INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
Zeil, Steven Joseph

SELECTING SUFFICIENT SETS OF TEST PATHS FOR PROGRAM TESTING

The Ohio State University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1981
SELECTING SUFFICIENT SETS OF TEST PATHS
FOR PROGRAM TESTING

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of the Ohio State University

By

Steven Joseph Zeil, A.A., B.A., M.S.

* * * * *

The Ohio State University
1981

Reading Committee:  Approved By

Professor Lee J. White
Professor Bruce W. Weide
Professor Stuart H. Zweben

Advisor
Department of Computer
and Information Science
ACKNOWLEDGEMENTS

The author would like to express his gratitude to his adviser and to the other members of the software engineering research group for their support and suggestions. Special thanks also go to the faculty and graduate students in the Department of Computer and Information Science whose influence has been felt throughout the author's course of study. The research described in this dissertation was supported by the U.S. Air Force Office of Scientific Research.
VITA

August 28, 1957 ............... Born - Newport, Ky.


1977 ......................... University Fellowship, The Ohio State University, Columbus, Ohio

1978 ......................... M.S. in Computer Science, The Ohio State University, Columbus, Ohio

1978-1979 ..................... Teaching Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1979-1981 ..................... Research Associate, Department of Computer and Information Science, The Ohio State University

PUBLICATIONS


FIELDS OF STUDY

Major Field: Software Engineering. Professor Lee White

Minor Fields: Numerical Analysis. Professors Howell Mei and Richard Underwood

Information Storage and Retrieval. Professor Anthony Petrarca
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>VITA</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td>I. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>II. A Model of Linearly Domained Programs</td>
<td>12</td>
</tr>
<tr>
<td>III. Predicate Errors in Linearly Domained Programs</td>
<td>23</td>
</tr>
<tr>
<td>IV. Path Selection</td>
<td>48</td>
</tr>
<tr>
<td>V. Errors in Program Computations</td>
<td>82</td>
</tr>
<tr>
<td>VI. Testing for Non-Linear Errors</td>
<td>98</td>
</tr>
<tr>
<td>VII. Summary and Conclusions</td>
<td>119</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>126</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>137</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Domain Error</td>
<td>7</td>
</tr>
<tr>
<td>2. Model Representation of Sample Program</td>
<td>18</td>
</tr>
<tr>
<td>3. Equality Blindness</td>
<td>29</td>
</tr>
<tr>
<td>4. Self Blindness</td>
<td>32</td>
</tr>
<tr>
<td>5. Euclid's Algorithm for GCF</td>
<td>37</td>
</tr>
<tr>
<td>6. Error Terms Involving Undefined Variables</td>
<td>44</td>
</tr>
<tr>
<td>7. Program with $10^{12}$ Paths</td>
<td>50</td>
</tr>
<tr>
<td>8. Euclid's Algorithm for GCF</td>
<td>57</td>
</tr>
<tr>
<td>9. Domain with No Equalities</td>
<td>60</td>
</tr>
<tr>
<td>10. Domain with One Equality</td>
<td>60</td>
</tr>
<tr>
<td>11. Path Selection Algorithm</td>
<td>62</td>
</tr>
<tr>
<td>12. Counting Loop</td>
<td>71</td>
</tr>
<tr>
<td>13. Transforming Nested Loops</td>
<td>75</td>
</tr>
<tr>
<td>14. Euclid's Algorithm for GCF</td>
<td>84</td>
</tr>
<tr>
<td>15. Loan History Program</td>
<td>108</td>
</tr>
</tbody>
</table>
I. Introduction

In the normal course of developing a computer program a major portion of the total effort goes into testing the program for possible errors. Indeed, one might safely assume that any non-trivial program was tested before being released for use. Yet it would be a mistake to assume that all this experience in testing has led to any great show of expertise. On the contrary, software vendors have come to regard the presence of undetected errors in released software as unavoidable and as normal circumstance. One might conclude that either insufficient effort is being put into testing or that much of the testing being done is misguided. Neither claim can be readily confirmed, however, since it is not clear how much testing effort is really necessary nor how testing should ideally be conducted.

As this state of affairs has become apparent, interest in the prospects of automating part of the testing process has grown. Among the earliest attempts to do so were DAVE and similar static analysis systems [FOSDL76,SAIBS79] which analyze the program code, searching for unusual, illegal, or dangerous constructions such as anomalies in the handling of variables, incorrect matching of data types, etc. While such tools are useful, and have proven to be among the easiest to implement in production environments, they do not address the issue of "correctness", the question of whether a program actually computes the proper function.
Efforts at guaranteeing correctness can be placed into two groups. The first of these involves attempts to demonstrate that the correct function is being computed through algebraic proof techniques. There is at yet no consensus on the applicability of these strategies for practical problems, although they can clearly be of great utility when employed for more conservative goals [DERSN81]. The second group consists of those employing testing in its commonly understood form, attempting to demonstrate correctness by running the program with a limited selection of input points. This research addresses itself to the effectiveness of the testing process. We will assume that the programmer or other person responsible for testing is capable of examining a set of input data and a set of output results and determining whether the output is indeed correct for that input.

There are alternatives to requiring this judgment of the tester. We might simply require the programmer to determine whether the output was "reasonable" for the given input data (i.e. not obviously wrong) [WEYUE80]. The ambiguity implied in the term "reasonable" would make it virtually impossible to discuss the reliability of a testing process under such an assumption.

A more plausible alternative would be the requirement of some formal description of the intended function of the program. Testing could then be conducted by comparing the program output to this specification of the program's function. Such an approach has been recommended, but raises many questions regarding the nature of the specification [RICH81]. The degree of confidence to be gained by testing in this manner will depend on the means used to specify the intended program function and upon the confidence that this specification is correct. A specification which contains the same amount of functional information as the program itself would permit testing at least as rigorous as that which is conducted using only the
program, but the drafting of such specifications may be as error-prone as the task of writing the actual program. More general specifications would contain less information about the function to be computed by the program, and so would yield less confidence when used in testing. It is possible that a specification might provide more information than the program itself, such as a guarantee that a correct program would fall within some simple class of easily tested programs. Such a specification would be extremely valuable in extending the reliability of testing. The science of designing languages for specifying the intended functions for programs is in its infancy. Precisely what features are practical or possible in program specifications is unclear. The question of testing with specifications must therefore await further research in this area.

The simple fact is that demonstrating the correctness of a program via testing or any other method is, at best, extremely difficult. The procedural languages employed for most modern programming are extremely powerful, capable of representing any partial function. In attempting to show that a program is correct, we are attempting to demonstrate that an arbitrarily complex function represented by a program is identical to another arbitrarily complex function represented by the specification for that program or perhaps by someone's vague notion of what that program is supposed to do. In general, the problem of proving two functions to be identical is known to be undecidable [HOWD76]; so program testing may legitimately be viewed as an attempt to do the impossible.

It is not likely, however, that this fact will deter the computer science community from testing their programs. On the contrary, there is no sign that testing is in any danger of losing its status as the only universally recognized means of determining program correctness. This does indicate, however, that testing strategies will have to restrict their scope somewhat. The crucial question becomes, "What knowledge about the program is required for the testing problem to
become solvable?" The undecidability result stems from the generality of the two functions. If we can impose some functional restrictions on the program, then many of the issues in testing may become solvable.

Such an approach is not unreasonable. Evidence exists to indicate that programmers do not in practice make use of the full power of modern programming languages, but instead work with combinations of rather simple expressions. The earliest evidence of this comes from Knuth[KNUTD71], whose study demonstrated that addition and subtraction account for the majority of the operators employed in practical programs. Similar results have been obtained by Elshoff [ELSHJ76]. This would seem to indicate a preponderance of simple linear expressions in the program computations. More direct evidence is provided by Cohen [COHEE78], whose study of 50 programs taken from a data processing environment found that only 1 out of 1225 program control statements involved non-linear expressions.

The major components of most programs are computations and predicates. Computations assign new values to program variables, some of which may be sent directly as output while others are employed as intermediate results. Most programming languages include some form of assignment statement specifically for this purpose. Predicates are the decision functions of a program, commonly seen in "IF" statements or as the control statement of a loop. These functions are used to determine which block of computations will be executed next. The different sets of program statements which may be selected via a predicate are often called the branches of the predicate.

Predicate functions often make use of variables whose values have been determined by previous computations. It is therefore necessary to know the results of the preceding computations before a predicate can be evaluated. The process of substituting the results of previous computations for the variables of a predicate is called interpreting the
predicate, and the resulting function is known as the predicate interpretation.

Much of the awesome power of a programming language stems from the use of predicates. A typical program does not apply a single function uniformly over the entire domain of possible input data. Instead it employs predicates to break the set of possible inputs into a large collection of subdomains. Each subdomain corresponds to a unique path through the program. The set of computations encountered along that path make up the program function for that subdomain. In the two-dimensional case (i.e. the program takes two numbers as inputs) this can be viewed as a kind of patchwork quilt, where each patch represents a subdomain. For every patch there will be a unique path through the program. The function computed by the program is identical, for all points within a given patch. The borders of a patch are determined by the interpretations of all the predicates encountered along the corresponding path.

There is no guarantee that a non-empty subdomain will exist for each path through the program. The predicate interpretations which determine the shape of the path subdomain may be mutually contradictory. Such a contradiction means that no data exists satisfying all the conditions for the execution of that path, and consequently the subdomain for that path will be empty. If, for example, we take a path which states that "IF A<B" is true, and later in that path choose a branch for which "IF A>B" is true, then there will be no input point for which that path can be executed. Such a path is called infeasible. Nor will each predicate interpretation along a feasible path affect the shape of the path subdomain. If early in some path we have stated that the condition "IF A>1" holds, then a later predicate "IF A>0" places no additional restriction on the set of points which cause that path to be executed since the truth of this predicate is implied by the truth of the earlier one. The latter predicate would then be termed a redundant predicate.
Redundancy and infeasibility are related phenomena. In the example just given, choosing to say that $A$ is greater than zero results in a redundant predicate. If however we choose to state that $A$ is not greater than 0, the resulting path is infeasible since the earlier predicate guaranteed $A$ to be greater than 1.

Determining feasibility is not easy. If all the constraints imposed by the predicates along the chosen path are linear functions, then a solution is possible using linear programming techniques. For general functions, the problem of determining whether a solution exists to a system of inequalities is known to be undecidable [DAVIS73].

Viewing a program as an elaborate function formed by combining several simpler functions over various subdomains implies a natural classification for program errors. One possibility is that a given data point may fall within the correct subdomain, but an incorrect function is being applied over that subdomain. In terms of program constructs, an input point causes the correct path through the program to be executed; but an incorrect assignment statement causes the wrong value to be calculated. Such errors have been named computation errors [HOWDW76].

Another possible type of error occurs when the borders of a subdomain have shifted, so that some data point now falls into a different subdomain than was intended; and therefore the wrong function is computed. Note that in this case the function may be correct for most of the points in its subdomain. The error lies in the set of points to which that function is applied. In program terms, a wrong path has been followed for some input point, causing a different set of assignments to be utilized and therefore causing an incorrect result to be computed. These errors are called domain errors [HOWDW76]. Figure 1 shows this situation for a program with exactly two input variables, so
that the set of possible inputs to the program forms a flat plane. The bold line represents the correct border, and the dashed line indicates the border actually generated by the (incorrect) program. Input data points which fall within the regions between the two lines are associated with the wrong domain, causing the wrong path to be executed, creating a domain error.

Domain errors can occur due to an error in a program predicate, or due to an error in some computation which affects the interpretation of a later predicate. The term *predicate error* has been employed to describe the first case [WHITL80]. The second case is particularly interesting in that an error in a computation might generate either a domain error or a computation error. In recognition of this ambiguity, we shall refer to domain errors resulting from incorrect computations as *hybrid errors*.

A final possibility is that a function and its subdomain may be missing altogether. This generally occurs when a programmer forgets that certain values are to be treated as separate or special cases. In the program this means that a predicate is missing and hence no
constraint is generated to split some subdomain. These are called \textit{missing path errors} [WHITL80].

This error classification is by no means exhaustive. Clearly there exist other types of errors such as syntax errors, errors in input and output, etc., which do not fall into these classes and do not bear a natural relationship to this functional model of computer programs. What we are attempting to capture with this model are those problems commonly thought of as "run-time" errors, faults which are not normally detectable by static checks such as compiler diagnostics nor by common hardware/software monitoring such as checks for division by zero or range checks. The notion sought here is that of correctness, whether or not the program is computing the correct function rather than whether or not it computes a legal function.

Various strategies for automating the testing process have been advanced, yielding different levels of confidence in the correctness of the program being tested and placing different demands on the programmer (assuming that it is the programmer who is responsible for testing). Most of the proposed methods fall within a class of strategies called \textit{path analysis} testing, where the testing process is conducted in two steps [CLARL76, HOWD75, RAMAC76, WHITL80]:

1. Select a path or set of paths for testing.

2. Perform testing along the chosen paths.

The problem of selecting paths is not well understood. Frequently this step is left unspecified or the assumption is made that the programmer will select the paths [CLARL76, WHITL80]. It is not clear, however, why a programmer can be expected to perform any better at this task than he would do choosing the test points directly, so some sort of automatic
guidance is desirable. Various attempts at formulating such strategies have been made, frequently without analytical justification. These will be discussed in Chapter IV.

The second stage of path analysis testing is the conducting of testing along the chosen paths. Techniques for this stage vary widely. It is not unusual to require the selection of exactly one point for each test path [GOODJ75,HOWDW75]. This has the advantage of simplicity, at the cost of totally abandoning all considerations of reliability. A few researchers have chosen to employ symbolic evaluation, in which the program variables are manipulated as algebraic symbols rather than as receptacles for numeric data [CHEAT79,CLARL76]. An algebraic expression for the function computed along the test path is prepared and displayed, and the programmer must judge the correctness of the expression. While this involves considerably more effort for the programmer, it has the advantage of generality, being reliable for virtually any functions. Others have concentrated on strategies to select small numbers of test points for each path, under the assumption that the program is known to fall within some functional class. Domain Testing describes the selection of test points to detect domain errors in programs where the predicate interpretations are linear functions [WHITL80]. Algebraic Testing deals with the selection of test points for errors in computations known to be restricted to various well-defined functional classes [HOWDW78].

The work which has been done on the second stage of path analysis testing permits us the luxury of assuming that a reliable method of testing a selected path is available to us. This research is therefore concerned with the first stage of path analysis testing, the selection of test paths.

Not all of the automated testing strategies which have been proposed fall under the heading of path analysis testing. One of the
more interesting methods proposed is called *mutation testing*. Mutation testing is a means of measuring the quality of a programmer's test data [DEMIR78, DEMIR79]. It proceeds from the assumptions that:

1. A complex error may be expressed as a combination of a finite number of simpler errors.

2. If a set of test data is good enough to detect all simple errors, it will therefore be good enough to detect all complex errors.

A set of "mutations", simple changes to a program, are chosen. These would include such operations as exchanging one variable name for another, adding one to the iteration limit on some loop, etc. These are applied to the original program, generating a large number of "mutant" programs. The original program and the mutants are executed using the programmer's test data. Any mutant which yields different output from the original program is eliminated, on the basis that the test data was sufficient to detect the error represented by that mutation. Those mutant programs not eliminated are then presented to the programmer who must decide whether

1. The mutant is merely an equivalent, correct version of the original. No test data exists which can tell the difference since both programs compute the same function.

2. The mutant is incorrect, but the test data was insufficient to detect the error. The programmer must now devise additional test cases aimed at the detection of this particular error.

*Mutation testing is conceptually straightforward and applicable to a
variety of program constructs such as arrays which pose great difficulty to path analysis testing. However it is not clear whether these assumptions are truly valid, nor how much confidence can be gained by mutation testing if they are not. These questions will be taken up in Chapter III.

This research presents an analytic investigation of many of the problems inherent in selecting test paths. The goal is to demonstrate that reliable path selection strategies can be formulated for restricted but powerful classes of programs. Such strategies are important not only in their own right as possible practical testing methods, but also as a basis for comparison, as a means of evaluating other proposed strategies.

Chapter II presents a basic model of program behavior and the manner in which predicate errors affect program execution. Chapter III employs that model to discuss path analysis testing for predicate errors in programs where the predicate interpretations are linear functions of the program input data. Knowing that the functional forms of the program predicates form a vector space enables some powerful measures of the effectiveness of a test path to be derived. These measures are used to discuss path selection strategies in Chapter IV. Chapter V then expands the set of errors which can be treated by these methods to include errors in the program computations. Chapter VI indicates how these results may be extended to programs involving higher order functions than the linear expressions discussed earlier.
II. A Model of Linearly Domained Programs

In this chapter we begin an investigation of path analysis testing strategies. A model will be presented which describes a class of programs whose domain borders are straight line segments. This model will be employed in following chapters to discuss the effectiveness of strategies for selecting test paths.

Since we are concentrating on the question of path selection it will be necessary to assume the availability of a reliable method of conducting testing along a given path.

A strategy for testing a program construct for some given path will be considered **reliable** if, whenever that construct behaves incorrectly along that path, the strategy guarantees that incorrect output will be observed by the tester. For program predicates this means that whenever a predicate is incorrect along some path, the test strategy detects the fact that for certain data, the program will execute an incorrect path. The key here is the stipulation that the construct behaves incorrectly "along that path." An incorrect statement in a program may happen to behave correctly for all data in the domain of a certain path, in which case no testing strategy, no matter how reliable, could detect the error. For example, suppose that a program erroneously adds the value of X to some calculation. Certain paths may exist on which the value of
X is guaranteed to be zero. If these paths were used for testing, the erroneous computation could not possibly be detected. A sufficient path selection strategy would have to guarantee that such paths were not the only ones being tested.

The control flow statements in a computer program partition the input space into a set of mutually exclusive domains, each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. Since a predicate error alters the domain over which paths are executed, two distinct possibilities exist. If the error increases the domain of the path under test, then data will exist for which the correct computations lie along some other path, but the program follows the test path. Alternatively if the error decreases the domain of the test path, points will exist just outside the path domain for which the correct action would be to take the test path, but the actual path chosen will be some other path [COHEE78]. Of course, errors may in general cause some combination of these two cases, but a reliable testing strategy should detect both situations. (It is important to note the implication that test points for a particular path need not be restricted to the domain of that path.)

There are limitations inherent in any testing strategy. One such limitation has been previously identified as a "missing path" error, in which a required predicate does not appear in the given program to be tested [HOWDW76]. Especially if this predicate were an equality, no testing strategy based on the program text could systematically determine that such a predicate should be present. The analysis which follows is therefore subject to the following assumptions:

1. Missing path errors do not occur;
2. The input space is continuous;
3. Predicates are simple, not combined with AND.
OR, or other conjunctions;
(4) adjacent domains compute different functions.

Assumption (2) simplifies the analysis of testing methods using standard mathematical tools. In practice this should cause little difficulty as long as the size of the domain is not comparable to the discrete resolution of the space [WHITL78]. If assumptions (3) and (4) are imposed, the testing strategy is considerably simplified, as no more than one domain need be examined at one time in order to select test points. Assumption (3) need not actually restrict the set of acceptable programs, since any program can be easily transformed to eliminate compound predicates. More motivation for assumptions (1) and (4) will be presented in Chapter III.

Even given the above assumptions, most programming languages will be powerful enough to represent any computable function, and so the problem of determining the correctness of program predicates will remain unsolvable. Some knowledge of the class of permissible functional forms for the predicates is required. The final assumption is:

(5) predicate interpretations are linear functions of the input values.

Any program satisfying the five constraints given above will henceforth be referred to as a linearly domain program. Within these assumptions it is possible to describe reliable strategies for selecting test points. For example, the domain testing strategy chooses a small set of test points which are reliable for arbitrarily small errors [COHEE78, WHITL80]. Another possibility is the use of symbolic execution to generate algebraic descriptions of the path domain and function, effectively sampling over all points in the path domain.
Although assumption (5) appears to be severely limiting, some evidence exists to indicate that it may hold for a surprisingly large class of programs. The studies by Knuth, Elshoff, and Cohen have already been discussed in Chapter I. Chapter VI will consider the results of relaxing this assumption to permit non-linear functions.

It is the goal of this research to provide a mathematical justification for some of the intuitive arguments presented in the preceding pages. Towards this end we now present a model for the behavior of linearly domained programs. In this model the program itself is represented as a static set of transformations and predicates, while the execution state is represented using the dynamic attributes of environment, path, and constraints.

The central element in this model is the environment. Properly speaking, the environment of a program represents the values of all variables at any point in the program's execution. However since the subject of this analysis is the detection of domain errors, we shall restrict our representation of the environment to those variables and other factors which may affect the flow of control. Then the environment may be represented as the following vector:

$$\bar{v} = (1, x_1, \ldots, x_m, y_1, \ldots, y_n)^T$$

The y_i represent those program variables which may directly or indirectly affect the program control flow through their effects on the evaluation of program predicate expressions. The x_i represent the values of input data. It is convenient for purposes of illustration to treat these as special variables whose values are established prior to execution and held fixed thereafter, although in practice no such special variables need exist. The first element of the environment vector is held to the constant "1" as a notational convenience so that computations involving constants as well as variables might be expressed
in a uniform manner. Initially, only this constant term and the \( x_i \) terms are considered to be defined. The program must define the program variables as functions of these terms.

The components of the program itself can then be described in terms of their interactions with the environment vector. A program is considered as a set of pairs of the form \((C_i, T_i)\) where \( C_i \) is a computation or transformation to be applied to the current environment to generate the new environment and \( T_i \) is a predicate which is applied to the new environment. These predicates may be either equalities or inequalities. The next \((C_j, T_j)\) pair to be used is determined by the result of the application of \( T_i \).

