12] Timothy A. Budd and Richard J. Lipton.
Mutation Analysis of Decision Table Programs.
In Proceedings of the 1978 Conference on Information Sciences and Systems, pages 346-349. The Johns Hopkins University, 1978.

[13] Timothy A. Budd and Richard J. Lipton.
Proving LISP Programs using Test Data.
In Digest for the Workshop on Software Testing and Test Documentation, pages 374-403. Fort Lauderdale, Florida, December, 1978.

[14] Timothy A. Budd, Robert Hess and Frederick G. Sayward.
EXPER Implementors Guide.
(In preparation).

[15] Timothy A. Budd, Robert Hess and Frederick G. Sayward.
User's Guide for EXPER: Mutation Analysis system.
(Yale university, memo).

[16] Timothy A. Budd, Richard J. Lipton, Richard A. DeMillo and Frederick G. Sayward.
Mutation Analysis.
Technical Report 155, Yale University, 1979.

[17] Rudolf Carnap.
Logical Foundations of Probability.
University of Chicago Press, 1950.

[18] John C. Cherniavsky.
On Finding Test Data Sets for Loop Free Programs.
Information Processing Letters 8(2):106-107, February, 1979.

[19] Tsun S. Chow.
Testing Software Design Modeled by Finite-State Machines.
IEEE Transactions on Software Engineering SE-4(3):178-187, May, 1978.

[20] Lori A. Clarke.
A System to Generate Test Data and Symbolically Execute Programs.
IEEE Transactions on Software Engineering SE-2(3):215-222, September, 1976.

[21] Lori A. Clarke.
Automated test data selection techniques.
In Proceedings of the Infotech State of the art conference of Software Testing, pages 9/1-9/22. London, England, September, 1978.

[22] Richard A. DeMillo, Richard J. Lipton and Frederick G. Sayward.
Hints on Test Data Selection: Help for the Practicing Programmer.
Computer 11(4):34-43, April, 1978.

[23] Ditto, Hurley, Kessler and Mills.
Safeguard Code Certification Experimental Report.
IBM Systems Assurance Department Report, Federal Systems Division, Gaithersburg, Md., 1970.
Cited in [Hetzel].

[24] Albert Endres.
An Analysis of Errors and Their Causes in System Programs.
IEEE Transactions on Software Engineering SE-1(2):140-149, June, 1975.

[25] Richard E. Fairley.
An Experimental Program-Testing Facility.
IEEE Transactions on Software Engineering SE-1(4):350-357, December, 1975.

[26] W. Feller.
An Introduction to Probability Theory and its Applications.
Wiley, 1957.

[27] R. A. Fisher.
Statistical Methods for Research Workers.
Hafner Publishing Company, New York, 1958.

[28] Ann Fitzsimmons and Tom Love.
A Review and Evaluation of Software Science.
ACM Computer Surveys 10(1):3-18, March, 1978.

[29] Lloyd D. Fosdick and Leon J. Osterweil.
Data Flow Analysis in Software Reliability.
ACM Computer Surveys 8(3):305-330, September, 1976.

[30] Kenneth A. Foster.
Error sensitive test cases.
In Digest for the Workshop on Software Testing and Test Documentation, pages 206-225. Fort Lauderdale, Florida, December, 1978.

[31] Harold N. Gabow, Shachindra N. Maheshwari and Leon J. Osterweil.
On Two Problems in the Generation of Program Test Paths.
IEEE Transactions on Software Engineering SE-2(3):227-231, September, 1976.

[32] Carolyn Gannon.
Error Detection Using Path Testing and Static Analysis.
Computer 12(8):26-32, August, 1979.

[33] M. Geller.
Test Data as an aid in proving program correctness.
Communications of the ACM 21(5):368-375, May, 1978.

[34] Susan L. Gerhart and Lawrence Yelowitz.
Observations of Fallibility in Applications of Modern Programming Methodologies.
IEEE Transactions on Software Engineering SE-2(3):195-207, September, 1976.

[35] Girard and J-C Rault.
A Programming Technique for Software Reliability.
In Symposium on Software Reliability. IEEE, Montvale, New Jersey, 1977.
(Cited in [Glib]).