The process of executing a program consists of determining a path \( P = (p_0, p_1, \ldots, p_k) \) where the \( p_j \) are the indices of the \((C_i, T_i)\) pairs which are to be successively applied to the environment. As a convention, we shall let \( p_0 = 0 \) designate the start of the program.

The term subpath will be used to designate a path which does not begin with \( p_0 = 0 \) or does not end at a valid HALT statement, that is, a path which does not describe a complete execution of the program. An initial subpath shall be defined as a subpath beginning at the start of the program, for which \( p_0 = 0 \).

For linearly domained programs, after any step along such a path, the old and new environments will be linearly related. The computations \( C_i \) may therefore be treated as linear transformations. Taking \( C_i \) as a matrix, the \( k^{th} \) step along a path \( P \) causes the environment to undergo the transformation:

\[
\bar{v}_{k+1} = C_k \bar{v}_k
\]
The environment after $k$ steps along path $P$ is therefore given by:

$$
\bar{v}_{k+1} = C_{p_k} \cdots C_{p_1} C_0 \bar{v}_0
$$

where $\bar{v}_0$ is the initial environment. It will often be convenient to represent this long string of matrices as a single matrix

$$
C_A = C_{p_k} \cdots C_{p_1} C_0
$$

representing the total transformation along subpath $P_A$.

Since by assumption (5) the predicate interpretations in a linearly dominated program must be linear functions, and assumptions (2) and (3) insure that the predicate functions are continuous, the predicates themselves must be linear expressions. The predicates may then be treated as vectors, $\bar{t}_i$, such that the scalar product $\bar{t}_i \cdot \bar{v}_{k+1}$ is compared with zero to determine the next index $p_{k+1}$. (The mechanism by which the next index is selected has deliberately been left unspecified as it is not of importance to this analysis.)

Figure 2 shows a short program segment and its representation under this model. If we treat the variables $A$ and $B$ as restricted input variables in the sense described earlier, then the environment vector has six components $(1, x_1, x_2, y_1, y_2, y_3)^T$ corresponding to $(1, A, B, S, T, U)^T$. Let the values in the input stream for $A$ and $B$ be designated as "a" and "b". Then the initial environment $\bar{v}_0$ is $(1, a, b, ?, ?, ?)^T$ where "?" indicates an undefined value.

Two initial subpaths are available up to location PRED depending on the result of the test for $A > 2$, $P_A = (0, 1, 3)$ and $P_B = (0, 2, 3)$. After the first step along either path, the new environment would be

$$
\bar{v}_1 = C_0 \bar{v}_0 = (1, a, b, ?, 1, a)^T
$$
Then applying the predicate $T_0$ involves comparing the values

$$T_0 \cdot \vec{v}_1 = (-2, 1, 0, 0, 0, 0)(1, a, b, ?, 1, a)^T_0 = -2 + a$$

to zero. Note that the values in $\vec{v}_1$ for $S$, $T$, and $U$ and the expression for $T_0 \cdot \vec{v}_1$ do indeed correspond to the results expected of the program at this point.

Completing the execution along both subpaths, define $C_A$ and $C_B$:

<table>
<thead>
<tr>
<th>Program</th>
<th>i</th>
<th>$C_i$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ A, B;</td>
<td>0</td>
<td>1 0 0 0 0 0</td>
<td>-2</td>
</tr>
<tr>
<td>$T = 1$;</td>
<td></td>
<td>0 1 0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>$U = A$;</td>
<td></td>
<td>0 0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>IF $A &gt; 2$ THEN</td>
<td></td>
<td>0 0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>$T = 2 \times U$;</td>
<td>1</td>
<td>1 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>ELSE</td>
<td>2</td>
<td>1 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>$T = 2<em>A + 2</em>B$;</td>
<td></td>
<td>0 1 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>$U = U + B$;</td>
<td></td>
<td>0 0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>END IF</td>
<td></td>
<td>0 0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 2 2 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>$S = 1$;</td>
<td>3</td>
<td>1 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>PRED: IF $U=B$ THEN ...</td>
<td></td>
<td>0 1 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0 0 0</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2: Model Representation of Sample Program
Taking the products of the transformation matrices along some path is equivalent to symbolically executing that path. The total transformation matrix represents the equivalent assignments along that path. But not all such paths are valid. An initial subpath \( P = (0, p_1, \ldots, p_k) \) shall be called a testable subpath if there exists a subpath \( P' = (p_k, \ldots, p_h) \) such that

1. \( P' \) ends with a HALT statement;

2. There exists some input value causing the path \( P'' = (0, p_1, \ldots, p_k, p_{k+1}, \ldots, p_h) \) to be executed;

3. The predicate \( \overline{\tau_{p_k}} \) is not implied by the conjunction of other predicates on \( P'' \).

These three conditions mean that an initial subpath is testable if it can be completed without making the final predicate infeasible or redundant. This requirement is important since it guarantees that a shift in the interpretation of the final predicate in the test path will not be masked by the other predicates in the completed test path.

\[
C_A = C_3C_1C_0 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix} \quad (A > 2)
\]

\[
C_B = C_3C_2C_0 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix} \quad (A \leq 2)
\]

so that \( CA\bar{V}_0 = (1, a, b, 1, 2a, a)^T \)

\( CB\bar{V}_0 = (1, a, b, 1, 2a + 2b, a + b)^T \).
Just as the expression $C_A \bar{v}_0$ denotes the symbolic execution along path $P_A$, with each element of the environment replaced with its new value after execution, the expression $C_A^T$ carries an important meaning related to program execution. As noted earlier, $T^T C_A \bar{v}_0$ denotes the application of the predicate $T$ to the transformed environment created by executing along $P_A$. This interpretation is consistent with the grouping $T^T(C_A \bar{v}_0)$. If instead we think in terms of $(T^T C_A) \bar{v}_0$, this expression represents the interpretation of the predicate $T$ applied to the input space. The vector $C_A^T$ is the interpretation of $T$ along path $P_A$, that is, the linear function obtained by substituting the appropriate constant and input expressions for each program variable in the predicate $T$. $C_A^T$ is a vector with non-zero terms in the constant and input positions only, representing a linear expression in terms of only constants and inputs.

Simply put, multiplication by $C_A$ represents execution along $P_A$ while multiplication by $C_A^T$ represents a back-substitution appropriate to $P_A$. That both interpretations should arise from the same expression should not be considered unusual. Both views are consistent, generating the same expressions. They merely arise from two different perspectives, one moving forward through a program and the other tracing backwards along the same path.

For the program in figure 2, a back substitution along path $A$ for the final predicate would be given by:

$$C_A^T = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

Thus the interpretation of $T_A^3$ along path $A$ is $a-b$, identical to what was obtained using forward execution.
One final item remains to be modeled. Every predicate encountered along a path places restrictions on the legal set of input values for that path. However the constraints imposed by equality predicates are qualitatively different from those imposed by inequalities, since a valid equality predicate reduces the dimension of the space of legal inputs $x_1$. Equality restrictions can arise from two sources. The more obvious source are equality predicates such as "IF $A = B$". However equality restrictions can also arise from combinations of two or more predicates. An example of this would be the pair "IF $A \geq B$" and "IF $A \leq B$''. If both of these predicates are taken to be true, then the condition "$A = B$" is implied to be true. Equality conditions arising from such combinations will be referred to as coincidental equalities. The detection of coincidental equalities will be discussed in Chapter IV.

As we move along a path, the set of equality restrictions can be collected. Since these are generated by predicates, these restrictions will be of the same functional form as the program predicates. Suppose we have such a restriction imposed by the final predicate of $P_A$. Denote this restriction as $q$, with $q \cdot \vec{v}_A = 0$ for all environments $\vec{v}_A$ satisfying the equality. This expression has the drawback of being dependent on the current transformation of the environment. If we continue down the path, $\vec{v}_A$ is lost and $q$ can no longer be interpreted. The problem is again one of perspective. $q \cdot \vec{v}_A$ is equal to $q^T C_A \vec{v}_0$, so if we take the interpretation of $q$ along the path, defining $\vec{r} = C_A^T q$, then we have $\vec{r} \cdot \vec{v}_0 = 0$. The vector $\vec{r}$ is in terms of the inputs and constants only, so it represents the effect of $q$ on the set of points comprising the path domain. This is precisely the notion we were trying to capture.

For example, the final predicate in figure 2 imposes the equality condition "$U = B$" when true. As more of the program is executed, it is possible that the value of $U$ would be changed. So in order to keep
track of the restriction imposed on the path domain, the interpretation of 
"U - B = 0" is taken. This was shown above to be "A - B = 0" for path A. Since A and B are inputs, they cannot be changed and so this 
representation of the equality restriction is unaffected by any future 
transformations of the environment.

In recognition of this, there is associated with each testable 
subpath \( P_A \) a set of restriction vectors \( \mathcal{R}_{j}^A \), for \( 0 < j < k_A \), such that if \( \mathcal{V}_0 \) is an initial environment which might cause path \( P_A \) to be executed, then

\[
\mathcal{R}_{j}^A \cdot \mathcal{V}_0 = 0, \quad 0 < j < k_A
\]

To summarize, this model represents linearly domained programs in 
terms of computation-predicate pairs \( (C_i, \mathcal{T}_i) \) with execution being 
described in terms of environment \( \mathcal{V} \), paths \( P \), and equality restrictions \( \mathcal{R}_j \). In the next chapter this model will be employed to investigate the 
detection of errors in program predicates.
III. Predicate Errors in Linearly Domained Programs

In this chapter, the basic model of linearly domained programs will be used to investigate errors in the program predicates. This relatively simple case will serve to define the basic concepts to be used in the more general problems discussed in later chapters.

The pivotal concept of this investigation is the definition of the set of predicate interpretations as a finite-dimensioned vector space, in this case, the space of linear expressions. A vector space provides a powerful way of describing and manipulating an infinitely large set of functions. Two properties of vector spaces are particularly significant here. First, a vector space can be completely described using a finite set of "characteristic vectors", such that every member of the vector space can be expressed as a linear combination of those characteristic vectors. Second, any linear combination of two or more vectors from some vector space will also be in that space. Therefore, given two predicates $T$ and $T'$, where $T$ is a "correct" predicate and $T'$ an "incorrect" one, the error term $(T' - T)$ is also a vector. Hence the set of possible errors is also a vector space and can be described using a finite set of "characteristic" errors.

Whenever a path has been chosen to test a given predicate, it is possible for certain predicate errors to escape detection no matter how
reliable a testing method is used. The goal of this chapter is to
demonstrate that the set of undetected predicate errors is a
well-defined subspace of the set of all possible predicate errors.

To characterize these undetected errors, consider a program where
for some pair \((C_i, T_i)\) the predicate is replaced by an erroneous
predicate \(T'_i\) such that
\[
T'_i = T_i + \alpha \tilde{e}, \quad \alpha \neq 0.
\]

\(\tilde{e}\) is a unit vector giving the "direction" of the error and \(\alpha\) is a
scalar giving the magnitude of the error. Let \(P_A\) be a testable subpath
ending with \((C_i, T'_i)\). The environment after executing along \(P_A\) will be
\(\bar{v}_A = C_A \bar{v}_0\) where \(C_A\) is the total transformation along \(P_A\). The assumptions
made in Chapter II that adjacent domains compute different functions and
that no missing path errors have occurred imply that this error is
detectable exactly when it results in a shift in one of the domain
boundaries. A reliable strategy for selecting test data will therefore
be able to detect the erroneous predicate if and only if there exists no
nonzero real number \(h\) such that for all \(\bar{v}_A\)
\[
T'_i \cdot \bar{v}_A = h T_i \cdot \bar{v}_A.
\]

Conceptually, there are two possibilities here. The case \(h=1\) occurs
when the error term evaluates to zero along the test path, leaving the
predicate interpretation unchanged. Clearly no testing strategy can
detect this case. The case \(h \neq 1\) arises when the interpretation of the
erroneous predicate is a multiple of the correct one. The predicates
"IF \(X-1 > 0\)" and "IF \(2X-2 > 0\)" represent this case. Clearly they
cannot be distinguished by any choice of test points. If neither of
these two possibilities hold, then an actual shift in the direction of
the border generated by the predicate under test must have occurred
(figure 1). Such a shift guarantees the existence of data points with
which the domain error can be detected.

Since the correct form of the predicate is unknown, substitute for $T_i$:

$$T_i \cdot \vec{v}_A - \alpha \vec{e} \cdot \vec{v}_A = hT_i \cdot \vec{v}_A$$

$$(h-1)T_i \cdot \vec{v}_A + \alpha \vec{e} \cdot \vec{v}_A = 0$$

$$[(h-1)T_i + \alpha \vec{e}]^T \omega A \vec{v}_0 = 0 \quad (1)$$

Consider first the solution set for $h=1$. Then if $\vec{e}^T \omega A \vec{v}_0 = 0$ for all $\vec{v}_0$ in the domain of the path $P_A$, then the error $\vec{e}$ will go undetected. Consider the various cases which may force this expression to zero:

1. $\omega A \vec{v}_0 = 0$

2. $\vec{e}^T \omega A = 0$

3. $\vec{e}^T \omega A \neq 0$, $\vec{e}^T \omega A \vec{v}_0 = 0$ for all $\vec{v}_0$ in the domain of $P_A$.

The first case is clearly impossible since $\vec{v}_A = \omega A \vec{v}_0$ will always have a constant "1" in its first position. as constants may not be reassigned new values.

If $\vec{e}^T \omega A = 0$ then transposing to get $\omega A \vec{e} = 0$ indicates that this is an eigenvalue problem $\omega A \vec{e} = \lambda \vec{e}$ with $\lambda=0$. The solution to this problem can be found by examining the structure of the individual $G_i$. 
Each matrix $C_i$ may be partitioned into the form

$$C_i = \begin{bmatrix} Q & R \\ S & T \end{bmatrix}$$

where $Q$ is $(m+1)$ by $(m+1)$ and $T$ is $n$ by $n$.

The matrix $Q$ maps the inputs and constants from the old environment into the new environment, and so must be the identity matrix $I$. $R$ maps the variables of the old environment onto the inputs and constants in the new environment. Such assignments are forbidden and so $R$ must be entirely zero. $S$, mapping the old inputs and constants into the new variables, may contain any real values. $T$ maps the old variables into the new variables. This mapping is unrestricted for all $C_i$ except $C_0$, the initial assignments where all variables are initialized in terms of inputs and constants. For $C_0$ the component $T$ must be entirely zero, so that

$$C_0 = \begin{bmatrix} I & 0 \\ S_0 & 0 \end{bmatrix}$$

$$C_i = \begin{bmatrix} I & 0 \\ S_i & T_i \end{bmatrix}$$

and for any initial subpath $P_A$,

$$C_A = C_P C_{P_{k-1}} \cdots C_{P_1} C_0 = \begin{bmatrix} I & 0 \\ S_A & 0 \end{bmatrix}$$
Now the solution to the eigenvalue equation $c_A^T\vec{z} = \lambda \vec{z}$ is given by inspection. If $\vec{u}_i$ are the vectors forming the columns of the identity matrix of the same dimension as $c_A$, then

\[
c_A^T\vec{u}_i = \vec{c}_i, \quad i = 1 \ldots (m+1) \tag{2}
\]

where $\vec{c}_i$ is the $i$th row of $c_A$. But the $\vec{c}_i$ for $m+2 \leq i \leq m+n+1$ are linear combinations of the $\vec{u}_i$ for $1 \leq i \leq m+1$. Therefore

\[
c_A^T\vec{u}_i = \vec{c}_i, \quad 1 \leq i \leq m+1, \quad \lambda = 1
\]

\[
c_A^T(\vec{c}_i - \vec{u}_i) = 0, \quad m+2 \leq i \leq m+n+1, \quad \lambda = 0
\]

describes the eigenvalues and eigenvectors of $c_A^T$.

Thus an error $\vec{e}$ will go undetected if

\[
\vec{e} \in \text{null-space}(c_A^T) \tag{4}
\]

where $\text{null-space}(c_A^T) = \text{span}(\vec{c}_i - \vec{u}_i)$, $m+2 \leq i \leq m+n+1$.

This vector space has a simple interpretation in terms of symbolic evaluation. In figure 2 consider the subpath leading to PRED for the case $A > 2$. Since $T = 2a$ along this path, one error satisfying the above criterion is $\vec{e} = (0, 2, 0, 0, -1, 0)^T$. Adding this to the vector representation of $T_3$ would give a vector equivalent to the predicate "IF $U + 2A - T > B". For any data which causes this path to execute, this
erroneous form will be indistinguishable from "IF U>B" since the added term "2*A-T" will evaluate to zero. This behavior is termed "assignment blindness", because it results solely from the assignment statements encountered along the test path.

Finally if $\mathbf{e}^T \mathbf{C}_A \neq \mathbf{0}^T$ but $\mathbf{e}^T \mathbf{C}_A \mathbf{v}_0 = 0$ for all $\mathbf{v}_0$ in the domain of $\mathbf{P}_A$, then the error still goes undetected but assignment blindness cannot be a factor since $\mathbf{e}^T \mathbf{C}_A \neq \mathbf{0}^T$.

Let $\mathbf{u} = \mathbf{C}_A^T \mathbf{e}$. Then $\mathbf{u}^T \mathbf{v}_0 = 0$, but neither $\mathbf{u}$ nor $\mathbf{v}_0$ can be a zero vector. If the set of legal $\mathbf{v}_0$ for the path $\mathbf{P}_A$ forms a space of dimension $m$, there will always exist some $\mathbf{v}_0$ in that domain such that $\mathbf{u}^T \mathbf{v}_0 \neq 0$. However, equalities might restrict the $\mathbf{v}_0$ to a hyperplane of that space of inputs. If $\mathbf{u}$ is orthogonal to that hyperplane, then $\mathbf{u}^T \mathbf{v}_0 = 0$ for all legal $\mathbf{v}_0$. This implies that some form of "equality blindness" exists to complement assignment blindness. Since $\mathbf{P}_j^A \cdot \mathbf{v}_0 = 0$, for $j=1..k_A$ represent these equality restrictions, it is apparent that the necessary condition for the inability to detect an error due to equality blindness is

$$\mathbf{C}_A^T \mathbf{e} \in \text{span}[\mathbf{P}_j^A]$$

(5)

An example of equality blindness is given in figure 3, where the two expressions for the second predicate will be indistinguishable for any test path where the first predicate is true.

The above analysis of the conditions under which $\mathbf{e}^T \mathbf{C}_A \mathbf{v}_0$ would go to zero has identified a number of isolated vectors representing undetectable errors for the path being tested. Clearly any linear combination of these errors will also go undetected, implying that the total undetected space is described by the span of these vectors. The total undetected space for the case where $h=1$ would then be given by the
span of the null-space of $C_A^T$ and of the set \( \{ \tilde{e} : C_A^T \tilde{e} \in \text{span}(F^A) \} \).

Equation (4) describes assignment blindness directly in terms of the assignments performed along a path. In contrast, the description of equality blindness requires the solution of (5). Such a procedure would be, at best, awkward. This indirect method of specifying the characteristic vectors for equality blindness can be simplified. When the spaces for assignment and equality blindness are combined in this way, there may be some overlap. Choose any \( \tilde{e} \) which is subject to equality blindness. Decomposing \( \tilde{e} \) into its components,

\[
\tilde{e} = \sum_{i=1}^{m+n+1} \beta_i \bar{\mu}_i
\]

Since we have specified that this \( \tilde{e} \) is subject to equality blindness,

\[
C_A^T \tilde{e} \in \text{span}(F^A);
\]

Correct Code  

\[
\text{IF } D=1 \text{ THEN }
\]

\[
\text{IF } C+D > 1 \text{ THEN }
\]

Incorrect Code  

\[
\text{IF } D=1 \text{ THEN }
\]

\[
\text{IF } C > 0 \text{ THEN }
\]

Figure 3: Equality Blindness
and there must exist a set \( \{ \gamma_i \} \) such that

\[
\mathbf{C}^T_A \mathbf{e} = \sum_j \gamma_j \mathbf{f}_j^A.
\]

Referring back to the expressions for \( \mathbf{C}^T_A \mathbf{e} \) given in (2) and (3):

\[
\begin{align*}
\mathbf{C}^T_A \mathbf{e} &= \sum_{j=1}^{m+1} \beta_j \mathbf{u}_j + \sum_{j=m+2}^{m+n+1} \beta_j \mathbf{c}_j \\
\mathbf{C}^T_A \mathbf{e} &= \sum_j \gamma_j \mathbf{f}_j^A
\end{align*}
\]

\[
\begin{align*}
\sum_{j=1}^{m+1} \gamma_j \mathbf{f}_j^A &= \sum_j \gamma_j \mathbf{f}_j^A - \sum_{j=m+2}^{m+n+1} \beta_j \mathbf{c}_j \\
\sum_{j=1}^{m+1} \beta_j \mathbf{u}_j + \sum_{j=m+2}^{m+n+1} \beta_j \mathbf{u}_j &= \sum_j \gamma_j \mathbf{f}_j^A - \sum_{j=m+2}^{m+n+1} \beta_j (\mathbf{c}_j - \mathbf{u}_j)
\end{align*}
\]

The expression on the left is the decomposition of \( \mathbf{e} \) as given in equation (6), and so

\[
\mathbf{e} = \sum_j \gamma_j \mathbf{f}_j^A - \sum_{j=m+2}^{m+n+1} \beta_j (\mathbf{c}_j - \mathbf{u}_j)
\]

The first sum on the right is an arbitrary combination of the equality restriction vectors. The second summation is over those vectors which have been previously derived as forming the null space of \( \mathbf{C}^T_A \); so the solution set to equation (1) for the case \( h=1 \) is given by

\[
\mathbf{e} \in \text{span}[\text{null-space}(\mathbf{C}^T_A), \{ \mathbf{f}_j^A \}].
\]

Now consider equation (1) when \( h \neq 1 \). The quantity in square brackets is simply a vector, and we have just solved the equation

\[
\alpha \mathbf{e}^T \mathbf{C}_A \mathbf{u}_0 = 0
\]

for all \( \alpha \). Substituting the quantity in brackets for \( \alpha \mathbf{e} \) gives
\[(h-1)\mathbf{T}_1 + \alpha\mathbf{e} \in \text{span}[\text{null-space}(\mathbf{c}_A), \{\mathbf{r}_j\}].\]

The expression on the left is a sum of two vectors. Since \((h-1)\) and \(\alpha\) can have any ratio, this is an arbitrary combination of a known and an unknown vector. Solving for the unknown gives the following solution:

\[\mathbf{e} \in \text{span}[\text{null-space}(\mathbf{c}_A), \{\mathbf{r}_j\}, \mathbf{T}_1].\]

Although this is the solution for \(h \neq 1\), the \(h = 1\) solution set is a subset of this one, so this equation represents the total solution set for equation (1).

The final term in this vector space may seem unnecessary. The \(\mathbf{T}_1\) term indicates that adding a predicate to itself is not detectable. This is not particularly surprising. If a correct form of some predicate were "IF X-1 > 0", then substituting the predicate "IF 2*X-2 >0" will never cause a domain error. Such error expressions are always undetectable. In fact, it is debatable whether they can really be termed errors.

The reason the \(\mathbf{T}_1\) term cannot be ignored is that the vector \(\mathbf{T}_1\) can combine with assignment and equality blindness vectors to provide entirely new undetectable errors. This is illustrated in figure 4. The error in this example cannot be detected, but this undetectability does not arise from assignment nor equality blindness. Such errors cannot be detected because their interpretations are parallel to the predicate being tested. This behavior will be termed **self blindness**.

The derivation of assignment, equality, and self blindness is summarized in the following theorem:
Theorem 1. Characterization of Undetected Predicate Errors

Let \( P_A \) be a testable subpath in a linearly domained program. Then an error \( \tilde{e} \) in the final predicate \( T' \) of \( P_A \) will be undetectable if and only if

\[
\tilde{e} \in \text{span}[\text{null-space}(C_A^T), \{r_j^A\}, T'].
\]

Although the examples of assignment and equality blindness given above may seem trivial or may appear to involve awkward or unlikely errors, it is important to note that any linear combination of these errors will also go undetected. Such combinations can involve simple expressions, yet may not be apparent from an inspection of the program.

<table>
<thead>
<tr>
<th>Correct Code</th>
<th>Incorrect Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X := A )</td>
<td>( X := A )</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>IF ( X &gt; 1 ) THEN</td>
<td>IF ( X + A &gt; 2 ) THEN</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Figure 4: Self Blindness
It is interesting to compare the implications of this theorem with the assumptions underlying the mutation testing strategy described in Chapter I. Since Theorem 1 can be considered to be describing a set of "mutations" which may be applied to the program predicates without being detected, some sort of close relationship appears likely. The similarity is strengthened by the fact that typical mutations, such as substitution of one variable for another in an expression, are equivalent to addition of linear expressions. However, mutation testing assumes that test data which detects a set of simple errors will detect any combination of those errors. Theorem 1 states that the inverse is actually true - test paths which fail to detect simple errors will also fail to detect combinations of those errors. These two statements are not compatible. Consider a program with three variables, X, Y, and Z, and a predicate "IF X + Y - Z > 1". One common set of mutations involves replacing "+" by "-" or "-" by "+" in summations [DEMIR79]. Mutation testing using this set of mutations would guarantee that predicates such as "IF X + Y + Z > 1" and "IF X - Y - Z > 1" could not be correct. In the language of Theorem 1, this means that the test data was not blind to the expressions "Y" and "Z". The test data must have included cases where Y and Z were not zero. Now suppose that the correct predicate were "IF X - Y + Z > 1". The error term would then be "2Y - 2Z". It is entirely possible that the chosen test paths would be blind to this expression. The guarantee that non-zero values of the variables were tested does not imply that test cases were employed for which Y and Z were not equal. Yet the expression "Z - Y" represents a simple combination of the mutation expressions "Y" and "Z", a combination obtained by switching both operators in the original predicate rather than only one of the two.

It is entirely possible that another set of mutations might be defined which would detect this particular error. In fact, substitution of one variable for another is a common mutation which would require testing of cases where Y and Z had different values; For any given error
expression, there will exist classes of mutations which would detect that error. The simplest example of such classes is the class of mutations consisting of the addition of the given error term. However, for any finite set of linear mutations, there will exist linear combinations of that set whose detection cannot be guaranteed. This is not due to the linearity restriction. In general, mutation testing tests only isolated hyperplanes of the total space of possible errors. An infinite number of possible error expressions will escape mutation testing because it assumes that detected errors combine to increase confidence rather than the proper assumption that undetected errors combine to decrease confidence in a set of test data.

Geometrically, for a two variable program we would like to eliminate the entire plane of possible error expressions. If we were to miss any two linearly independent errors, then the entire plane would be missed. Mutation testing effectively tests only isolated lines (single expressions) in that plane. An infinite number of other lines exist, representing error expressions which may have gone undetected.

This does not mean that mutation testing is not a useful testing strategy. However mutation testing is not a reliable strategy in the sense of eliminating all possible error expressions. As more complete sets of mutations are defined, the error terms missed by mutation testing may become increasingly complicated, and possibly become increasingly improbable. The utility of this method must therefore depend on whether a set of mutations can be found which captures the more common errors made by programmers, rather than depending on a belief in the ability of this strategy to automatically account for more complex errors.

The existence of a characterization theorem suggests a return to the question posed earlier, "After testing several paths, what is the marginal advantage of testing still another path?" Since the undetected
errors are described by a well-defined vector space, a new proposed path will form a useful test only if some portion of the (previously) untested space is detectable along the new path.

Theorem 2. Path Rejection Criteria

If a set $K=\{P_k\}$ of testable subpaths ending at $T^*_i$ has been previously tested, then a proposed testable subpath $P_A$ also ending with $T^*_i$ need not be tested if and only if

$$\cap_{k \in K} \text{span}[\text{null-space}(C^T_k), \{\bar{r}_j\}] \subseteq \text{span}[\text{null-space}(C^T_A), \{\bar{r}_j\}].$$

The $T^*_i$ terms disappear in this theorem because that vector is shared by all the test paths and so cannot possibly affect the subset relation.

A set of previously tested paths may leave a certain error space unchecked. If $\bar{e}$ is in that space, but $\bar{e}$ is detectable over subpath $P_A$, then any errors in the untested space which have $\bar{e}$ as a component will be detected. Testing along such a path will reduce the dimension of the undetected error space by at least one. This naturally suggests the following corollary:

Corollary 1. A set of testable subpaths $K=\{P_k\}$ all ending with $T^*_i$ is sufficient for $T^*_i$ if

$$\cap_{k \in K} \text{span}[\text{null-space}(C^T_k), \{\bar{r}_j\}] = \{\}. $$
Normally certain errors will remain undetectable no matter what test paths are used. These may occur because no options exist to certain assignments or equalities. For example, in figure 2 the statement "S=1;" immediately before PRED means that no path can detect errors of the form \( \vec{e} = (1,0,0,-1,0,0)^T \) which would result in predicates like "IF U+S>B+1". Alternatively, some errors will go undetected because some functional relationship is preserved along all paths. In figure 2 the path for \( A>2 \) transforms the environment to \( C_A\vec{V}_0 = (1,a,b,1,2a,a)^T \). Applying the rules for assignment blindness to this environment shows that testing along this path misses errors in PRED involving the expressions "A-U" and "2*A-T". The path for \( A<2 \) with an environment \( C_B\vec{V}_0 = (1,a,b,1,2a+2b,a+b)^T \) is blind to error expressions "A+B-U" and "2*A+2*B-T". But neither path will detect the expression "T-2*U" if it is added to the predicate, because for both paths that expression is a linear combination of the undetected errors.

Such errors are undetectable for any test path and hence for any input data. Consequently they have no real bearing upon the correctness of the program, but they do complicate the problem of judging when a predicate has been sufficiently tested. Although we may reasonably believe a set of paths to be nearly sufficient because they reduce the dimension of the untested space to a small number, the smallest possible dimension of that space may not be known.

In this context, the value of a proposed test path may be measured by the number of dimensions it would subtract from the total untested space.

The computations necessary to find the null space of \( C_A^T \) and the \( \vec{e}_j \) are not as difficult as the notation might suggest. Since the characteristic vectors of these spaces represent the variable assignments and equality interpretations along the test path, both are directly derivable from a symbolic execution without requiring an
explicit construction of the $C_i$ matrices. The procedure for computing the spanning vectors of the error space described by Theorem 1 is based on and of the same order of complexity as Gaussian elimination [GEWIA74]. Such a procedure is described in the Appendix.

To demonstrate how these results can be applied without rewriting the computations in terms of matrices, consider the program shown in figure 5. This program computes the greatest common factor of any two positive integers by what must be one of the oldest of algorithms, Euclid's algorithm. Rather than breaking the program into individual matrices, we shall work directly from a symbolic execution. The

```plaintext
INPUT A, B
S := A
T := B
U := 0
T1: WHILE S ≠ T DO
T2: IF S > T THEN
    S := S - T
ELSE
    U := S
    S := T
    T := U
ENDIF
ENDWHILE
T3: IF S = 1 THEN
    PRINT A, " AND " B, " ARE RELATIVELY PRIME."
ELSE
    PRINT "THE GREATEST COMMON FACTOR OF " A, " AND " B, " IS " S
ENDIF

Figure 5: Euclid's algorithm for GCF
```
predicates are labelled in order to provide reference points for describing paths. Paths will be specified by listing the predicate names in the order in which the predicates are encountered, followed by "T" or "F" to indicate the branch chosen according to the truth or falsehood of the predicate. Thus the shortest complete paths through the program would be (T1:F, T3:T) and (T1:F, T3:F). Subpaths will be denoted by listing all predicates up to and including the end of the subpath, with the final predicate marked with "?" since no branch has yet been chosen following that predicate. Thus the portion of those shortest paths up to the final predicate will be indicated by (T1:F, T3:?). This is the form which would be given for testing T3 along the shortest possible subpath, since it is only the statements preceding T3 which are of concern when selecting paths. It is likely that the second stage of path analysis testing would complete this path, determining whether T3 should be true or false.

The environment vector for this program will contain six elements, corresponding to (1, A, B, S, T, U)^T. Let's begin by testing T1. The natural starting point is the (T1:?), the subpath consisting of the first five program statements. The symbolic execution along (T1:?), is:

\[
\begin{align*}
S &= A \\
T &= B \\
U &= 0.
\end{align*}
\]

The assignment blindness vectors are therefore given by the expressions "S=A", "T=B", and "U". The self-blindness vector is based on the expression "S-T" (from "S - T ≠ 0" as the simplified form of the predicate T1). There are no equality restrictions. The undetectable space for (T1:?), is therefore the space spanned by the columns of the following matrix:
Hence if testing along the subpath (T1:?) does not reveal an error, the predicate T1, whose functional part here is "S - T", may be incorrect with the correct form being "A - T", "S - B", "S - T + U", or any of the infinite expressions which can be obtained from combinations of the four characteristic errors which cannot be detected using this test path.

Next consider the path (T1:T, T2:T, T1:?). The symbolic execution gives

\[
S = A - B \\
T = B \\
U = 0.
\]

With no equalities and the same self blindness vector (since the same predicate is being tested), the undetected error space for this subpath is the span of the columns of

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
1 & 0 & 0 & -1 \\
0 & 1 & 0 & 1 
\end{bmatrix}
\]

The set of errors which will have escaped detection using both of these paths is given by the intersection of the two sets of vectors, which is the span of
The set of errors undetectable after testing with both paths is therefore given by the expressions "S - B", "T - B", and "U".

Since the "ELSE" branch of T2 has not yet been executed, the subpath (T1:T, T2:F, T1:?/T1:T) is the natural next choice. Unfortunately this path is not testable since the final predicate interpretation in (T1:T, T2:F, T1:T) is redundant while (T1:T, T2:F, T1:F) is infeasable. Consequently an extension of that path will be used - (T1:T, T2:F, T1:T, T2:T, T1:?/T1:T). After execution along this subpath the variables will have the values:

\[ S = B - A \]
\[ T = A \]
\[ U = A \]

This gives the following blindness space:

\[
\begin{bmatrix}
0 & 0 & 0 \\
-1 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

The intersection of this space with the total undetected space after the first two paths is
The total undetected space for T1 is therefore generated by the expressions \( U + S - B \) and \( T - S \). Since the latter is the self blindness vector which can never be eliminated, the next path will be chosen so that the first expression does not evaluate to zero. The simplest way to alter the relation between U and S is to simply extend the last test path by one additional iteration. The next test path will therefore be \((T1:T, T2:F, T1:T, T2:T, T1:T, T2:T, T1:?)\).

The variables after this subpath has been executed will be

\[
\begin{align*}
S &= B - 2A \\
T &= A \\
U &= A
\end{align*}
\]

The blindness space for this path is therefore

\[
\begin{vmatrix}
0 & 0 & 0 & 0 \\
-2 & 1 & 1 & 0 \\
-1 & 0 & 0 & -1 \\
0 & -1 & 0 & 1 \\
0 & 0 & -1 & 0
\end{vmatrix}
\]

Taking the intersection of this and the total undetectable space yields:

\[
\begin{vmatrix}
0 \\
0 \\
-1 \\
0
\end{vmatrix}
\]

The total undetected space has been reduced to the self blindness vector. Since this error term can never be eliminated, we have found a sufficient set of test paths for T1.
Testing the predicate $T_2$ is now quite simple. There are no assignment statements between $T_1$ and $T_2$, so the assignment blindness vectors are the same. Both predicates have the same functional form, "$S - T$", so the self blindness vector is unchanged. As with $T_1$, there are no equalities to be concerned with. The net result is that extending each of the test paths for $T_1$ by assuming $T_1$ is true gives a path with the same blindness space for $T_2$ as for $T_1$, so the same set of paths form a sufficient test set for $T_2$.

The situation is not so simple when choosing test paths for $T_3$. This predicate can be reached only upon exiting the loop, at which time predicate $T_1$ must be false. This means that the condition "$S = T$" must hold, so test paths for $T_3$ will always contain an equality restriction. If, for example, the subpath $(T_1:F, T_3:')$ is chosen, the symbolic execution gives:

\[
\begin{align*}
S &= A \\
T &= B \\
U &= 0
\end{align*}
\]

The condition "$S = T$" evaluates to "$A = B$" when the symbolic results are substituted for the variables $S$ and $T$. The blindness space for this path is therefore given by:

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & -1 \\
1 & 0 & 0 & -1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
-1 & 0 & 0 & 0 & 1 \\
0 & -1 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0
\end{bmatrix}
\]

The first three columns are assignment blindness vectors, the next column due to equality blindness, and the last is the self blindness vector for $T_3$. 
If we continue testing T3 by simply extending each of the four test paths for T1 (i.e., assuming the last interpretation of T1 to be false so that T3 is the next predicate to be chosen), the total space of undetectable errors for T3 will be

\[
\begin{pmatrix}
0 & -1 \\
0 & 0 \\
1 & 1 \\
-1 & 0
\end{pmatrix}
\]

The second vector here is still the self blindness vector which, as has been noted before, cannot be eliminated. Interestingly, the first vector denotes the expression "s - T", which must evaluate to zero since "S=T" is the condition for exiting the loop. It is interesting to note that this form is obtained even though the blindness spaces before the intersections are taken have this condition expressed in terms of the input values rather than the variables S and T. One consequence of utilizing this testing criterion is that eventually all invariant linear relations for a program will be derived at each predicate. Since this relation must hold for any halting path, the set of test paths for T3 is completed.

It is interesting to note how testing would have proceeded if the fourth statement of this program, the statement which initializes the variable U, had been omitted. With U undefined there would be no way to construct an assignment blindness vector. This raises a problem since it is syntactically possible that the correct predicate should have involved U. (This would, of course, imply that an error also existed in one of the computations in that U should have been defined.) Since the value returned by a reference to U may vary from run to run or at different times within a run, no inferences can be reasonably drawn regarding error expressions involving U. The best that can be done is to assume that all expressions involving U are undetectable. The undetectable error space will then no longer be a vector space. Even worse, such an assumption may force extra test paths to be chosen in
which U is defined. These extra paths might be unnecessary if U had been defined for all paths.

One reasonable alternative is requiring the programmer to assert that the predicate in question could not possibly involve U. Such assertions are made frequently, and are often enforced by the programming language syntax, when segmenting a program into modules. Within a module, however, justifying such assertions is far more difficult and is likely to be more error-prone.

Figure 6 shows a case where the inability to deal with error terms involving undefined variables could be critical. The only use of the variable Y in this program is as a loop control variable, ensuring that at least one iteration of the loop occurs and that the loop is abandoned as soon as X becomes negative after the first iteration. The error term in the loop predicate is "Y - X". All paths in which the loop has

```
Correct Code

INPUT A
X := A
Y := 0
WHILE Y >= 0 DO
  X := f(A,X)
  Y := X
ENDWHILE
.
.
.

Incorrect Code

INPUT A
X := A
WHILE X >= 0 DO
  X := f(A,X)
  Y := X
ENDWHILE
.
.
.
```

Figure 6: Error Terms Involving Undefined Variables
already been executed are blind to this error. Hence this error is
detectable only using the path leading up to, but not including, the
first iteration of the loop. However, \( Y \) is undefined along this
subpath. All expressions involving \( Y \) must therefore be classed as
undetectable, and so no guarantee can be made that the predicate error
will be detected. Since this is the only place in the program where \( Y \)
is used, it is also apparent that the computation error implied by the
missing definition of \( Y \) cannot be detected in general. The message is
clear. Variables must be properly initialized if testing is to maintain
any level of confidence and consistency.

At this point we have characterized those predicate errors which
escape detection for a given test path and have shown that the value of
a test path lies in its ability to reduce the space of potentially
undetected errors. We have yet to justify the claim that a small,
finite set of paths will be sufficient for detecting predicate errors.
This is accomplished in the final theorem.

Theorem 3. Minimal Set for Sufficient Testing

A minimal set of subpaths sufficient for testing a given predicate in a
linearly domained program will contain at most \( m+n \) subpaths, where \( m \) is
the number of input values and \( n \) the number of program variables.

After a single path has been tested, the untested error space due
to assignment blindness is of dimension \( n \), and the space due to equality
blindness is at most dimension \( m \). In constructing a minimal sufficient
set of test paths, any subpath which fails to reduce the dimension of
the total untested space by at least one would be rejected under Theorem
2. After testing two paths the dimension of the total untested error
space is at most \( m+n-1 \). Continuing in this fashion it is clear that a
minimal sufficient test set can have at most \( m+n+1 \) paths minus the
minimum possible dimension of the space of undetected errors. That minimum dimension is 1, since the self-blindness vector \( T^1 \) will always be present. Hence no more than \( m+n \) paths will be required for that set.

The importance of this theorem is that it shows that a finite number of test paths will suffice for a wide class of programs. This limit is linear in the number of inputs and variables, so it should not grow inordinately large. Furthermore in most cases this limit should prove to be unnecessarily pessimistic, for several things may act to reduce the actual number of paths required. If the number of equalities is small, the dimension of the initial untested space will be reduced. If paths with widely different computations are used, the untested space due to assignment blindness can be reduced by far more than one dimension at a time. This subject will be discussed in more detail in the next chapter.

To summarize the results of this chapter, in programs where the program predicates and the computations affecting control flow are linear in the input variables it is possible to determine the set of predicate errors which must escape detection for a given test path. Although linearity itself yields considerable simplification, another implication of this assumption is conceptually more important. Restricting predicate interpretations to a well-behaved functional class makes possible the description of the infinite set of possible predicate errors using a small finite set of linearly independent, characteristic errors. This characterization has led to criteria for determining whether a proposed test path is capable of detecting any errors not already revealed by previous tests. These criteria are directly derivable from the assignments and equality predicates encountered along the test paths. The value of a test path is defined in terms of its ability to eliminate one or more of the characteristic errors which had escaped previous tests.
The number of test paths which may be selected under these criteria is limited by the finite number of independent errors. For linearly domained programs any predicate may be sufficiently tested using at most $m+n$ paths where $m$ is the number of program input values and $n$ is the number of program variables. This limit is independent of the complexity of the program control flow.

These results do not constitute a method for selecting paths for testing. The question of which paths are to be examined under this criterion is taken up in the next chapter.
IV. Path Selection

Surprisingly little has been written regarding strategies for selecting paths for testing, while much of what has been written has been largely ad hoc, with little analytical or empirical justification. Many of those who have implemented testing systems have chosen to leave this process unspecified or to require the programmer to specify the test paths before the automated strategies could begin [CLARL76, RAMAC76, WHITL80]. Such an approach effectively shifts the programmer's burden from the choice of test points to the choice of paths along which testing will be conducted. This approach is questionable, at best, considering the importance of path selection to the reliability of testing as revealed in the previous chapter.

One means of aiding the path selection process would be to determine the current set of undetectable error expressions and ask the programmer to choose test paths for which those expressions did not evaluate to zero. This could be extremely effective as long as the error expressions remain reasonably simple. It seems quite reasonable to ask the programmer to find a path for which "A - B" is nonzero (i.e., for which A and B are not equal). Asking the programmer to find a path for which "3.5*A + 2*B - C + 6" does not equal zero is less likely to be of help. It is not clear which of these alternatives is more likely to hold in practice. As testing continues, the remaining undetectable expressions will converge towards a set of relations which are invariant
over the entire program. A program may compute an elaborate function even though the functions associated with each subdomain are quite simple. Similarly one must believe that invariant relations which hold over all subdomains of a program can become quite complex. The question is whether such relations would be expressible as combinations of simpler expressions, and whether the undetected error expressions would tend to become simpler or more complex as repeated testing narrowed the undetected space. It is possible that heuristics could be found for transforming a set of spanning vectors into a "simplest" form to aid this process.