[36] Tom Glib.
Software Metrics.
Winthrop Publishers, 1977.

[37] John B. Goodenough and S. L. Gerhart.
Towards a Theory of Test Data Selection.
IEEE Transactions on Software Engineering SE-1(2):156-173, June, 1975.

[38] John D. Gould and Paul Drongowski.
An Exploratory Study of Computer Program Debugging.
Human Factors 16(3):258-277, May, 1974.

[39] David Gries.
Compiler Construction for Digital Computers.
Wiley, 1971.

[40] Richard Hamlet.
Testing programs with finite sets of data.
The Computer Journal 20(3):232-237, March, 1977.

[41] Richard Hamlet.
Testing programs with the aid of a compiler.
IEEE Transactions on Software Engineering SE-3(4):279-290, July, 1977.

[42] Richard Hamlet.
Critique of Reliability Theory.
In Digest for the Workshop on Software Testing and Test Documentation, pages 57-69. Fort Lauderdale, Florida, December, 1978.

[43] Sidney L. Hantler and James C. King.
An Introduction to Proving the Correctness of Programs.
ACM Computing Surveys 8(3):331-353, September, 1976.

[44] Steven Hardy.
Synthesis of LISP programs from Examples.
In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, pages 240-245. Tbilisi, Georgia, USSR, 1975.

[45] Carl G. Hempel.
Studies in the logic of Confirmation.
In Aspects of scientific explanation and other essays in the philosophy of science, chapter One,pages 3-51. Free Press, New York, 1965.

[46] P. Henderson and R. Snowden.
An Experiment in Structured Programming.
BIT 12:38-53, 1972.

[47] William C. Hetzel.
An Experimental Analysis of Program Verification Methods.
PhD thesis, University of North Carolina at Chapel Hill, 1976.

[48] C. A. R. Hoare.
Proof of a program: FIND.
Communications of the ACM 14(1):31-45, January, 1971.

[49] Mark A. Holthouse and Mark J. Hatch.
Experience with Automated Testing Analysis.
Computer 12(8):33-36, August, 1979.

[50] John E. Hopcroft and Jeffrey D. Ullman.
Formal Languages and their Relation to Automata.
Addison-Wesley, 1969.

[51] William E. Howden.
Models of Correct Programs and Program Testing.
Technical Report 10, Applied Physics and Information Science Department, University of California, San Diego, 1976.

[52] William E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions on Software Engineering SE-2(3):208-214, September, 1976.

[53] William E. Howden.
An Evaluation of the Effectiveness of Symbolic Testing.
Software: Practice and Experience 8:381-397, 1978.

[54] William E. Howden.
Symbolic Testing and the DISSECT Symbolic Evaluation System.
IEEE Transactions on Software Engineering SE-3(4):266-278, July, 1977.

[55] William E. Howden.
Algebraic Program Testing.
Acta Informatica 10(1):53-66, 1978.

[56] William E. Howden.
Symbolic Testing- Design Techniques, Costs and Effectiveness.
U. S. National Bureau of Standards GCR77-89, SPRINGFIELD, VA, 1977.
National Technical Information Service PB268517.

[57] William E. Howden and Peter Eichhorst.
Proving Properties of Programs from Program Traces.
In Edward F. Miller and William E. Howden, editors, Tutorial: Software Testing & Validation Techniques, pages 46-56. IEEE Computer Society, 1978.

[58] J. C. Huang.
An Approach to Program Testing.
ACM Computing Surveys 7(3):113-128, September, 1975.

[59] International Business Machines.
System/360 Scientific Subroutine Package.
IBM Application Program H20-0205-3, 1966.

[60] B. W. Kernighan and P. J. Plauger.
The Elements of Programming Style.
McGraw Hill, 1974.

[61] S. C. Kleene.
Introduction to Metamathematics.
Van Nostrand, Princeton, N. J., 1964.

[62] P. Kugel.
Induction, Pure and Simple.
Information and Control 35:276-336, December, 1977.

[63] H. Ledgart.
The Case for Structured Programming.
BIT 13:45-57, 1973.

[64] Harry R. Lewis.
A new decidable problem, with applications.
In Proceedings 18th annual symposium on Foundations of Computer Science, pages 62-73. IEEE Computer Society, 1977.