While such an approach may be of benefit, it still leaves the major burden of path selection to the programmer. It would certainly be welcome if the actual selection could be automated. The problems involved are not to be underestimated. Simply attempting to find a path through the particular predicate we wish to test will require determining whether any input point exists which will cause execution of that predicate. In general, the problem of determining the existence or nonexistence of such a point is unsolvable. Consequently most of the research in path selection has concentrated on heuristic methods.

Perhaps the best known strategy for selecting test paths is the use of one of the following coverage measures [CLARL76,GOODJ75,HOWDW75]:

1. All statements in the program must be executed at least once.
2. All branches in the program must be executed at least once.
3. All paths through the program must be executed at least once.

The first measure requires that each program statement must appear in some test path, but makes no statement regarding what combinations of statements must be tested together. The second measure requires each predicate to be tested using both true and false values. This is more stringent than the first measure, since testing all branches implies that all statements will have been tested. The last measure guarantees
that, not only is each predicate tested using both true and false values, but that all feasible combinations of predicate evaluations are employed for testing.

These measures provide an ascending scale of confidence (and of difficulty). If we assume a reliable strategy is employed for the second stage (testing the selected paths), then the third measure clearly provides the maximum possible confidence in the correctness of the program being tested. Unfortunately the third measure is seldom attainable. The presence of a simple DO-WHILE loop in a program may be sufficient to generate an infinite number of paths. Even where this is not the case, the number of paths for even simple programs can be prohibitively large. The program flowchart in figure 7, for example, can easily be shown to contain over $10^{12}$ paths even though each loop can be executed no more than 12 times.

Figure 7: Program With $10^{12}$ Paths
This high level of complexity arises from defining a measure in terms of combinations of program elements. The other two measures are much less difficult to attain, since they are expressed directly in terms of textual elements of the program. However neither of these provides a high level of confidence in the program under test. Consider, for example, a program whose output will be erroneous whenever the variable A is equal to zero. Then the first measure fails if the following code is encountered:

\[
A := 0 \\
\text{IF } B > 0 \text{ THEN } A := A + 1
\]

If our goal is merely guaranteeing that all statements are executed, we would be content with the path taken when B is greater than zero. By not testing the case where the predicate fails, we would miss the error.

The second coverage measure avoids that problem by specifying that both branches of the IF statement must be exercised. While this is sufficient for the above example, it fails when an error depends on a particular combination of branches. For example, if the same program were run with the code given above replaced by:

\[
\text{IF } B > 0 \text{ THEN } A := 2 \\
\text{ELSE } A := 3 \\
\text{IF } C > 0 \text{ THEN } A := A - 1 \\
\text{ELSE } A := A - 2
\]

then the second coverage measure cannot guarantee that the error will be detected. In this case detection is possible only when the path for \(B > 0\) and \(C \leq 0\) is followed; but this coverage measure would be satisfied if we chose the test path for the case \(B > 0\) and \(C > 0\) and the path for \(B \leq 0\) and \(C \leq 0\), neither of which will serve to detect the error. In general this measure is inadequate whenever detection of an error depends on the branches selected for some combination of two or more predicates.
Some intermediate measure is required which falls between the second and third measures in the sense of retaining the reliability of the third measure but requiring only a finite number of test paths.

One method frequently employed to circumvent this problem is to limit the choice of test paths to those paths which involve no more than \( k \) iterations of any loop, with the value of the parameter \( k \) being an arbitrary choice of the implementer or the tester. No theoretical justification has been presented for such a limitation, nor has any means of determining \( k \) been proposed. These questions will be addressed in this chapter.

However even if such a limit on loop iterations is assumed, the problem still remains difficult due to the extremely large number of paths to be considered. Even a loop-free program like that in figure 7 can have prohibitively many paths. For this reason more stringent restrictions on the choice of test paths are frequently imposed. Howden proposed the definition of classes of related paths with the goal for path selection being to choose at least one from each class. The formation of these classes was designed to guarantee that for each "If" statement, paths taking different branches would be placed in different classes, and for each loop classes would be established for paths executing that loop exactly once and more than once [HOWDW75].

Almost any strategy for path selection is hampered by the need to guarantee the feasibility of the chosen paths. This problem has been circumvented in an interesting approach by Kundu [KUNDS79], where the set of path constraints are generated for the last chosen path and an input data point selected which violates at least one of those constraints. The next test path will then be the path followed by the execution for that data point. This path is guaranteed to be feasible since there is known to be at least one input which causes its execution
- the very data point used to generate the path. It is almost impossible to "direct" this method towards a particular portion of the program, so it is ill-suited for use even with simple coverage measures, much less with more sophisticated measures of testing coverage which are devised in terms of the program constructs. Nonetheless, this approach could be of use as a supplement to a more rigorous selection method. In particular, since the criteria derived in Chapter III for predicate testing describe only a subpath leading up to the predicate being tested, it may be necessary to complete the path before testing can be done, depending on the method employed to test the chosen paths. Simply choosing a point within the subpath's domain and taking the path followed by that point is probably the simplest possible way of completing the selected path.

The previous chapter derived the space of undetected errors for a given test path in terms of the computations and predicates along that path. Only a single predicate was considered then. The sets of errors which go undetected for different predicates along the same path are not unrelated. This chapter will begin by considering the question of how undetected errors "propagate" along a path as a preliminary step to discussing the class of paths which need to be examined as candidates for testing. This information will be used to develop an algorithm for selecting test paths. The remainder of this chapter will consider heuristics for reducing the cost of that algorithm. The most important of these will result from a theoretical justification of the practice of imposing limits on the number of loop iterations required for testing.

Suppose that some initial subpath \( P \) can be separated into two parts, an initial subpath \( P_A \) ending at predicate \( T_A \), and a subpath \( P_B \) ending with \( T_B \) such that \( P \) is the path formed by following \( P_A \) and then continuing along \( P_B \). Now select an error \( \tilde{e} \). If the total path \( P \) is blind to \( \tilde{e} \), then it is possible that this blindness is entirely due to \( P_B \). For example, if the only computation in \( P_B \) were the assignment "\( X := 1 \)", then the total path \( P \) will be blind to the error expression
"X-1", no matter what computations are contained in the first half of the path, $P_A$. If this is not the case, then there must exist a transformed version of $\tilde{e}$ to which the subpath $P_A$ is blind. This is established in the following lemma:

Lemma 1. If $\tilde{e} \in \text{span}[\text{null-space}(C_A^T), \{\tilde{r}_{AB}\}]$ and $\tilde{e} \notin \text{null-space}(C_B^T)$ then $C_B^T\tilde{e} \in \text{span}[\text{null-space}(C_A^T), \{\tilde{r}_{AB}\}]$.

The notation $\tilde{r}_{AB}$ is used to denote the set of equalities for the path represented by the concatenation of $P_A$ with $P_B$. The vector $C_B^T\tilde{e}$ represents a transformed $\tilde{e}$. The transformation in this case consists of substituting for each of the variables in $\bar{v}$ their values before the computation $C_B$ was performed, in other words, a partial back substitution.

Proof:

$\text{null-space}(C_A^T C_B^T) = \text{span}[\text{null-space}(C_B^T), \{\bar{y}: C_B^T \bar{y} \in \text{null-space}(C_A^T)\}]$

So we are given that

$\tilde{e} \in \text{span}[\text{null-space}(C_B^T), \{\bar{y}: C_B^T \bar{y} \in \text{null-space}(C_A^T)\}, \{\tilde{r}_{AB}\}]$.

Multiplying by $C_B^T$ on both sides, the first term on the right drops out and the second is simplified:

$C_B^T\tilde{e} \in \text{span}[\text{null-space}(C_A^T), \{C_B^T \tilde{r}_{AB}\}]$.

Since the $\tilde{r}_{AB}$ are in terms of input values and constants only,

$C_B^T \tilde{r}_{AB} = \tilde{r}_{AB}$ and

$C_B^T\tilde{e} \in \text{span}[\text{null-space}(C_A^T), \{\tilde{r}_{AB}\}]$,

and the lemma is proven.
If, on the other hand, the error $\tilde{e}$ were detectable along the total path $P$ then a stronger inference can be made. The later portion of the path must not be blind to $\tilde{e}$, otherwise the entire path would also be blind to $\tilde{e}$. This idea is captured in the next lemma:

Lemma 2. If $\tilde{e} \not\in \text{span[null-space}(C_A^TC_B^T)\{F_j^{AB}\}]$ then the following are true:

$$\tilde{e} \not\in \text{null-space}(C_B^T)$$

and

$$C_B^T\tilde{e} \not\in \text{span[null-space}(C_A^T)\{F_j^{AB}\}].$$

Proof: Both equations follow directly from the following observation:

$$\text{null-space}(C_A^TC_B^T) = \text{null-space}(C_B^T) \cup \{\tilde{y}: C_B^T\tilde{y} \in \text{null-space}(C_A^T)\}$$

In each of the above lemmas, the set of equality restrictions remained unchanged even as the path grew shorter. The third lemma provides a means of dealing with equality blindness errors or any other blindness vector to eliminate its contribution to some detectable error.

Lemma 3. Given errors $\tilde{e}$ and $\tilde{f}$ and a subpath $P$ such that $P$ is blind to $\tilde{f}$ but not blind to $\tilde{e}$, then $P$ is not blind to the vector $\text{ortho}(\tilde{e}, \tilde{f})$ where

$$\text{ortho}(\tilde{e}, \tilde{f}) = \tilde{e} - (\tilde{e} \cdot \tilde{f})\tilde{f}/(\tilde{f} \cdot \tilde{f}).$$

Proof: The vector $(\tilde{e} \cdot \tilde{f})\tilde{f}/(\tilde{f} \cdot \tilde{f})$ represents the component of the
detectable error $\tilde{e}$ which is contributed by the undetectable error $\mathcal{F}$. We would expect that by subtracting this "dead wood" from $\tilde{e}$, the remainder would still be detectable. If this were done for all undetectable errors, then we would be left with the "kernel" of $\tilde{e}$ which permitted its detection.

Note that

$$\mathcal{F} \cdot \text{ortho}(\tilde{e}, \mathcal{F}) = 0$$

$$\tilde{e} = \alpha \mathcal{F} + \text{ortho}(\tilde{e}, \mathcal{F}).$$

Assume by way of contradiction that $P$ is blind to $\text{ortho}(\tilde{e}, \mathcal{F})$. The set of errors to which $P$ is blind is a vector space and is therefore closed under addition. Since $P$ is blind to $\mathcal{F}$, it must be blind to $\tilde{e}$ since $\tilde{e}$ is a linear combination of $\mathcal{F}$ and ortho$(\tilde{e}, \mathcal{F})$. This contradicts the terms of the lemma, so $P$ is not blind to ortho$(\tilde{e}, \mathcal{F})$ and the lemma is proven.

The property of ortho$(\tilde{e}, \mathcal{F})$ which makes it of interest is that it is orthogonal to $\mathcal{F}$. Any detectable error can have components which are undetectable due to equality blindness. This lemma allows us to strip away such components as the equality restrictions are discovered in order to determine the portion of the error which must remain detectable.

To see how these lemmas can be used, let's return to the greatest common factor program in figure 8. When $T_1$ was being tested, we chose the subpath $(T_1:T, T_2:F, T_1:T, T_2:T, T_1:? )$. One of the errors which could not be detected was "$S - B + A"$. In vector form, this expression was $(0, 1, -1, 1, 0, 0)^T$. Now by lemma 1, this means that the error $C^T(0, 1, -1, 1, 0, 0)^T$ must be undetectable along the subpath $(T_1:? )$
where C represents the transformations during the two iterations of the loop (a purely arbitrary choice of places to split the path). We have already noted that multiplication by $C^T$ represents a back substitution using the assignment statements corresponding to C. Performing this substitution we find that $(0, 1, -1, -1, 1, 0)^T$ must have been undetectable on the subpath (T1:?). This vector corresponds to the expression "A - B - S + T". Indeed, when (T1:?) was tested we found that the expressions "A - S" and "T - B" were undetectable, so the lemma is correct in asserting that the simple sum of these two expressions is also undetectable.

```
INPUT A, B
S := A
T := B
U := 0
T1: WHILE S != T DO
    T2: IF S > T THEN
        S := S - T
    ELSE
        U := S
        S := T
        T := U
    ENDIF
ENDWHILE
T3: IF S = 1 THEN
    PRINT A," AND ",B," ARE RELATIVELY PRIME."
ELSE
    PRINT "THE GREATEST COMMON FACTOR OF ",
ENDIF
```

Figure 8: Euclid's algorithm for GCF
In a similar fashion, we can note that the error expression "A - S" is detectable on (T1:T, T2:F, T1:T, T2:T, T1:?). Performing the same back substitution as specified under Lemma 2 results in the claim that "T - S - B" must be undetectable on (T1:?). A cursory examination of the blindness vectors found for that path will show that "T - S - B" is not in the undetectable space, and so that error must indeed be detectable.

To demonstrate Lemma 3, consider the path (T1:F, T3:?). The error expression "S + A - B" will be detectable only if the vector formed by taking "ortho" of "S + A - B" and the equality restriction "A=B" is detectable. This orthogonalized vector will be (0, 0, 0, 1, 0, 0)^T, corresponding to the simple expression "S". So if "S" is detectable, "S + A - B" is detectable also.

These three "backup" lemmas provide a means of tracing an undetectable error back to its source within a path. At this stage it is possible to envision a path selection strategy based on these lemmas. Begin at the predicate to be tested, with the set of undetected errors for that predicate. For each of the linear computation blocks which are direct predecessors of this predicate, check whether for each undetected error \( \tilde{e} \in \text{span}\{\text{null-space}(C_i^T), \{r_j\}\} \), where \( C_i \) is the computation block being examined, \( \{r_j\} \) is the set of equalities accumulated so far in moving backwards, and \( T \) is the current predicate. If all the \( \tilde{e} \) are in that span, then any path leading up to this computation block and following the path taken so far will be of no use (by Lemma 1) and the investigation of this path is terminated. If any of the \( \tilde{e} \) are not in that spanning set, then back up through the computation block and repeat the whole process from that point, this time using the set of undetected errors \( \{C_i^T \tilde{e}\} \). This process halts when all paths have been terminated as described above, or when the start of the program is reached, indicating that a useful path has been found.
A certain degree of nondeterminacy is implied here, since at any point there may be many direct predecessors which are not rejected, each of which will have to be examined until one of them yields a complete path or all of them have terminated.

Realistically, it would also be necessary to perform a feasibility check at each step also, rejecting those paths for which no possible input data exists. It has been claimed that such feasibility considerations are not well suited to backward symbolic substitutions because inconsistent path conditions will be detected earlier using forward substitution [RAMAC76]. It is not clear why this should be, unless this claim is limited to searches for inconsistent input conditions. When checking for feasibility of backward substitution methods, the path conditions will generally be in terms of both input and program variables. The feasibility question then is simply modified from "Is there an input vector which causes this path to be executed?" to "Is there an environment vector which causes this (non-initial) subpath to be executed?". If, as has been claimed [GABON76], inconsistent path conditions can frequently be reduced to a pair of conflicting predicates, there is no reason to believe that backward substitution should be any slower at encountering these contradictory predicates than would forward substitution.

Another task necessary for this method is the accumulation of the set of equalities for the current path. As noted earlier, equality restrictions may be imposed on a path domain directly by the execution of an equality predicate or indirectly as the result of two or more inequalities, with the latter case named coincidental equalities in Chapter I. The problem of detecting these coincidental equalities has not yet been discussed. We shall do so here since this will be a necessary step for any path selection algorithm.
If feasibility checking is being employed, it is fairly easy to
tell when a coincidental equality has occurred. All that is required is
a feasible point, a set of vectors \( \{\delta_i\} \) spanning the legal space of
inputs (not counting the most recently encountered predicate), and the
predicate interpretations determining the current path condition. If \( \bar{v}_0 \)
is a legal input for the current subpath, and the last predicate has not
contributed to a coincidental equality, then for all \( i \) either \( \bar{v}_0+\delta_i \) or
\( \bar{v}_0-\delta_i \) or both will satisfy the current path condition, as shown in
figure 9. If, on the other hand, a coincidental equality has occurred,
then there will exist a \( \delta_i \) such that both \( \bar{v}_0+\delta_i \) and \( \bar{v}_0-\delta_i \) will fail to
satisfy the current path condition as illustrated in figure 10.

If a coincidental equality has occurred, then we must determine
what the equality condition is, both to add it to the set of equality

---

**Figure 9: Domain With No Equalities**

**Figure 10: Domain With One Equality**
blindness errors and to update the set of $\mathcal{E}_i$. This is more difficult than merely detecting when a coincidental equality has occurred. Clearly the last predicate may be treated as an equality. Thus if the predicate "IF $A \geq B$" causes a coincidental equality, the domain is restricted to points where $A=B$. This may not capture the entire restriction however. If, for example, previous predicates include "$B \geq C$" and "$C \geq A$", then the coincidental equalities resulting from "IF $A \geq B$" are not only $A=B$ but also $B=C$. Determining these additional restrictions is far more difficult and consequently more expensive in terms of computing time, but it can be done by using linear programming techniques to find a set of feasible points which span the dimensions of the domain.

Assuming that feasibility checks and detection of coincidental equalities will be employed, the path selection strategy can be expressed in more formal terms. Remember that a path consists of a list of indices of successive $(C_i, T_i)$ blocks in the order in which they are executed. Two subpaths may be joined to form a longer subpath. This operation will be designated as the addition of subpaths (e.g. $P_A + P_B$). An initial subpath is a subpath whose first computation corresponds to the legal starting point of the program. Finally, define Predecessor(P) as the set of indices of those $(C_i, T_i)$ pairs which can legally be executed immediately before the subpath P according to the rules of whatever programming language is being employed.

The procedure Select-Path($\vec{e}, P$) shown in figure 11 returns a set of one or more initial subpaths ending with P on which $\vec{e}$ is detectable. If no such path exists, it returns the empty set. Thus the call to find a path capable of detecting error $\vec{e}$ in the predicate $T_i$ would be Select-Path($\vec{e}, (i)$), since $(i)$ is the subpath consisting only of the pair $(C_i, T_i)$. While for clarity this procedure will be defined for a single error vector, it can be naturally extended to a set of undetected errors; or the procedure could be invoked separately for each characteristic error vector in the undetected space.
This procedure is based both on the discussion immediately preceding and the backup lemmas which opened this chapter. Line 2 invokes a feasibility check. If $P$ is infeasible then no path ending with $P$ can be feasible, and so the empty set is returned.

Lines 3 through 6 derive from the third lemma, dealing with the disposition of equalities. If an equality restriction has occurred, then $\vec{e}$ may not be detectable. However if $\vec{e}$ is still detectable for some path ending with $P$, then by lemma 3 the vector $\text{ortho}(\vec{e}, r_j)$ must be detectable along the same path. If $\vec{e}$ is not detectable then $\vec{e}$ must be in the span of the equality restriction vectors, and $\text{ortho}(\vec{e}, r_j)$ applied for all $j$ will yield a zero vector.

1) procedure Select-Path($\vec{e}, P$)
2)  if $P$ is infeasible then return {} 
3)  else if the first predicate in $P$
4)    causes equality conditions \{r_j\} then
5)        For all j  $\vec{e} := \text{ortho}(\vec{e}, r_j)$
6)    end if
7)  if $\vec{e} = 0$ then return {} 
8)  else if $P$ is an initial subpath then return \{P\} 
9)      else return [$\cup$ Select-Path($C_j^T\vec{e}$, (j)+P) 
10)        $\forall j$ in Predecessor(P)]
11)    end if
12)  end if
13)  end if
14) end Select-Path

Figure 11: Path Selection Algorithm
Zero vectors are captured in line 7, where the empty set is returned to indicate that \( \bar{e} \) has been found to be undetectable along \( P \).

When line 8 is reached, we know that \( P \) is feasible and not blind to \( \bar{e} \). If \( P \) is also initial, then the procedure has been successful and \( P \) is returned as an answer. If \( P \) is not initial, then it is necessary to back up one step more in the program. By the first two lemmas, \( \bar{e} \) is detectable along \( P \) only if for some \( C_j \) which immediately precedes \( P \) in some legal path. \( C_j \bar{e} \) is detectable along \( (j)+P \), the path \( P \) with \( (C_j,T_j) \) added at the front end. Hence in lines 9 and 10, Select-Path is invoked for all predecessors of \( P \), continuing the search over subpaths one step longer than \( P \). If there is no path ending in \( P \) which detects \( \bar{e} \), then those components of \( \bar{e} \) which are susceptible to equality blindness will have been trimmed away by lines 3-6, and those components which are susceptible to assignment blindness will go to zero under the transformation \( C_j \bar{e} \) in line 9. The result will be that \( \bar{e} \) goes entirely to zero, in which case it will be trapped in line 7.

This algorithm does not deal explicitly with self-blindness, since this error is easily eliminated. The self blindness vector is simply the predicate being tested, so by lemma 3 its effect can be disposed of by invoking Select-Path(ortho(\( \bar{e},T_i \)),(i)) when attempting to find a path for which \( \bar{e} \) is detectable. In this manner the portion of \( \bar{e} \) which is susceptible to self blindness can be eliminated at the start of the search.

To illustrate the workings of Select-Path, consider how it would behave when searching for a path in the GCF program of figure 8 which would detect the error expression "S - A" in predicate \( T_l \). The invocation would be

\[
\text{Select-Path}( (0, -1, 0, 1, 0, 0)^T, (?, T_l:?) )
\]
where the path is given with a leading question mark since the preceding portion is unknown. This path is feasible and involves no equality restrictions. The error vector is not zero, and the path is not initial. Control therefore passes to line 9 and the possible predecessors of T1 must be enumerated.

T1 has three predecessors. The first is the start of the program. The second is the block of code corresponding to T2:T, and the third is the block of code corresponding to T2:F. Considering the first of these, we substitute in the error expression using the assignment statements "S := A", "T := B", and "U := 0". This causes the expression "S - A" to become zero, so we invoke

Select-Path(0, (T1:?)).

Since the error term is zero, this will return the empty set.

Choosing the second of the three options, we get the path (? T1:T, T2:T, T1:?). The assignment statement "S := S - T" is substituted into the error expression "S - A" to give the next recursive call of Select-Path:

Select-Path( (0, -1, 0, 1, -1, 0)T, (? T1:T, T2:T, T1:?)).

The new error expression is "S - T - A". Once again control will pass down to line 9. Again the same three choices of predecessors to the current path will be available. The first choice will yield the call

Select-Path( (0, 0, -1, 0, 0, 0)T, (T1:T, T2:T, T1:?)).

The time control will pass down to statement 8 where, since this path is initial, the current path will be returned.
So at least one path will be chosen to detect "S - A", the path (T1:T, T2:T, T1:?). This was in fact one of the paths examined in Chapter III, and it was seen then that it did indeed detect the error "S - A".

Select-Path represents a straightforward application of the blindness measures to path selection. It may be helpful to think in terms of a tree of subpaths, with its root at the predicate being tested and each branch representing one of the possible predecessors of the, previous node. The advantage of this algorithm lies with the fact that branches can examined individually and eliminated (along with all dependent branches) after examining only the local branches, without requiring that each branch be extended all the way to a terminal node. The major problem which becomes apparent is that there is no limit to the depth of this tree, and no guarantee that the algorithm will halt. The problem here is the same one cited earlier for full path coverage. The presence of a loop implies the possibility of an infinite number of paths. If, when working backwards through the program. Select-Path determines e to be undetectable before or within that loop, then no additional iterations of the loop will be considered. If however Select-Path cannot make such a determination until it has backed up all the way through the loop using some number of iterations k, then the algorithm will continue trying k+1 iterations, then k+2, etc.

The assumption made by several researchers to limit the number of loop iterations to some fixed number begins to look very attractive here. While this limit has been imposed rather arbitrarily in the past, the theorems presented here provide a basis for a partial support of that assumption.

Theorem 4. Iteration Limit Theorem

Given that there exists an initial subpath P with more than m+n iterations of some single-entry single-exit loop and an error e
detectable along $P$, then there exists an initial subpath $P'$ with no more than $m+n$ iterations of that loop such that, ignoring coincidental equalities, $e$ is detectable along $P'$.

**Proof:**

Separate $P$ into three sections such that $P = P_F + P_C + P_H$ where $P_F$ represents the subpath before the loop, $P_C$ the portion within the loop, and $P_H$ the remainder following the loop.

Define $\text{Blind}(C)$ to be $\text{span}\{\text{null-space}(C^T), \{r_j\}\}$. Let $\{C_i\}$ be an enumeration of the possible transformations within a single iteration of the loop. Each $C_i$ represents the total transformation we might encounter in a single iteration. Since the interior of the loop may contain branches and other loops, there are many such transformations possible. Indeed the number of elements in the set $\{C_i\}$ may be infinite. Define the set of transformations obtainable from exactly $k$ iterations of the loop as $D_k$ where

$$D_1 = \{C_i\} \quad \text{and} \quad D_{k+1} = \{C_i C = C_i C' \mid C' \in D_k\}.$$  

Define $E_k$ as the space of errors which cannot be detected in $k$ or few iterations as follows:

$$E_0 = \{\text{all vectors of dimension } m+n+1\}$$

$$E_k = E_{k-1} \cap \left( \bigcap_{C \in D_k} \text{Blind}(CC_F) \right)$$
Clearly $E_{k+1} \subseteq E_k$. The $E_k$ may not be effectively computable, but this will not be necessary for our purposes. The $E_k$ represent the errors which would be undetectable for a predicate placed exactly at the exit point of the loop with no assignment statements in between. To prove this theorem we will first prove the existence of a limit on the number of iterations required to reduce sequence $\{E_k\}$ to its minimum possible dimension.

Lemma 4. Given a $k$ such that $E_k = E_{k+1}$ then for all $j$ greater than $k$, $E_j = E_k$.

Proof: Given such a $k$, assume by way of contradiction that

$$E_k = E_{k+1}$$

For this to hold, there must exist some error $\bar{e}$ which is in $E_k$ and $E_{k+1}$ but not in $E_{k+2}$. If $\bar{e}$ is not in $E_{k+2}$ then there must exist a path involving $k+2$ iterations of the loop which is not blind to $\bar{e}$. We will designate the computations along this path as $C_a C_t C_b$, where $C_b$ represents the transformations up to and including the first $k$ iterations of the loop, $C_t$ represents the transformation employed along the $k+1$ iteration, and $C_a$ the transformation encountered during the $k+2$ iteration of the loop. By definition, $C_a$ and $C_t$ must be in $D_1$. We will designate the equality restrictions encountered along each of these path segments as $\bar{F}_j^b$ corresponding to $C_b$, $\bar{F}_j^{B_t}$ for the additional equalities imposed during the $k+1$ iteration, and $\bar{F}_j^{B_t s}$ for those encountered during the $k+2$ iteration.

Since we have stated that this long path is not blind to $\bar{e}$,

$$\bar{e} \notin \text{span}\{\text{null-space}(C_t^T S T), \{\bar{F}_j^b\}, \{\bar{F}_j^{B_t}\}, \{\bar{F}_j^{B_t s}\}\}.$$  \hspace{1cm} (8)

However, we have asserted that $\bar{e}$ cannot be detected using any path with
only \( k \) iterations of the loop:

\[
\bar{e} \in \text{span}[\text{null-space}(C_B^T), \{f_j^B\}].
\]

Define \( \bar{e}_g = C_B^T \bar{e}_g \). Note that \( \bar{e}_g \) cannot be zero since that would mean \( \bar{e} \) was in the null-space\((C_B^T C_g^T C_s^T)\). Then by Lemma 1

\[
\bar{e}_g \notin \text{span}[\text{null-space}(C_B^T C_g^T), \{f_j^B\} u \{f_j^{B_k}\} u \{f_j^{B_k^B}\}].
\] (9)

By application of one of the "backup" lemmas, the condition that \( \bar{e} \) be detectable using \( k+2 \) iterations has been transformed into a requirement that \( \bar{e}_g \) be detectable using \( k+1 \) iterations.

Recall that \( E_{k+1} = E_k \). Since \( \bar{e} \) is in \( E_k \), it must also be in \( E_{k+1} \). Consider the path formed by removing the \( k+1 \) iteration from the \( k+2 \) iteration path described above. Since this new path has \( k+1 \) iterations it must be blind to \( \bar{e} \).

\[
\bar{e} \in \text{span}[\text{null-space}(C_B^T C_s^T), \{f_j^B\} u \{f_j^{B_k}\}]\] (10)

The term \( f_j^{B_k} \) denotes the equalities encountered during execution of the \( C_s \) iteration. Since these equality restrictions depend on the environment at the time they are encountered, these cannot in general be assumed to be equivalent to the sets \( \{f_j^{B_k}\} \) or \( \{f_j^{B_k^B}\} \) encountered earlier.

Comparing equations (8) and (10), an interesting observation can be made. The spaces denoted by the null-space and that denoted by the equality restrictions are disjoint. Since \( E_k = E_{k+1} \) both of these equations must hold for any \( \bar{e} \), so whatever component of \( \bar{e} \) lies within the equality space of equation (10) must also lie within the equality space of equation (8).
\[ \bar{e} \in \text{span}\{\text{null-space}(C_B^T C_B), \{r_j^B\}\} \]

Now by Lemma 1,

\[ \bar{e}_b \in \text{span}\{\text{null-space}(C_B^T), \{r_j^B\}\} \quad (11) \]

So if \( \bar{e} \) is in \( E_k \) and \( E_k = E_{k+1} \), then \( \bar{e}_b \) is also in \( E_k \). Once again, any error which is undetectable along all paths having \( k \) iterations of the loop is also undetectable along paths having \( k+1 \) iterations. So extending the path of equation (11) by adding one more iteration using computation \( C_k \), we have

\[ \bar{e}_b \in \text{span}\{\text{null-space}(C_B^T C_B^T), \{r_j^B\}\} \]

However this contradicts equation (9), and so the assumption that \( E_{k+1} \subseteq E_{k+2} \) fails and we have

\[ [E_k = E_{k+1}] \rightarrow [E_{k+1} = E_{k+2}] . \]

By induction, the lemma holds.

Returning now to the main proof, we are given a path \( P = P_F + P_G + P_H \) with transformations \( C_H C_G C_F \) and an error \( \bar{e} \) which is detectable along that path. By lemma 2, \( \bar{e} \) cannot be in the null-space of \( C_H \). If \( \{r_j^H\} \) represent the equality restrictions imposed by the segment \( H \) of this path, then by Lemma 3 we can find a vector \( \bar{f} = \text{ortho}(\bar{e}, r_j^H) \) (applied successively over all \( j \)) which is also detectable along this path but has no components corresponding to the equality predicates in subpath \( P_H \). By lemma 2 the error \( C_H^T \bar{f} \) must be detectable along the path \( P_F + P_G \).
We are given that segment $P_G$ involves more than $m+n$ iterations of some loop; but since $E_0$ has dimension at most $m+n+1$, and $E_k$ has dimension at least 1, by lemma 4 any error which cannot be detected in $m+n$ or fewer iterations cannot be detected using more iterations. Hence there must exist a subpath $P_I$ involving no more than $m+n$ iterations of the loop on which $C_H$ is detectable.

Now since $C_H$ is detectable on $P_I+P_F$ and $P$ has no components corresponding to the non-coincident equalities in $P_H$, $P$ must be detectable along $P_H+P_I+P_F$ and the theorem is proven.

Having established the theorem, let us now consider its limitations. One limiting assumption was that no coincidental equalities occur in the portion of code following the loop. Clearly this requirement becomes more and more difficult to justify as we move further from the loop, encountering larger numbers of predicates. It is not clear how constraining this assumption really is, nor whether this assumption is truly a requirement of the theorem or a limitation of the model.

One can argue that for floating point calculations, coincidental equalities represent a dangerous construction and should be avoided. As noted earlier, these equalities arise from the exact coincidence of two or more inequality expressions (for example, IF $A<B$ ... followed by IF $B<A$ ... where both expression evaluate as true). The danger here lies in fact that a special domain has been created which is entered only when $A-B$ is exactly zero. The behavior of such programs is determined as much by round-off error as by the intent of the programmer. Hence coincidental equalities for floating point variables should be avoided.
The situation is far different for integer variables. Here coincidental equalities are not only benign, but are quite routine. Consider the program segment charted in figure 12. Clearly if I and N are integers, then the choice of a path which executes the loop k times imposes the equality constraint \( N=k \). Yet this is nothing more complicated than an ordinary FORTRAN DO-loop. (Note that if I and N were floating point numbers, we would not have an equality but merely the constraints that \( N > k-0.5 \) and \( N \leq k+0.5 \).) The situation can become complicated when variable increments are involved. Consider the loop formed by \( \text{DO } I=I_1, N2, M \). The conditions which hold upon exiting the loop after k iterations will be

\[
(k-1)M + N1 < N2 \quad \text{and} \quad N2 \leq kM + N1.
\]

Now if \( M>1 \) these conditions form parallel borders of the subdomain for the path executing that loop \( k \) times. When \( M=1 \), however, these conditions degenerate to

![Diagram of Counting Loop](image)

Figure 12: Counting Loop
\[ k - 1 + N_1 < N_2 \quad \text{and} \quad N_2 \leq k + N_1. \]

A coincidental equality occurs since these conditions are equivalent to

\[ N_2 = k + N_1 \]

when \( N_1 \) and \( N_2 \) are integer variables. Using this loop, coincidental equalities arise only when \( M=1 \), so we may not even be able to tell whether a particular loop will present problems or not. Clearly some special consideration will have to be given to coincidental equalities on integer variables.

Perhaps the most serious limitation of this theorem has to do with feasibility and redundancy. The predicate at the end of a subpath is called "testable" if the subpath is feasible and the predicate interpretation along that subpath is not redundant. The Iteration Limit Theorem guarantees only that a short path path exists for which the error is detectable; it does not provide any assurance that this shorter path is testable. Of course, there is also no requirement (or guarantee) that the longer path be testable if the theorem is to hold.

It therefore appears likely that this theorem, even if it could be strengthened to include coincidental equalities, does not provide a halting algorithm for path selection (i.e. test only paths with fewer than \( m+n+1 \) iterations). Nevertheless, we shall advance this as a testing heuristic which brings the problem of path selection into a (barely) manageable domain. This is not entirely arbitrary. Both coincidental equalities and testability problems are caused by the effects of the conjunction of a number of predicates. As such, we would expect such problems to increase as we move to longer and longer test paths. It is worth noting that most of the research done in this area has started with the assumption that an arbitrary limit could be placed on loop iterations, with far less justification and no good way of determining what the actual limit should be.
If we consider the conditions under which testability issues are likely to cripple a method using this heuristic, one common practice stands out. Many loops use variables as simple counters. A subsequent predicate involving such counters (for example, checking to see if the loop was repeated k times, where k is a constant) will frequently be testable only after some minimum number of iterations has been taken. Therefore we will phrase our suggested heuristic as follows: For any loop in the program, test only paths which perform that loop no more than m+n iterations more than the minimum required for a testable subpath.

This does cloud the issue slightly. The minimum number required for a testable subpath may not be known. But operationally, any short testable subpath provides an upper bound on that number.

We have not yet considered the full richness of the Iteration Limit Theorem. Our original discussion concerned only a single isolated loop with arbitrary internal complexity. This restriction was glossed over when we generalized this result for the above heuristic. Clearly the theorem has no problems with extension to consecutive non-nesteded loops. But what of nested loops? Since the theorem proof is independent of the surrounding structure, we could claim that the iteration limit applies to the inner loop each time we repeat the outer loop. In fact, we can make a much stronger statement.

The basic model of program transformations and the theorem proof contain no notion of "nestedness". When the path transformations for several iterations of two nested loops are written in linear form, they will be indistinguishable from the transformations resulting from a single loop in which the calculations of the original outer loop are treated as alternate internal branches. This situation is diagrammed in figure 13. The two programs shown are equivalent, yet applying the this
theorem to the second indicates that no more than n+m iterations are required. Mapping this back onto the nested loop version, we face the implication that, given two nested loops, the inner loop need to be driven through a total of no more than n+m iterations, where the total is taken across all the repetitions of the outer loop.

Again, to avoid problems with counting loops, we might prefer to determine the minimum number of repetitions required to produce a testable subpath and adjust our heuristic to say that the sum of the number of iterations above that minimum (each time the loop is encountered) need not exceed m+n.

If we assume that a loop limit heuristic is incorporated into the Predecessor function, then the procedure Select-Path will halt. The worst-case behavior of Select-Path would now occur when paths were rejected only after being traced all the way to the start of the program. The time required by Select-Path would then be proportional to the number of paths from the program start to predicate being tested which satisfy the loop heuristic. Even for loopless programs this can result in exponential growth as the number of "IF" statements increases. While in practice Select-Path might often halt earlier either by finding a useful test path or by rejecting all paths due to some equality or assignment appearing late in the program, it seems likely that the worst case would be approached as more and more testing were done and the remaining blindness vectors became more and more difficult to detect.

There is room here for a great deal more work on inserting some "intelligence" into Select-Path. Some issues which may bear future research are:

1. How often in practice does the proposed limit on loop iterations fail? What common practices such as counting loops cause this to happen?
Figure 13: Transforming Nested Loops
2. Can part of the path selection process also be run forward from the start of the program to help ease the worst-case behavior? It seems likely that relationships among variables established in the earliest part of the code would be responsible for many of the undetectable error expressions which could cause the performance of Select-Path to degrade.

3. Can invariant expressions such as the loop invariants utilized in program proof techniques be detected and used to simplify the testing process? Such invariant expressions would be examples of error expressions which would be undetectable for all paths, so their early elimination from the total undetected space might help avoid sending Select-Path on fruitless searches.

So far the discussion has been focused on the selection of test paths for a single predicate. Normally a program will contain many predicates, and we may assume that all of these should be tested. The thought of applying a computationally expensive procedure such as Select-Path repeatedly to each predicate is not encouraging. Luckily this effort can be minimized by taking advantage of the knowledge gained when selecting paths for other predicates. The path rejection criterion given by Theorem 2 effectively screens for paths involving new, varied computations. Suppose that we have a set of short paths involving significantly different computations. We might expect longer paths formed by uniform extensions of the shorter paths would still involve significantly different computations and so would form a good initial set of test paths.

This type of behavior was seen in Chapter III when discussing the GCF program (figure 8). Recall that there were three predicates in that program. Once a sufficient set of test paths was generated for the first predicate, we found that sufficient sets could be generated for
the other two by simply extending the chosen test paths by the shortest path which would lead from the first predicate to each of the others. This was not an accident. In fact, the next theorem will establish this as a general property of linearly domained programs.

Theorem 5. Concatenation Rule

Given a set of subpaths \( K = \{ P_k \} \) and a subpath \( P_A \) satisfying the terms of Theorem 2, define \( K' = \{ P'_k \} \) and \( P'_A \) as the initial subpaths given by

\[
P'_k = P_k + P_B \quad P'_A = P_A + P_B.
\]

Suppose that \( \{ P'_k \} \) and \( P'_A \) are testable and no coincidental equalities are encountered in \( P_B \).

Then if testing is performed on \( \{ P'_k \} \), \( P'_A \) need not be tested.

Proof: Since the \( P_A \) and \( \{ P_k \} \) satisfy the terms of Theorem 2, any \( \tilde{e} \) which is in \( \text{Blind}(P_k) \) for all \( P_k \) is also in \( \text{Blind}(P_A) \). Assume by way of contradiction that there exists some \( \tilde{e} \) which is in \( \text{Blind}(P_k + P_B) \) for all \( k \) but not in \( \text{Blind}(P_A + P_B) \). So for all \( k \)

\[
\tilde{e} \in \text{span}[\text{null-space}(C_k^T C_B^T), \{ \bar{r}_j^k \} \cup \{ \bar{r}_j^B \}]
\]  

(12)

and

\[
\tilde{e} \notin \text{span}[\text{null-space}(C_A^T C_B^T), \{ \bar{r}_j^A \} \cup \{ \bar{r}_j^{AB} \}].
\]

(13)

If none of the equalities caused by the path extension are coincidental, then they must come directly from the predicates encountered in \( P_B \). Denote these as \( \{ \tilde{a}_j \} \). Since
\( \mathbb{F}_1^B \in \text{span}\{\text{null-space}(C_B^T)\}, \{\mathbb{q}_j\} \}

and

\( \mathbb{F}_1^{AB} \in \text{span}\{\text{null-space}(C_B^T)\}, \{\mathbb{q}_j\} \}

we can substitute the \( \{\mathbb{q}_j\} \) for the \( \{\mathbb{F}_1^B\} \) and \( \{\mathbb{F}_1^{AB}\} \) in equations (12) and (13). But from equation (13) and lemma 3 we know that the non-zero vector \( \mathbb{f} \) formed by taking \( \text{ortho}(\mathbb{e}, \mathbb{q}_j) \) for all the \( \mathbb{q}_j \) must satisfy

\[ \mathbb{f} \in \text{span}\{\text{null-space}(C_k^T C_B^T)\}, \{\mathbb{F}_j^k\} \] for all \( k \)

and

\[ \mathbb{f} \notin \text{span}\{\text{null-space}(C_A^T C_B^T)\}, \{\mathbb{F}_j^A\} \].

Now applying lemmas 1 and 2 to these equations.

\[ C_{A}^T \mathbb{f} \in \text{span}\{\text{null-space}(C_k^T)\}, \{\mathbb{F}_j^k\} \] for all \( k \)

and

\[ C_{A}^T \mathbb{f} \notin \text{span}\{\text{null-space}(C_A^T)\}, \{\mathbb{F}_j^A\} \].

So the vector \( C_{A}^T \mathbb{f} \) is in the blindness space for all \( P_k \) but not in the blindness space for \( P_A \). However we were given that no such vector can exist since the \( \{P_k\} \) and \( P_A \) were given as satisfying Theorem 2. Consequently the assumption that such an \( \mathbb{f} \) exists fails and the theorem is proven.

While this theorem suffers from the same weaknesses as the Iteration Limit Theorem, it provides another heuristic for path selection. Predicates at the beginning of the program should be tested
earliest and extensions of those test paths used to begin the tests of later predicates. The advantage of this approach is not only avoiding a duplication of effort, but also the fact that earlier predicates will usually have fewer paths leading to them so that searches will involve less effort.

Other strategies for selecting paths have been proposed. While most of these would not preserve the level of reliability maintained with Select-Path, they may involve considerably less effort.

One such strategy which has already been introduced is the classification scheme proposed by Howden[HOWDW75]. The theorems of this chapter permit some interesting observations regarding this strategy. Howden's classification distinguishes between boundary and interior paths of loops, where a boundary path is one in which a loop is entered but not iterated and an interior path is one which causes at least one iteration. Two paths are considered to fall into separate classes if

1. one is a boundary and the other an interior path of some loop;

2. both are boundary tests but follow different paths through the loop;

3. both are interior paths and follow different paths on their first iteration of the loop.