[65] 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. Fort Lauderdale, Florida, December, 1978.

[66] Thomas J. McCabe.
A Complexity Measure.
IEEE Transactions on Software Engineering SE-2(4):308-320, December, 1976.

[67] G. J.Meyers.
Software Reliability: Principles and Practices.
Wiley, 1976.

[68] E. F. Miller and R.A. Melton.
Automated Generation of Testcase Datasets.
In Proceedings 1975 International Conference on Reliable Software, pages 51-58. Los Angeles, CA, 1975.
IEEE catalogue number 75 CHO 940-7CSR.

[69] H. D. Mills.
On the Statistical Validation of Computer Programs.
Technical Report FSC 72-6015, IBM, (undated).

[70] M. Montalbano.
Decision Tables.
Science Research Associates, 1974.

[71] P. Naur.
Programming by Action Clusters.
BIT 9:250-258, 1969.

[72] Daniel L. Ostapko and Se June Hong.
Fault Analysis and Test Generation for Programmable Logic Arrays (PLA's).
IEEE Transactions on Computers C-28(9):617-627, September, 1979.

[73] T. J. Ostrand and E. J. Weyuker.
Remarks on the Theory of Test Data Selection.
In Digest for the Workshop on Software Testing and Test Documentation, pages 1-18. Fort Lauderdale, Florida, December, 1978.

[74] David J. Panzl.
Automatic Software Test Drivers.
Computer 11(4):44-50, April, 1978.

[75] S. L. Pollack, H. T. Hicks and W. J. Harrison.
Decision Tables: Theory and Practice.
Wiley, 1971.

[76] C. V. Ramamoorthy, Siu-Bun F. Ho, W. T. Chen.
On the Automated Generation of Program Test Data.
IEEE Transactions of Software Engineering SE-2(4):293-300, December, 1976.

[77] Hartley Rogers, Jr.
Theory of Recursive Functions and Effective Computability.
McGraw-Hill, 1967.

[78] D. E. Shaw, W. K. Swartout and C. C. Green.
Inferring LISP programs from Examples.
In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, pages 260-267. Tbilisi, Georgia, USSR, 1975.

[79] Sylvia B. Sheppard, Phil Milliman, and Bill Curtis.
Factors Affecting Programmer Performance in a Debugging Task.
Technical Report TR-79-388100-5, Office of Naval Research, February, 1979.

[80] M. L. Shooman and M. I. Bolsky.
Types, Distribution, and Test and Correction Times for Programming Errors.
In Proceedings 1975 International Conference on Reliable Software, pages 347-357. Los Angeles, CA., 1975.
IEEE catalogue number 75 CHO 940-7CSR.

[81] Edward Hance Shortliffe.
Computer-Based Medical Consultations:MYCIN.
American Elsevier, 1976.

[82] Leon G. Stucki.
A prototype automatic program testing tool.
In Proceedings AFIPS Fall Joint Computer Conference, pages 829-836. AFIPS press, 1972.

[83] Leon G. Stucki.
Automatic Generation of Self Metric Software.
In Symposium on Computer Software Reliability, pages 94-100. IEEE, Montvale, New Jersey, April, 1973.

[84] Philip Dale Summers.
Program Construction from Examples.
PhD thesis, Yale University, 1975.

[85] Patrick Suppes.
A Bayesian Approach to the Paradoxes of Confirmation.
In J. Hintikka and P. Suppes, editor, Aspects of Inductive Logic, pages 198-207. North-Holland, 1966.

[86] T. A. Thayer, M. Lipow and E. C. Nelson.
Software Reliability Study.
TRW-SS-76-03 (1976), TRW, One Space Park, Redondo Beach, CA 90278.

[87] L. J. White, E. I. Cohen and B. Chandrasekaran.
A Domain Strategy for Computer Program Testing.
Technical Report OSU-CISRC-TR-78-4, Ohio State University, 1978.

[88] N. Wirth.
PL360, A programming language for the 360 computer.
Journal of the ACM 15(1):37-74, January, 1968.

[89] E. A. Youngs.
Error-Proness in Programming.
PhD thesis, University of North Carolina at Chapel Hill, 1970.