The strategy for path selection is to choose at least one path from each of the classes generated in this manner.
The first criterion effectively states that each loop will be tested using 1 and \( k \) iterations, where \( k \) is an arbitrary number greater than 1. The proof of Theorem 4 is of interest here. Two reasons can be advanced for the existence of an iteration limit. One possibility is that the set of blindness vectors approaches some constant set as the number of loop iterations is increased. The other possibility is that the set of blindnesses oscillates (with period \( \leq m+n+1 \)), so that after some minimal number of iterations there will exist an integer \( u \) so that the errors detectable using \( k \) iterations will also be detectable using \( k+u \) iterations. If such were the case, we would want to choose a number of iterations for the interior path tests so that the tests for that number of iterations are not equivalent to the tests for a single iteration.

The answer here is rather surprising. The internal path test should be conducted using two iterations of the loop, as long as testable subpaths can be found using two iterations. The argument for this is quite simple, and draws on Lemma 4. One possibility is that a path exists with two iterations which is not blind to some error which is undetectable using only one iteration. While there may be paths which would detect a larger number of the undetected errors, we are at least assured that this path yields some information. If, on the other hand, every two-iteration path is blind to all errors undetectable with one iteration, then Lemma 4 assures us that these undetected errors will not be detectable for any number of iterations, so the choice of a two-iteration path represents the least effort expended in a hopeless search.

This rule may strike some people as counter-intuitive. It is quite easy to construct examples for which this rule appears to break down. In fact, it does break down when coincidental equalities are allowed to occur. More likely, problems with feasibility, similar to the problems with counting loops described earlier, will arise. This is why the rule states that two iterations should be used "as long as testable subpaths
can be found using two iterations."

Other strategies have worked from the assumption that some upper limit should be placed on loop iterations. Theorem 4 provides a direct answer to this, indicating that such limits are at least partly defensible from a theoretical standpoint, and that the iteration limit should be \( m+n \).

This chapter has presented a discussion of the problem of selecting paths to test predicates in linearly domained programs. A procedure has been presented to reliably perform such a selection, but which is not guaranteed to halt. A limit has been established for the number of loop iterations which must be tested, enabling the selection procedure to halt, at the cost of losing some reliability due to problems with feasibility and coincidental equalities.

This selection procedure, even when guaranteed to halt, will be quite expensive computationally. Other strategies for selection such as Bowden's classification scheme or the branch coverage measure may be useful for generating an initial set of test paths, after which the procedure described here could be employed to track down the more stubborn expressions which are still undetectable. More work needs to be done on the possibilities of such combined strategies, as well as on the incorporation of more intelligent heuristics with the rather brute-force strategy presented in this chapter.
V. Errors in Program Computations

The last two chapters have dealt with domain errors caused by incorrect predicates. In this chapter the subject of errors due to incorrect computations will be discussed. In Chapter I we noted that erroneous computations can cause domain errors or computation errors. The term "hybrid error" was introduced to describe domain errors caused by incorrect computations (as opposed to "predicate errors" where domain errors are caused by incorrect predicates).

There is a certain temptation to underestimate the problems involved in testing for errors in computations. A straightforward approach would be to assume that the program function falls within some functional class, and to generate test points designed to distinguish between any two functions in that class. Techniques for choosing such sets of points for functional classes such as linear or polynomial functions are well known, and Howden has developed methods for choosing points for multinomials [HOWDW78].

The danger here lies in confusing the program function with the functions computed for individual subdomains. Such brute force point selection is useful only for relatively simple classes of functions. We have argued previously that the individual functions for each subdomain will tend to be simple, but when these subdomains are combined the resulting program function can be extremely complex. It follows then that such point selection strategies may by useful means of testing the function computed along a particular path, but they will not suffice for
testing the entire program. This is, of course, another way of expressing the path analysis testing strategy. A reliable means of selecting test points for a path may exist. The concern of this chapter is to discuss selecting the paths to be tested.

Intuitively one would feel that the Characterization Theorem of Chapter II should apply here also. The notions of assignment and equality blindness appear to generalize quite easily to discussions of errors in computations. If we have previously executed the assignment statement "X := A", it makes sense that the term "X - A" could be added to the right side of any assignment statement without being detected.

Assignment and equality blindness are not alone sufficient to describe undetectable errors in computations. When testing predicates, Chapter III indicated that only the statements preceding the predicate were important. When testing computations, a hybrid error will occur only if an incorrect computation affects some later predicate, and a computation error will occur only if an incorrect computation affects a variable in some output statement. Hence the statements following the computations being tested will also be of importance.

As in Chapter III, the initial goal is to characterize the set of computational errors which go undetected for a given path. The program construct to be tested will not be individual assignment statements, but blocks of "straight-line" assignments, that is, blocks of code with single entry and exit points containing only assignment statements. By testing blocks of assignments rather than individual ones, we allow for the possibility of errors in the computations of more than one variable at a time. A more subtle point is the possibility of missing computations. By treating such blocks of code as a transformation of the program environment, it is clear that variables not explicitly changed by an assignment statement undergo an identity transformation. This may not be correct. It is possible that the value of some variable
should be changed, but that the error consists of neglecting to make that change. For this reason testing should be done even on "empty" blocks of assignment statements such as the interval between lines 5 and 6 of figure 14.

Suppose that some transformation $C_i^*$ is replaced by $C_i^1$ where

$$C_i^1 = C_i + X$$  \hspace{1cm} (14)$$

$X$ is a matrix where each row denotes an error term added to the corresponding term of the environment. The top $m+1$ rows of $X$ must be zero, since error terms cannot be added to constants and input values.

1) INPUT A, B
2) $S := A$
3) $T := B$
4) $U := 0$
5) T1: WHILE $S \neq T$ DO
6) T2: IF $S > T$ THEN
7) $S := S - T$
8) ELSE
9) $U := S$
10) $S := T$
11) $T := U$
12) ENDIF
13) ENDWHILE
14) T3: IF $S = 1$ THEN
15) PRINT A," AND ", B," ARE RELATIVELY PRIME."
16) ELSE
19) ENDIF

Figure 14: Euclid's algorithm for GCF
Now consider a test subpath $P_A + (i) + P_B$. $P_A$ represents the subpath preceding $C_i$ and $P_B$ the subpath following $C_i$. Let $T$ be the predicate terminating $P_B$. A hybrid error occurs exactly when the substitution of $C_1^T$ causes a domain error. As in Chapter III, this is detectable for the chosen path when a shift occurs in the direction of the interpretation of $T$. Hence the substitution is undetectable when there exists some nonzero $h$ such that

$$T^T C_B C_1 C_A \bar{v}_0 = h T^T C_B C_1 C_A \bar{v}_0$$

for all $\bar{v}_0$ in the path domain. The set of matrices $X$ which satisfy this requirement is given in Theorem 6.

Theorem 6. Characterization of Hybrid Errors

A hybrid error $X$ is undetectable using the test subpath $P_A + (i) + P_B$ exactly when

$$X \in \text{span} \{X_A, X_B\}$$

where $\{X_A\}$ and $\{X_B\}$ are the sets of $(m+n+1)$ by $(m+n+1)$ matrices with their first $m+1$ rows being zero such that

1. $X_A$ has each row zero or in the span$[\text{Null}(C_A^T), \{F_A^B\}, C_{iB}^TC_B^T]$;
2. $X_B$ has each column zero or orthogonal to $C_{iB}^T$.

Proof: Define $T_B$ such that $T_B = C_B^T$. Substituting for $T$ and $C_i$ in equation (15) using this definition and equation (14) gives:

$$T_B^T C_1^T C_A \bar{v}_0 = h T_B (C_1 - X) C_A \bar{v}_0$$

$$T_B^T (X - (h-1)C_1^T) C_A \bar{v}_0 = 0$$
This last equation was solved in Chapter III. The solution set is

\[ x^T T_B \in \text{span}[\text{Blind}(P_A), (C_i)^T T_B] \]

(16)

This proof will proceed in two stages. First we will show that the transformations described by the theorem are solutions to (15). The second stage will consist of demonstrating that any solution to (15) will lie within the span of these transformations.

Several solutions to this equation are evident. The simplest would be \( \{X_B\} \), the set of matrices whose columns are either zero or are orthogonal to \( T_B \). In addition, if \( \bar{x}_i \) is the \( i \)th row of \( X \) and \( t_i \) the \( i \)th element of \( T_B \), we have

\[ x^T T_B = \sum t_i \bar{x}_i. \]

So another solution occurs when each of the \( \bar{x}_i \) is in the space described by the right side of equation (16). In other words, the set \( \{X_A\} \) is also a solution to (16).

It remains to be shown that any solution to (16) must lie within the span of \( \{X_A\} \) and \( \{X_B\} \). Consider any solution \( X \) to this equation. There must exist some set of scalar coefficients such that

\[ x^T T_B = \sum a_i \bar{u}_i \]

where the \( \bar{u}_i \) are a set of linearly independent vectors spanning the space described by the right side of equation (16). \( T_B \) must have at least one of its last \( n \) elements non-zero, otherwise (16) would have no solutions (meaning that all hybrid errors would go undetected, since a
hybrid error cannot be detected by a predicate involving only constants and input values). Let \( j \) be the index of such a non-zero element. Now define \( \{W_i\} \) as the set of matrices of the same dimension as \( X \) which are entirely zero, except for the \( j \)th row which is \( \bar{u}_i/t_j \). Then

\[
W_{1:B}^T = \bar{u}_i.
\]

By definition, each \( W_i \) is in the span of \( \{X_A\} \). What is more,

\[
(\sum \alpha_i W_i)^T B = \sum \alpha_i \bar{u}_i
\]

Denote the parenthesized term on the left as \( W \).

\[
X^T_B = W^T_B.
\]

\[
(X - W)^T B = 0
\]

The matrix \( (X - W) \) is therefore in \( \{X_B\} \). Now \( X = W + (X - W) \), and \( W \in \text{span}[\{X_A\}] \) and \( (X - W) \in \text{span}[\{X_B\}] \). Hence any solution \( X \) must be in the span of \( \{X_A\} \) and \( \{X_B\} \). Since we have already shown that any matrix in \( \{X_A\} \) or \( \{X_B\} \) is a solution, the theorem is proven.

The set \( \{X_A\} \) represents a set of errors which go undetected due to the subpath \( P_A \) which precedes the erroneous computation. The blindnesses involved should be sufficiently familiar by now. The set \( \{X_B\} \) represents a set of errors which go undetected because of the choice of subpath following the incorrect computation. A few examples should help to reveal why these are needed.

Suppose that the chosen test path for a program with variables \( X \), \( Y \), and \( Z \) ended with the predicate "IF \( X - Y > 0 \). We will assume for the sake of simplicity that none of these variables is used in the
evaluation of the other two. Ignoring the constants and inputs, this predicate would have a vector representation \((1, -1, 0)\) corresponding to \((X, Y, Z)\). A spanning set of perpendicular vectors would be \((0, 0, 1)\) and \((1, 1, 0)\). The first of these corresponds to the expression "\(Z\)". This means that any error term can be added to the evaluation of \(Z\) and not be detectable. This makes sense since the predicate does not involve \(Z\), and so a domain error cannot be caused. The second vector corresponds to "\(X + Y\)". This means that any error expression which is added to both \(X\) and \(Y\) will not be detected. Again this makes sense, since such an error term would be canceled when the predicate "\(IF X - Y > 0\)" is evaluated.

Theorem 6 deals with domain errors caused by incorrect computations. We would also like to test for computation errors. Such tests may not be quite as widely applicable as those for domain errors. The logical assumption in view of the previous theorems is that all program computations must be linear. This is far stronger than saying that computations affecting predicate interpretations must be linear. Nevertheless, if more complicated programs are to be sufficiently tested, this simple class of programs may provide some insight into how such testing may be accomplished.

Consider an output statement which prints a single variable. One way of viewing such a statement is that it selects the element of the environment corresponding to the output variable. In Chapter III the selection vectors \(\tilde{\mu}_j\) were introduced as the vectors which are entirely zero except for a one in the \(j\)th element. The product \(\tilde{\mu}_j \cdot \vec{v}\) gives the \(j\)th element of the environment vector \(\vec{v}\). Consequently we shall represent an output statement which prints the \(j\)th element of the environment as \(\tilde{\mu}_j\). If an output statement prints more than one variable, it will be represented by successive applications of the appropriate \(\tilde{\mu}\). The value which is printed by a subpath \(P_A \rightarrow (i) \rightarrow P_B\) is therefore \(\tilde{\mu}_j^T C_B C_i C_A \vec{v}_0\). As with hybrid errors, we will now replace \(C_i\) with an erroneous transformation \(C'_i\). The condition for detecting this
substitution is somewhat less stringent than for hybrid errors. Domain errors occur only when a border changes direction. A change in the scale or "size" of a border is undetectable. A computation error occurs when any alteration in the value of the printed variables occurs. The condition for undetectability is therefore

\[ \tilde{\mu}_j^T C_B C_i C_A \bar{V}_0 = \tilde{\mu}_j^T C_B C_i C_A \bar{V}_0 \]  

Comparing this to equation (15), a definite similarity can be seen. The major difference is the absence in (17) of the arbitrary multiplier \( h \). The set of undetectable computation errors representing the solution to (17) is presented in the next theorem.

**Theorem 7. Characterization of Computation Errors**

A computation error \( X \) is undetectable using the test subpath \( P_A + (i) + P_B \) exactly when

\[ X \in \text{span}\{\{X_A\},\{X_B\}\} \]

where \( \{X_A\} \) and \( \{X_B\} \) are the sets of \((m+n+1)\) by \((m+n+1)\) matrices with their first \( m+1 \) rows being zero such that

1. \( X_A \) has each row zero or in \( \text{Blind}(P_A) \)
2. \( X_B \) has each column zero or orthogonal to \( C_B^T \bar{C}_j \).

**Proof:** The proof of this theorem is virtually identical to that of Theorem 6. The only difference is that the lack of the multiplier \( h \) in (17) means that no self-blindness term is required in the definition of \( \{X_A\} \).
Since both of these theorems state that the undetected errors form a vector space, we can also formulate a path rejection criterion by comparing the set of undetected errors for a given path to the intersection of the undetectable error spaces for all previously tested paths. This idea should be familiar, since it represents a direct analogue to the path rejection criterion of Chapter III.

Theorems 6 and 7 describe the errors which go undetected in a block of assignment statements. These theorems differ according to whether the chosen test path ends with a predicate or with an output statement. This does not imply that testing should be conducted separately for hybrid and computation errors. An error which is detected using one of these theorems should not be considered undetectable when the other theorem applies. A plausible testing procedure here would begin by choosing a path leading up to the computations to be tested using Select-Path or any other method discussed in Chapter IV. This initial subpath would then be extended to various predicates and output statements. Since many such extensions may be possible, a number of test paths can be obtained in this manner. It may even be possible to reduce the space of undetected errors to \( \{X_A\} \) alone.

For an example of this, consider once more the GCF program listed in figure 14. For this example, we will examine some test paths for the block of computations represented by line 7. This block is represented by the following matrix:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

with the rows and columns corresponding to \( \{1, A, B, S, T, U\} \).
A simple initial path will be used. The subpath \( (T_1:T, T_2:T) \) leads to the block being tested. For this example we will consider various ways of completing this path in order to test for both hybrid and computation errors. The shortest possible subpath for testing this block is \( (T_1:T, T_2:T, T_3:?) \).

The first step is to find the set \( \{X_A\} \). When this example was used in Chapter III, the assignment blindness vectors were shown to be the columns of

\[
\begin{pmatrix}
0 & 0 & 0 \\
-1 & 0 & 0 \\
0 & -1 & 0 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \
\end{pmatrix}.
\]

Part of the definition of \( \{X_A\} \) is the set of matrices whose rows are either zero or else in the span of these vectors. A set of matrices for this part of \( X_A \) is

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\n\end{pmatrix}
\]

These matrices indicate that the expressions "S - A", "T - B", and "U" may be added to any of the variables or to any combination of variables and not be detected. This raises an interesting point. Suppose that
the correct computation should have "S - A" added to the variable T. The assignment statements for this block would be "S := S - T" and "T := T + S - A", but in what order should these statements be placed? Each affects the interpretation of the other. The answer, of course, is that both statements should be performed simultaneously, just as a transformation matrix performs each of its computations simultaneously. Most programming languages do not easily support such simultaneous computations. In this sense the matrix model is somewhat more powerful than conventional programming languages. This does not mean, however, that such error expressions are meaningless or illegal. Such conflicts are often encountered in practical programs, in problems as simple as exchanging the values of two variables. These conflicts can be resolved by the introduction of "dummy" variables to save the old values of one or more of the conflicting variables. In fact, lines 9 through 11 of the GCF program represent just such a conflict. The model presented is capable of telling when such temporary variables (used entirely within a block of computations) should have been created.

The remaining portion of \( \{X_A\} \) is the set of matrices whose rows are zero or equal to \( C_B^T T_1 \), which in this case is \((0, 0, 0, 1, -1, 0)^T\).

The remaining spanning matrices for \( \{X_A\} \) are

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

The next step is determining the set \( \{X_B\} \), the set of matrices whose columns are zero or orthogonal to \( C_B^T T_1 \) but whose first \( m+1 \) rows are zero. The vector \( C_B^T T_1 \) is simply \( T_1 \) \((0, 0, 0, 1, -1, 0)^T\). The set of vectors which qualify as possible columns of \( X_B \) are therefore \((0, 0, 0, 1, 1, 0)^T \) and \((0, 0, 0, 0, 0, 1)^T \). A set of spanning matrices for \( \{X_B\} \) is therefore
These matrices mean that any expression added to both S and T or to U will be undetected. The 24 total matrices presented here describe the entire set of undetectable errors. This does not mean that the undetected space has a dimension of 24. In fact this set of 24 matrices could be reduced to a set of 16 matrices spanning the same set.

Now suppose that the path (T1:T, T2:T, T1:F, T3:) is used. The first nine matrices presented above will also be in \{X_A\} for this path. Because the final predicate has changed, the last three matrices in \{X_A\} will be generated using \(C_{B,F_1,F_3}^{T,F,T} \):
The set \( \{X_B\} \) will be matrices with columns orthogonal to \( T_3 \) and with zeros in the first \( m+1 \) rows. These matrices will be

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

So this path cannot detect error terms added to \( T \) or \( U \), since the predicate involves only \( S \). Taking the intersection of the undetectable spaces for these two test paths gives a total undetected space spanned by the first nine matrices of \( \{X_A\} \) and by

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
Finally, if the path (T1:T, T2:T, T1:F, T3:T) is used to test for computation errors, the space \( \{X_A\} \) consists of the same first nine matrices. These matrices have remained for each path because they derive from the assignment blindness vectors which will not change unless a different initial path is taken. Since computation errors do not involve the \( C^T B_1^T \) term, no additional matrices are needed for \( \{X_A\} \).

The set \( \{X_B\} \) for this path will be matrices whose columns are orthogonal to \((0, 0, 0, 1, 0, 0)^T\) but zero in the first \( m+1 \) positions. This happens to give the same \( \{X_B\} \) set as for the previous path. Since so little is changed from the previous path, this additional test path has minimal effect on the total undetected space. Taking the intersection of the undetected space for this path with that of the two previous paths yields a total undetected space spanned by the nine matrices of \( \{X_A\} \) and by

\[
\begin{align*}
\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\end{align*}
\]
The number of matrices generated for this simple example appears to be disturbingly large. In Chapter III a limit was established for the number of paths required to sufficiently test a predicate. A similar limit for computations is needed as reassurance that the set of undetected error matrices will be bounded. Such a limit is easily derived. Intuitively an erroneous computation may affect the assignment of any one or more of n program variables. The error terms for any such assignment may involve the addition of any of n variables, m input values, and/or a constant. Hence we would expect no more than n(m+n+1) distinct errors to be possible.

More rigorously, an error term to a block of assignments is represented as a matrix X. The first m+1 rows of X must be zero. The set of possible error matrices therefore forms a vector space of dimension n(m+n+1). When zero paths have been tested, the set of undetectable errors is the space of all X. As each path is tested, it must reduce the dimensionality of this space by at least one. Any path not causing such a reduction need not be tested according to the path rejection criterion outlined above. Consequently the following theorem must hold:

Theorem 8. Minimal Set for Sufficient Testing of Computations

A minimal set of subpaths sufficient for testing a given computational block in a linearly domained program will contain at most n(m+n+1) subpaths, where m is the number of input values and n the number of program variables.
While this limit is substantially larger than that required for testing predicates, the problem is still polynomial-bounded. A major part of the effort involved in testing computations lies in choosing the initial paths leading up to the block to be tested. Since the criterion for this selection is identical to that for predicate testing, a significant degree of shared effort is possible. If test paths are first chosen for predicate testing, these same paths may be used for the blocks of assignment statements immediately preceding each predicate.

This chapter completes the analysis of testing for linear error expressions. Error expressions for hybrid and computation errors have been derived and a path rejection criterion proposed. The number of paths required for sufficient testing of computations has been shown to be no greater than $n(m+n+1)$. In the next chapter the restriction to linear functions will be relaxed, and the problems of testing for error expressions involving higher order functions will be discussed.
VI. Testing for Non-Linear Errors

Chapters III, IV, and V have concentrated on the detection of error terms which formed linear functions of the input values to a program. Reliably detecting such errors required the assumption that any program computations affecting the tested construct must also be linear. While there is no doubt that linearity permits considerable simplification, it would be easy to overstate its importance to the concepts presented in those earlier chapters. The crucial requirement is not linearity, but the ability to describe the set of error terms as a vector space.

In this chapter the linearity requirement will be relaxed. The goal is to indicate how the results presented in this research can be extended to a more general class of programs. Although we have argued that the class of linearly domained programs is sufficiently large to include many useful programs, there can be no denying that this class is restricted. If undetectable error spaces can be characterized for some non-linear functions, the theory presented here will have a much wider potential for application.

Intuitively, it seems that many of the ideas presented earlier should have a wider application. The notions of assignment, equality, and self blindness, for example, appear to be applicable to any class of programs. The critical question will be whether the sets of errors described by these concepts form a vector space. If not, we will be
unable to form a path rejection criterion as was done in Chapters III and V.

The first step in this extension is to modify the model of Chapter II to deal with non-linear functions. The primary change will be a switch from matrix notation to functional notation, with the composition of functions being represented directly rather than via matrix multiplication.

The environment will still be represented as a simple vector in $\mathbb{R}^{m+n+1}$. Computations in the program transform an old environment into a new one. Consequently, for a transformation $C$ we have

$$ C: \mathbb{R}^{m+n+1} \rightarrow \mathbb{R}^{m+n+1}. $$

Predicates operate on the current environment to yield a single number which will be compared with zero. Such predicates will be represented as

$$ T: \mathbb{R}^{m+n+1} \rightarrow \mathbb{R}. $$

The definitions of paths and subpaths remain unaltered. The restriction imposed by an equality predicate $q$ encountered along a certain path follows from the descriptions of predicates. An equality occurs when the equality predicate $q(\vec{v})$ is compared to zero and the result is

$$ q(\vec{v}) = 0. $$

If $\vec{v} = C(\vec{v}_0)$ then we can represent the restriction in terms of inputs and constants as
$q(C(v_0)) = (q \circ C)(v_0) = 0$

where the symbol "\(\circ\)" represents the composition of two functions (i.e. the application of the first function to the results of the second). We will define the equality restriction \(r\) to be the function \((q \circ C)\), so that

\[ r(v_0) = 0 \]

represents the restriction on the path domain due to an equality.

These changes will permit the basic model of Chapter II to be employed with non-linear functions. At this point it is appropriate to repeat something discussed in Chapter I. In general, determining the correctness of a program via testing or any other method is not possible. The question of whether or not a program computes the same function as its specification is generally undecidable. This fundamental limitation arises because of the undecidability of the problem of determining whether two arbitrary functions are identical. Some knowledge of the functions computed by a program is necessary if reliable testing procedures can be formulated.

There may be many different limitations which can be imposed on a program in order to obtain useful testing methods. This chapter will propose the use of vector spaces as one such limitation. In particular, the following assumptions regarding the program computations will be made:

1. All program computations fall within some class \(\{C_i\}\) which is closed under functional composition.

\[
[C_a \in \{C_i\} \text{ and } C_b \in \{C_i\}] \rightarrow [(C_a \circ C_b) \in \{C_i\}]\]
2. All predicates fall within some class \( \{T_i\} \) which is closed under composition with \( \{C_i\} \).

\[
[C \in \{C_i\} \text{ and } T \in \{T_i\}] \rightarrow [(T \circ C) \in \{T_i\}]
\]

3. \( \{T_i\} \) is a vector space over \( \mathbb{R} \) of dimension \( k \).

The first assumption states that the programmer has some knowledge of the class of functions which may be used to determine the values for his variables. This set must be closed under composition because it represents the actual functions computed (in terms of the inputs and constants) rather than simply the syntactic form of the expressions appearing in the text of the program. This assumption can be relaxed slightly when testing for predicate errors by only requiring those computations affecting later predicate interpretations to be in \( \{C_i\} \).

The second assumption states that the programmer also can specify the functional class representing the predicate interpretations of his program. The closure requirement means that this class must contain all interpretations, not merely the functional forms of the predicates before the variables are evaluated.

The third assumption is the key which will permit an analysis of errors in non-linear predicates. There are many possible classes of predicate functions satisfying this assumption. Besides the set of linear functions explored in the earlier chapters, the set of polynomials of degree \( p \) or the set of multinomials of degree \( p \) would fit this description. While rational functions and many other forms frequently encountered will not satisfy this requirement, these can be approximated via polynomial or multinomial expressions.
These assumptions replace the assumption in Chapter II that all predicate interpretations must be linear functions. The remaining assumptions from that chapter, that missing path errors do not occur, that the input space is continuous, the predicates are simple, and that adjacent domains compute different functions, will remain unchanged. A program satisfying all of these conditions will be termed a vector bounded program.

The properties of the composition $T \circ C$ are important to the discussion which follows. This expression, corresponding to $C^T_1 T$ in the linear notation of Chapter II, represents the interpretation of $T$ along some path computing $C$. This interpretation takes place by substitution of variables, so the following properties, true for any functions, must hold for $T$ and $C$.

1. The interpretation of the sum of two predicates is equivalent to the sum of their interpretations.

\[
(T_a + T_b) \circ C = T_a \circ C + T_b \circ C
\]

2. The interpretation of a scalar times a predicate is equivalent to the product of the scalar with the predicate interpretation.

\[
(\alpha T) \circ C = \alpha (T \circ C)
\]

For example, if the predicate $T_a$ corresponds to "IF $X>0$" and $T_b$ corresponds to "IF $Y>0$" where previous assignments have established that $X=A$ and $Y=A+B$, we could rewrite the two predicates as "IF $A>0$" and "IF $A+B>0$", respectively. The sum of these predicate interpretations is the predicate "IF $2A+B>0$". The same function is obtained by taking the interpretation of the sum of the predicates. This sum is "IF $X+Y>0$".
Its interpretation is "IF 2*A+B>0". Scalar multiplication works similarly. While the operation of adding two predicates may not seem to make much sense, this is in fact the operation involved in adding an error term to a correct predicate.

If \( \{T_i\} \) is a vector space and \( \{T_i\} \) contains both an incorrect predicate \( T' \) and its correct form \( T \), then the term \( (T'-T) \) must also be in \( \{T_i\} \) since vector spaces are closed under addition and scalar multiplication. The expression \( (T'-T) \) represents the error term which was added to \( T \); so the set of possible error terms is also a vector space.

Define the function \( e \) as an error term for which

\[
T' = T + ae \quad a \neq 0.
\]

Then the interpretation of the incorrect predicate \( T' \) is given by

\[
T'(\vec{v}) = T(\vec{v}) + ae(\vec{v})
\]

As with linear error terms, testing will only be done using testable subpaths, feasible subpaths ending with a predicate which is not implied by the earlier predicates. Assuming a reliable testing strategy, we will again note that the error \( e \) is undetectable if and only if there exists a non-zero scalar \( h \) such that

\[
T(\vec{v}) = hT'(\vec{v})
\]

for all \( \vec{v} \) in the path domain. Define \( \text{Null}(C) \) as the set

\[
\{e: e \in \{T_i\} \text{ and } \forall \vec{v}, e \circ C(\vec{v}) = 0\}.
\]

Then the set of functions satisfying this condition is specified in the next theorem.

Theorem 9. Characterization Theorem for General Predicates
Let $P$ be a testable subpath in a vector bounded program. Let $C$ be the function computed along that path, $T'$ be the final predicate in $P$, and $\{r_i\}$ be the set of equality restrictions on the domain of $P$. Then an error $e$ in $T'$ will be undetectable if and only if

$$e \in \text{span}([\text{Null}(C), \{r_i\}, T'])$$

Proof: Assume that $e$ is undetectable. Then

$$T'(\vec{v}) + ae(\vec{v}) = hT'(\vec{v})$$

$$ae(\vec{v}) + (1-h)T'(\vec{v}) = 0$$

Since $\vec{v} = C(\vec{v}_0)$,

$$aeC(\vec{v}_0) + (1-h)T'c(\vec{v}_0) = 0$$

(18)

Consider first the case $h=1$. Then for all $\vec{v}_0$ in the path domain,

$$e^cC(\vec{v}_0) = 0.$$  

Two possibilities exist. The first is that $e^cC(\vec{v})$ is zero for any $\vec{v}$; the second is that a $\vec{v}$ exists for which $e^cC(\vec{v})$ is not zero, but $e^cC(\vec{v}_0)$ is zero for any $\vec{v}_0$ in the path domain. Consider each of these in turn:

1. $e^cC = \text{zero function}$.

The solution set to this equation has already been defined as $\text{Null}(C)$. Clearly $\text{Null}(C)$ is a subset of $\{T_i\}$. If it is also closed under addition and under multiplication by a scalar, then
Null(C) is a vector subspace of \{T_i\}. This is easily proven. Given \(x\) and \(y\) both in Null(C),

\[(x + y) \circ C = x \circ C + y \circ C = 0 + 0 = 0,\]

so \((x + y)\) is also in Null(C) and Null(C) is closed under addition. Similarly for any \(x\) in Null(C) and any scalar \(\alpha\),

\[(\alpha x) \circ C = \alpha(x \circ C) = \alpha(0) = 0,\]

so \(\alpha x\) is also in Null(C) and Null(C) is closed under scalar multiplication. Consequently Null(C) is a vector subspace of \{T_i\}.

2. \(e \circ C\) is zero function, \(e \circ C(\overline{v}_0) = 0\) for all \(\overline{v}_0\) in the path domain.

The function \((e \circ C)\) is zero for all \(\overline{v}_0\) in the path domain, but non-zero for some \(\overline{v}_0\) outside that domain. The set of functions in \{T_i\} which fit this description has been designated as \{r_i\}. Consequently, this condition reduces to

\[e \circ C \in \text{span}\{[r_i]\}.\]

(19)

Once again the relevant question is whether the set of error functions satisfying this equation comprises a vector subspace of \{T_i\}. Consider any two functions \(x\) and \(y\) from \{T_i\} which satisfy this equation.

\[(x \circ C)(\overline{v}_0) = 0 \quad (y \circ C)(\overline{v}_0) = 0\]

\[(x \circ C)(\overline{v}_0) + (y \circ C)(\overline{v}_0) = 0\]

\[(x \circ C + y \circ C)(\overline{v}_0) = (x + y) \circ C(\overline{v}_0) = 0,\]
so the solution set to (19) is closed under addition. Now let \( a \) be any scalar.

\[
\alpha[(x^\circ C)(\overline{v}_0)] = 0
\]

\[
(a x^\circ C)(\overline{v}_0) = 0,
\]

so the solution set is also closed under scalar multiplication.

Combining the solution sets of both cases gives

\[
e \in \text{span}[\text{Null}(C), \{x: x^\circ C \in \text{span}\{r_i\}\}] \quad (20)
\]

as the total solution set when \( h=1 \). This equation can be simplified in order to reach a form closer to that in the theorem. Suppose that \( e \) is a function in the set \( \{x: x^\circ C \in \text{span}\{r_i\}\} \). Then there must exist some set of coefficients such that

\[
x^\circ C = \sum \alpha_i r_i.
\]

Now since the \( r_i \) are wholly in terms of the input values and constants, \( r_i^\circ C = r_i \) and so

\[
(\sum \alpha_i r_i)^\circ C = \sum \alpha_i r_i = x^\circ C.
\]

It follows that the function \( (x - \sum \alpha_i r_i) \) must be in \( \text{Null}(C) \).

Consequently,

\[
\{x: x^\circ C \in \text{span}\{r_i\}\} \subseteq \text{span}[\text{Null}(C), \{r_i\}].
\]

Combining this expression with equation (20) gives

\[
e \in \text{span}[\text{Null}(C), \{r_i\}].
\]
Now consider equation (18) when \( h \neq 1 \). Then

\[
[ae + (1-h)T'] \cdot C(\vec{v}_0) = 0.
\]

The quantity in square brackets represents an arbitrary combination of two vectors in \( \{T_i\} \), one known and one unknown. Since this equation has the same functional form as the case \( h = 1 \), the solution is

\[
e \in \text{span}[\text{Null}(C), \{r_i\}, T'],
\]

and the theorem is proven.

Just as with linearly domained programs, a path rejection criterion can be formulated by noting that if the intersection of the undetectable spaces of all previously tested paths is a subset of the undetectable space for some proposed test path, then that proposed path is of no value for testing. Each path selected will therefore reduce the dimension of the total space of undetectable errors by at least one. Since this space has a finite initial dimension, the next theorem follows directly.

Theorem 10. Minimal Sufficient Set for General Predicates

A minimal set of subpaths sufficient for testing a given predicate in a vector bounded program will contain at most \( k \) subpaths, where \( k \) is the dimension of \( \{T_i\} \).

For any class of predicate functions satisfying the requirements of this theorem, a finite bound will exist on the number of paths required
for sufficient predicate testing. Exactly what this bound will be depends entirely on the class of functions chosen.

As an example of testing predicates for non-linear error expressions, consider the program in figure 15 which is designed to print out an annual loan summary given the amount of the principal \( P \), the rate of interest \( R \), and the amount of the annual payment \( A \). It is evident from statement 4 that this program will involve multinomial computations. The loop predicate \( T_1 \) is intended to force iteration until the balance for the current year is less than the amount of the next payment. This program is, in fact, incorrect since this predicate should read "WHILE \( B \times (1+R) \geq A \)" , the error term in this case being "\( B \times R \)".

1) INPUT \( P \), \( R \), \( A \);
2) \( B := P \);
3) \( T_1 : \) DO WHILE \( B \geq A \);
4) \( B := B \times (1 + R) \);
5) \( B := B - A \);
6) PRINT "BALANCE: ", \( B \);
7) END;
8) PRINT "FINAL PAYMENT IS", \( B \);
9) END;

Figure 15: Loan History
We will arbitrarily choose the set of predicate interpretations to be functions of the form

\[
T(P,R,A,B) = \alpha_0 + \alpha_1 A + \alpha_2 P + \alpha_3 B + \alpha_4 R + \alpha_5 AP \\
+ \alpha_6 AB + \alpha_7 AR + \alpha_8 PB + \alpha_9 PR + \alpha_{10} BR
\] (21)

Begin by examining the shortest possible test path for T1, (T1:?). The only computation along this path is the assignment \( B := P \). The set of undetectable errors due to this assignment is not simply \( B - P \), as it would have been in a linearly domained program. Instead the null space of this transformation is the set of multinomials of the same form as \( T(P,R,A,B) \) above which are identically zero when \( P \) is substituted for \( B \). Performing this substitution yields

\[
0 = \alpha_0 + \alpha_1 A + \alpha_2 P + \alpha_3 P + \alpha_4 R + \alpha_5 AP \\
+ \alpha_6 AP + \alpha_7 AR + \alpha_8 P^2 + \alpha_9 PR + \alpha_{10} PR.
\]

Grouping related terms together,

\[
0 = \alpha_0 + \alpha_1 A + (\alpha_2 + \alpha_3)P + \alpha_4 R + (\alpha_5 + \alpha_6)AP \\
+ \alpha_7 AR + \alpha_8 P^2 + (\alpha_9 + \alpha_{10})PR.
\]

Each term of this last equation must be identically zero, so we have a linear set of equations for the \( \alpha_i \) coefficients. It is interesting that finding blindness spaces for non-linear functions will require the solution of a set of linear equations, not a set of non-linear ones. The set of equations to be solved is

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
\alpha_0 \\
\alpha_1 \\
\alpha_2 \\
\alpha_3 \\
\alpha_4 \\
\alpha_5 \\
\alpha_6 \\
\alpha_7 \\
\alpha_8 \\
\alpha_9 \\
\alpha_{10}
\end{bmatrix}
= \begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
\]
The solution to this set of equations is spanned by the columns of

\[
\begin{bmatrix}
0 & 0 & 0 \\
-1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 1
\end{bmatrix}
\]

Adding the self blindness vector into this set gives the total undetectable space as

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\]

Substituting these vectors into equation (21), the set of undetectable errors for this path can be expressed as

\[a(B - P) + b(AB - AP) + c(BR - PR) + d(B - A)\]

for any real numbers a, b, c, and d. The first three terms arise from the assignment statement "B := P" while the last term represents the self blindness vector.

Next consider the path (T1:T2, T1:?). After executing this subpath the symbolic evaluation of B is "P + PR - A". Substituting for B in equation (21) yields the following condition for undetectability due to assignment blindness:
0 = a_0 + a_1 A + a_2 P + a_3 PR - a_3A + a_4 R
+ a_5 AP + a_6 A^2 + a_6 PR + a_7 AP + a_8 P^2
+ a_8 P^2 R - a_8 PA + a_9 PR + a_{10} PR + a_{10} P R^2 - a_{10} R A

Collecting the related terms,

0 = a_0 + (a_1 - a_3) A + (a_2 + a_3) P + a_4 R + (a_5 + a_6 - a_8) AP
+ (a_7 - a_{10}) AR + (a_3 + a_9 + a_{10}) PR + a_6 APR + a_6 A^2
+ a_8 P^2 + a_8 P^2 R + a_{10} P R^2.

Setting the coefficients of each term to zero gives the following system of equations:

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
\[\bar{a} = \bar{0}\]

This system is considerably larger than the one for the first test path. Not all of the equations are independent (in particular, note the eighth and ninth rows), but the greater variety of terms used in the computations for this path result in a more stringent condition for undetectability. The total blindness space for this path is
The first vector is the unique solution to the above system of equations, and the second one is the self blindness vector.

The intersection of this space with that of the first path is simply the self blindness vector, so these two test paths form a sufficient test set for the predicate $T_1$.

Theorem 9 demonstrates that the results of Chapter III for predicate errors can be extended quite naturally to more general functions. The next question will be whether such an extension can be formed for errors in computations. Unfortunately such extension does not appear possible. Consider an error $X$ in some computational block $C_i$, and a path which computes a sequence of computations $C_A, C_i, C_B$ such that

$$v = C_B \circ C_i \circ C_A(v_0)$$

represents the total path transformation. Then replacing $C_i$ by $C_i' = C_i - X$, the condition for undetectability for computation errors is

$$T \circ C_B \circ C_i' \circ C_A(v_0) = T \circ C_B \circ (C_i' - X) \circ C_A(v_0).$$

Now define $\bar{v}_A = C_i' \circ C_A(v_0)$, $\bar{x} = -X \circ C_A(v_0)$, and $T_B = T \circ C_B$. Then this condition reduces to

$$T_B(\bar{v}_A) = T_B(\bar{v}_A + \bar{x})$$
Now if the set of errors which go undetected is to be a vector space, the set must be closed under addition. So for any two errors $x_1$ and $x_2$, if we compute $\bar{x}_1$ and $\bar{x}_2$ in the same manner we require that

$$T_B(\bar{v}_A + \bar{x}) - T_B(\bar{v}_A) = 0$$

and

$$T_B(\bar{v}_A + \bar{x}_1) - T_B(\bar{v}_A) = 0$$

only if

$$T_B(\bar{v}_A + \bar{x}_1 + \bar{x}_2) - T_B(\bar{v}_A) = 0.$$

The only obvious situation for which this requirement appears appropriate is when $T_B(\bar{v}_A + \bar{x}) - T_B(\bar{v}_A)$ is a linear function of $\bar{x}.$ While this by no means constitutes a proof that vector space measures are inappropriate for non-linear computation errors, it does serve to indicate that such measures will not serve for most simple and useful classes of functions.

Nonetheless, this requirement is an interesting one. It seems to indicate that vector space measures would apply if the interpretation of the error term were linear, regardless of the functional class of $T_B$ or of the computations forming $\bar{x}.$ This is in contrast to our previous assumptions that all computations must fall in the functional class of interest.
As a final extension of the basic vector space model, we will generalize this idea of testing only for errors forming perturbations of a chosen functional class. Such a relaxation is not without its price. Since the functional class of both the correct and the incorrect program is not taken into consideration, we can no longer make any claim to absolute reliability. Instead of testing to eliminate all possible substitutions for some predicate, we will simply be testing to eliminate all error expressions of some class. A scale of reliability is reached in this manner as we test for differing classes of perturbations. Testing for linear perturbations of a predicate, for example, yields less confidence than testing for polynomial perturbations of degree $k$, which in turn gives less confidence than testing for polynomial perturbations of degree $k+1$, and so on.

Beginning with predicate errors, consider the interpretation of an erroneous predicate. The error will be undetectable when for some $h$, 

$$hT' \circ C(\overline{v}_0) = T \circ C(\overline{v}_0).$$

Define the error term of this interpretation as 

$$e \circ C = T \circ C - T' \circ C.$$ 

There may be many $e$'s which satisfy this definition. Our interest will be to solve for all those in some given functional class which satisfy the unsolvability condition. Note that this definition is considerably different from the requirement used earlier that $e = T' - T$. Whether this new requirement is stronger or weaker will depend on the particular choice of functions for $T$ and $C$.

Substituting into the undetectability requirement, the error is undetectable when 

$$e \circ C(\overline{v}_0) = (h-1)T' \circ C(\overline{v}_0).$$
Instead of requiring the predicate interpretations to fall within a specified class as was done previously, we will here specify that the set of possible perturbations $e \circ C$ must be a vector space. Now for all cases where $h \neq 1$, the right and left hand sides of this equation differ only by a scalar multiplication. Since vector spaces are closed under scalar multiplication, this means that the predicate interpretation lies within the space of error interpretations. Such a case is exactly what was discussed under Theorem 9, the characterization theorem for general predicates. Since we have postulated that this theorem is not applicable, this case will not be considered further. Setting $h = 1$ then gives

$$e \circ C(\bar{v}_0) = 0.$$ 

Now this equation has already been solved in various guises throughout this research. The only difference here is that certain assignment and equality blindness expressions may not fall within the desired functional class for $e$. Designate this class as $E^*$ and let $\text{Restrict}(E^*, \{f_i\})$ be a function returning the set of functions $f_i$ which are in $E$. Then the following theorem holds:

**Theorem 11. Characterization of Predicate Perturbations**

The set of error perturbations in some vector space $E$ which cannot be detected in the final predicate of a path whose total transformation is $C$ is the set spanned by

$$\text{Restrict}(E, \text{Null}(C) \cup \{r_i\}).$$
Actually this theorem provides the justification for some of the operations performed in the example presented earlier in this chapter. Some of the terms encountered in that example were second order multinomials, but we had stated that we would only test for errors represented by first order multinomials. The higher order terms were therefore assumed to be zero. This assumption was not actually justified at the time. The problem encountered in that example was that polynomial and multinomial computations are not generally closed under functional composition. Such closure was assumed in the derivation of Theorem 9. Theorem 11 now says that this closure requirement can be relaxed at the cost of losing the absolute reliability. The undetectable error expressions we found were indeed all the first order multinomial expressions which could be added to the test predicate without being detected. However there also existed undetectable higher order multinomial expressions which could legitimately result from the interpretation of a first order error term.

It might appear that all reliability is lost using this perturbational requirement, but this is not true. Consider for example a program with one input value, A, and one variable, X. Suppose for some subpath leading to a predicate being tested we have \( X = A^2 \). Testing for linear perturbations of the predicate gives the interesting answer that no such linear blindness vectors exist! This is entirely correct. There is no linear expression in X and A which is identically zero for this path. Hence any linear error in the predicate would be detected by a reliable path testing strategy. This is true despite the fact that "\( X - A^2 \)" is an undetectable polynomial error term.

The level of confidence which is implied by perturbational testing is therefore proportional to the probability that an arbitrary error will fall within the class of perturbations we have selected. If non-linear polynomial errors are relatively rare, testing for linear perturbations may be sufficient.
Turning once again to errors in computations, consider the change in the interpretation of some predicate (or output variable for computation errors) due to an error term $X$. Define this error term as

$$T \circ C_B \circ X \circ C_A(\vec{v}_0) = T \circ C_B \circ C_i \circ C_A(\vec{v}_0) - T \circ C_B \circ C_i \circ C_A(\vec{v}_0).$$

As with predicate perturbations, we will argue that the condition for undetectability will be

$$T \circ C_B \circ X \circ C_A(\vec{v}_0) \neq 0.$$

The arguments given previously regarding the possibility of finding a vector space solution to this equation still apply. Assume therefore that this expression is a linear function of the input values and variables. Then there must exist a set of linear transformations $C_B$, $C_A$ and $X$ and a linear function $\bar{T}$ such that $\bar{T}^T C_B X C_A$ represents the perturbation. Then the last equation becomes

$$\bar{T}^T C_B X C_A \vec{v}_0 = 0.$$

This equation was solved in Chapter V. Allowing $\bar{T}$ to be either a predicate or a selection function representing an output statement, the following must hold:

**Theorem 12.** Characterization of Computation Perturbations

The set of linear perturbations caused by an incorrect computation $C_i$ which cannot be detected using a test subpath $P_A^+(i) + P_B$ ending with a predicate or output statement $\bar{T}$ is the set spanned by $\{X_A\}$ and $\{X_B\}$ where $\{X_A\}$ and $\{X_B\}$ are the sets of $(m+n+1)$ by $(m+n+1)$ matrices with their first $m+1$ rows entirely zero and
1. $X_A$ has each row zero or in the span of Null-space($C_A^T$), \{ $A_i^T$ \};

2. $X_B$ has each column zero or orthogonal to $C_B^T$.

The ability to perform perturbational testing may be more important from a practical viewpoint than the ability to test for polynomial and multinomial functions. One problem with these higher order functions is that such computations seldom satisfy the closure requirement. Consequently the perturbation theorems must be invoked if the vector space of functions is to have a finite dimension.

Even when closure is satisfied, the dimension of the resulting vector spaces may be prohibitively large. Consider a program where the predicate interpretations are multinomials of degree two, with no more than two variables in each term. Then the dimension of the space of possible predicate errors is $(m+n)^2+1$. So even a simple program where $m+n=10$ would require setting up and solving a set of equations involving 101 unknowns for predicate errors and as many as 1001 unknowns for hybrid and computation errors. The relatively modest cost of 11 unknowns for linear predicate perturbations seems more reasonable, and the 101 unknowns for computation perturbations, while still large, is a vast improvement. It would seem likely, therefore, that reliable testing for polynomial or multinomial errors will be too expensive in most cases. Testing for linear perturbations may constitute a reasonable alternative.
VII. Summary and Conclusions

Reliable methods of choosing paths for path analysis testing strategies are not easily devised. The undecidability of testing in general implies that such reliable methods are possible only when additional knowledge is available regarding the class of functions computed by the program. Of course, there are many different types of knowledge which could be brought to bear on this problem, with some of these types being useful in testing and some being of no use. This research has sought to demonstrate that one useful type of information is the knowledge that the class of possible errors forms a vector space.

Two properties of vector spaces are of particular importance to the testing process. First, given any two members of some vector space, any linear combination of those two elements will be a member of the same vector space. In particular, suppose that certain program constructs such as predicates or computations could be described using a vector space. Consider two elements of that space, the first being the (erroneous) construct as it appears in the program, the second being the proper form of that construct. Then the difference between these two functions, which we would normally identify as the error term, must be a vector in the same space. Consequently the set of all possible error terms is a vector space.

The second property of vector spaces which is of particular importance is the ability to describe most useful vector spaces using a
finite number of characteristic vectors, such that any vector in the space can be formed as a sum of these characteristic vectors. Hence a vector space, which if not empty will contain an infinite number of functions, may be completely described using finite measures. This says a great deal regarding the "richness" of a vector space model of computer programs. Consider a strategy for testing based on the assumption that the program constructs must be one of a finite number of options. The expressive power of most programming languages is so much more powerful than this that any such strategy would be suspect. Vector spaces represent a way of classifying infinite sets of functions which can be described using finite measures.

Path analysis testing strategies involve a two step process starting with the selection of a set of test paths followed by the choice of test points for those paths. No matter how reliable a method is employed in this second step, certain possible errors will be undetectable. In general, there will be an infinite number of such undetectable error terms. We would like to know that, when the total set of possible errors is known to be a vector space, then those errors which are undetectable for a given test path will form a vector subspace of that larger set. If the set of undetectable errors for a path form a vector space, then a finite characterization can be given of all errors "missed" by testing with that path.

Such is the case when testing program predicates. If the set of possible predicate interpretations is a vector space, the set of possible predicate errors which may have escaped detection using a given test path is a vector space. The characteristic undetectable errors (vectors) are easily computed. If \( \bar{V} \) is the program environment (i.e., the set of all input values, variables, and constants used by the program), then for any variable \( X \) which has been assigned a value \( f(\bar{V}) \) by the computations along the chosen test path, the expression \( X - f(\bar{V}) \) must be zero along that path. Consequently, the expression \( X - f(\bar{V}) \) could be added to a predicate at the end of that path without
causing a detectable error. The addition could only be detected by the use of a different test path. This behavior was named "assignment blindness".

Similarly, predicates encountered along the test path might impose some equality restrictions of the form $f(\nabla) = 0$ on the set of points which cause that test path to be executed. Then the function $f(\nabla)$ could be added to a predicate at the end of a test path and the addition not be detected. This is called "equality blindness".

Finally, any predicate may be multiplied by a constant without causing an error. The condition "IF $X > 1$" is indistinguishable from "IF $2X > 2$". Hence the function represented by the predicate itself can be added to the predicate without being detected. This behavior is called "self blindness".

The vector space of undetectable predicate errors for a given test path is simply the space formed by all possible combinations of the functions described by assignment, equality, and self blindness.

Since the set of predicate errors which escape detection can be determined for any path, it is possible to determine whether a proposed test path will be useful for testing. If the set of errors which have gone undetected for all previously tested paths is also undetectable for a proposed test path, then the proposed path can give no additional information. In other words, the previously tested paths will have left some total undetected space. A proposed test path is useful only if it reduces the size of that space, where the size is measured by the number of characteristic vectors required to describe the space.

An interesting implication of this criterion for rejecting proposed test paths is a limit on the number of paths required to sufficiently
test program predicates. Each test path used must reduce the size of
the total undetected error space by at least one, but this space can not
have originally been any larger than the space of all possible errors.
Hence if the program predicates form a vector space of dimension k, no
more than k paths are necessary to test any predicate. If the predicate
interpretations are linear functions, then no more m+n paths are
required, where m is the number of input values and n is the number of
program variables.

Similar measures can be derived for errors in computations when
these computations are linear functions. The ideas of assignment,
equality, and self blindness apply here also. An additional requirement
exists which requires that those variables whose computations are
altered by the error must be employed in a later predicate or output
statement. A criterion for evaluating test paths can be derived in the
same manner as for predicate errors. The limit imposed by this
criterion on the number of paths required to test a block of
computations is n(m+n+1) paths.

The path evaluation criteria for predicate and computation errors
can be applied to a wider range of programs at the expense of absolute
reliability. Instead of requiring all predicates and/or computations in
the program to be members of some chosen class of functions, it is
possible to test only for error expressions whose interpretations are of
a chosen functional form. The path evaluation criteria then serve to
indicate when all possible perturbations of some form have been checked.
No guarantee can be given that an error expression of some other
functional form is not present except the assertion of the programmer
that such an error were unlikely. Of course, in the original derivation
of the path evaluation criteria, such an assertion forms the only
guarantee that the correct form of the program involves only predicates
and computations from the chosen class of functions.
An algorithm for selecting test paths can be formulated when testing linear predicates. The time required by this algorithm is proportional to the number of paths in the program being tested. If, as is often the case, this number is infinite, then the algorithm may not halt. Halting can be guaranteed by imposing a limit on the number of loop iterations. Such limits have been assumed by other researchers, purely on the basis of expediency. The model presented here provides partial theoretical support for an assumption that no more than $m+n$ iterations of any loop need to be considered. If a linear predicate error exists which is detectable using some test path involving more than $m+n$ iterations of some loop, then some path involving no more than $m+n$ iterations of that loop will exist which also detects that error.

In simpler terms, errors which are detectable using long paths can also be detected using short paths, where a path is considered long when it involves more than $m+n$ repetitions of some loop. The major limitation on this result is that the shorter path is not guaranteed to be feasible, so it may be useless for testing. Hence this result provides only partial theoretical support for an iteration limit.

The vector space model presented in this research is not without its failings. The rather high cost required for reliable testing of non-linear functions would seem to restrict all but the most ambitious testers to linear forms. Indeed, many common functional forms cannot easily be handled by vector spaces. In particular, ordinary division presents difficulties. The model is also weak with respect to the range of data types to which it may be applied. Floating point numbers are handled easily, integers not so well, but character strings and various compound data structures are not covered at all by this model. Of course, character strings could be treated as an array of integers, but arrays tend to be sore points for most path analysis strategies. One reason for this difficulty may be clearer due to the path limits described in this research: arrays tend to drastically increase the number of program variables.
Perhaps the most fundamental problem inherent in the vector space model of program errors is its limitation to arithmetic level operators. Vector spaces proved useful in grouping errors which could be added together in any fashion to generate another undetectable error. If a large program is designed as a hierarchy of abstractions, this model will only apply at the lowest levels. In fact, the very ideas of error terms, addition of errors, etc. may have no relevance at the higher levels.

Nevertheless, when dealing at low levels of abstraction this model has several advantages. As a practical tool, it features a stronger theoretical reliability than has been seen in other research. The blindness vectors are easily obtained from the type of information available in most symbolic execution systems. The path rejection criterion can be evaluated using polynomial time algorithms.

As a theoretical tool, this model may provide a standard for evaluating the reliability of proposed testing algorithms. The limits on the number of paths required for testing guarantee that a finite set will suffice, providing a much better bound than the total path coverage which is often cited. The vector space model was also used as an evaluation tool in Chapter III, where mutation testing was investigated, and in Chapter IV, where Howden's path classifications were discussed [HOWDW75]. With regard to mutation testing, the underlying assumption that the detection of simple errors guarantees detection of more complex errors (which can be considered to be combinations of the simpler ones) cannot be supported. In fact, the inverse actually holds. Failure to detect simple errors implies that combinations of those errors will not be detected. Consequently, mutation testing, although it may still in most cases be a valuable testing method, cannot claim to be a theoretically reliable technique. Howden's path classification scheme is also not reliable, but appears to provide a reasonable starting approximation to a sufficient set of test paths since it tends to choose
widely differing computations for its various test paths, an important consideration for eliminating assignment blindness.

Finally, this model can provide theoretical justification for many of the assumptions commonly accepted as a matter of expediency, such as limiting loop iterations or the ability of a finite number of test paths to suffice for testing.

Many issues related to this model remain unexplored. What type of error expressions occur in practice, and what type are generated by the undetectability criteria? How large is the total undetected space in practice, and how quickly can it be reduced by proper choice of additional paths? What types of heuristics can be advanced to guide path selection, whether selection is done entirely by the programmer or done automatically? Can the set of invariant expressions for a program be approximated, in order to judge when path selection can end?

In addition, more fundamental considerations will require future research. Some means of manipulating arrays without treating each element separately must be found, taking advantage of any homogeneity of operations over the array. The entire issue of reliable testing of modules involving high levels of abstraction must be addressed. Without progress in this area in particular, the testing process will be incompatible with the program design process.
Appendix. Algorithms for Manipulating Spanning Vectors

Some familiarity with computational methods in linear algebra is presumed of anyone pursuing this line of research. However, the algorithms employed in the examples in Chapters III through VI involve manipulation of singular matrices. The techniques for dealing with singular matrices are not as widely known as the simpler techniques used with nonsingular matrices. Consequently, this appendix will explain the algorithms employed to find and to manipulate the sets of spanning vectors described in those examples. This discussion is intended for those readers who are familiar with linear algebra and the more common algorithms from numerical linear algebra and who are interested in the example computations or in implementation issues associated with the vector space model with which this research is concerned.

I. Notation

In order to simplify the algorithms which follow, certain notational conveniences will be introduced. Most of these will describe operations applied uniformly over entire vectors or matrices, so that such operations may be viewed as a single concept rather than introduce the necessary loops, etc. required to perform the operation on an element by element basis. For the most part, the notation introduced here is based on that of the language APL.
For any matrix C, the notation "C(I,J)" denotes the element in the Ith row and Jth column of C. "C(I,:)" denotes the vector formed by the Ith row of C. "C(:,J)" represents the vector formed by the Jth column of the matrix C. Assignment statements will be permitted to pass single numbers, vectors, or matrices. The data type and dimensions of the result will be determined by the data type and dimensions of the right hand side of the assignment statement.

The dimensions of a matrix will be given by "Rows(C)" and "Cols(C)". Vectors will be treated as degenerate matrices where Cols(C)=1. The notation "A*B" will denote the concatenation of matrices A and B, the matrix composed of the columns of A followed by the columns of B. If A and B are vectors, for example, then the result of this operation would be a matrix having two columns, one for each vector.

Normal arithmetic operators will distribute over vectors and matrices. Hence the notation "A+B" will denote the matrix or vector whose every element consists of the sum of the corresponding elements of A and B. The operator "*" will denote the outer product of two vectors, so the assignment "C := A*B" where A and B are vectors yields a matrix such that C(I,J)=A(I)*B(J). Another special operator which will be used is a Sort function. Sort takes any vector, returning a vector of the same length. The returned vector contains a set of integer indices describing the sorted order of the original vector, so that for any vector X, X(Sort(X)) will be the elements of X in ascending order. For example, if X were the vector (2. 0. 1.5), then Sort(X) would be (2 3 1) since X(2 3 1) is (0. 1.5 2.).
II. Diagonalizing a Matrix

The various forms of Gaussian elimination employed to reduce a nonsingular matrix to triangular or diagonal form are well known. The algorithm presented here is a variant of this method which reduces a possibly singular matrix to a diagonal form where for each row there exists some column with a nonzero element such that all other rows are zero in that column. Reduction is accomplished by addition of rows and multiplication of rows by scalars. Partial pivoting is employed to improve numerical accuracy.

1) procedure DIAG(A);
2) define INDEX as a vector of length Rows(A);
3) INDEX(I) := I \begin{align} & \forall 1 \leq I \leq \text{Rows}(A) ; \\
4) & B := A ; \\
5) & ROW := 1 ; \\
6) & COL := 1 ; \\
7) & \text{do while } ROW \leq \text{Rows}(A) \text{ and } COL \leq \text{Cols}(A) ; \\
8) & \text{Let } R \text{ be the element of } B(*)^\text{COL} \text{ with the largest absolute value and } J \text{ be the first row in which it occurs.} \\
9) & \text{if } \text{abs}(B(J,COL)) \geq .0001 \text{ then} \\
10) & \text{/* pivot */} \\
11) & R := B(J,*)/B(J,COL); \\
12) & B(J,*):=B(ROW,*); \\
13) & K := INDEX(J); \\
14) & INDEX(J) := INDEX(ROW); \\
15) & INDEX(ROW) := K; \\
16) & \text{/* zero out this column */} \\
17) & B := B - B(*)^\text{COL} \cdot R; \\
18) & B(ROW,*):=R; \\
19) & \text{/* this row is done */} \\
20) & ROW := ROW + 1 ; \\
21) & \text{endif;}
III. Solving Sets of Simultaneous Equations.

The diagonalization procedure just given would suffice to solve a set of linear equations having exactly one solution. However, when the matrix of coefficients is singular, an infinite number of solutions are possible. In general these solutions will form an affine space, a space where each element can be expressed in the form \( \bar{x} + \bar{y} \) where \( \bar{x} \) is a constant "offset" and \( \bar{y} \) is any element of a vector space. The vector space in this instance will be the null space of the coefficient matrix. The offset vector can be any solution to the system, since all other solutions can be reached by addition of vectors from the null space. The algorithm which follows solves the system \( A\bar{x} = \bar{b} \), returning a matrix whose first column is the offset vector and whose remaining vectors are the spanning vectors of the null space. The algorithm begins by diagonalizing the matrix. Those variables which are on the resulting diagonals are used to construct an offset vector. Since any solution will suffice, the variables not on the diagonal are assumed to be zero. Then the equations are reduced to the form \( a_{ii}x_i = b_i \) (ignoring the effects of pivoting) and the solution is trivial.

The rest of the algorithm is concerned with determining the null space of the coefficient matrix. For purposes of this explanation, we will ignore the effects of pivoting. Then the problem of finding the
null space could be represented as

\[
\begin{bmatrix}
\bar{x} \\
\bar{y}
\end{bmatrix} = 0
\]

where \( \bar{x} \) represents the variables on the diagonal, \( \bar{y} \) the remaining variables, and the quantity in brackets represents the diagonalized matrix. We will refer to equations in this form as closed form equations.

The goal of this algorithm is to find a different matrix such that there exists a vector \( \bar{p} \) satisfying

\[
\begin{bmatrix}
\bar{p} \\
\bar{y}
\end{bmatrix} = \begin{bmatrix}
\bar{x}
\end{bmatrix}
\]

exactly when \( \bar{x} \) and \( \bar{y} \) satisfy the previous set of closed form equations. The columns of this new matrix will be the spanning vectors of the null space being sought. Equations in this form will be said to be in a spanning form.

Since the original matrix has been diagonalized, there are exactly as many elements of \( \bar{x} \) as there are rows in the diagonalized matrix. The closed form equations can therefore be written in the form

\[
\begin{bmatrix}
\bar{0} \\
\bar{y}
\end{bmatrix} = -\bar{x}
\]

without altering the matrix. Now those columns which are multiplied by zeros can be eliminated without altering the solution set of the whole system. Negating the matrix at the same time gives a system of the form
\[ y = x \]

which is closer to the desired spanning form. The spanning form includes a set of arbitrary multipliers contained in the vector \( \bar{p} \). These are introduced by adding a new set of equations, new rows to the matrix. Append an identity matrix to the top of the matrix in the last equations, and let \( \bar{p} \) be appended to the top of \( \bar{x} \) to describe the right hand side of these new equations:

\[ \bar{y} = \begin{bmatrix} \bar{p} \\ \bar{x} \end{bmatrix} \]

Since the equations involving \( \bar{p} \) are the rows of an identity matrix, \( \bar{p} = \bar{y} \) and \( \bar{p} \) and \( \bar{y} \) can be exchanged in the above system to give

\[ \bar{p} = \begin{bmatrix} \bar{y} \\ \bar{x} \end{bmatrix} \]

Swapping the rows will yield the desired spanning form. Since the solution set has been unchanged in each of these steps, the columns of this matrix will span the null space of the original closed form.

The algorithm to perform this solution follows

1) procedure SOLVE(A,B);

\( \quad \star \) solve the system of equations \( Ax = B \) \( \star \)

2) \( C := A,B; \)

3) \( C := DIAG(C); \)

4) \( \text{BOUND} := \text{Rows}(C); \)

5) \( \text{FREE} := \text{Cols}(C) - \text{BOUND} + 1; \)

6) define INDEX as a vector of length \( \text{Rows}(C) \);
7) define OFFSET as a vector of length FREE+BOUND;
8) J := 1;
   /* compute offset vector */
9) do while J < BOUND;
10) INDEX(J) := index of first non-zero element of C(J,*);
11) OFFSET(INDEX(J)) := C(J,Cols(C));
12) J := J + 1;
13) end do;
14) Remove last column from C;
   /* compute null space */
15) Remove INDEX(J)th column from C, \forall J, 1 \leq J \leq BOUND
16) if FREE = 0 then
   /* system is nonsingular */
17) C := OFFSET;
18) else
   /* system is singular */
19) I := FREE by FREE identity matrix;
20) C := (I , (-C)^T)^T;
21) I := all integers from 1 to BOUND+FREE not in INDEX,
      arranged in ascending order;
22) INDEX(J+FREE) := INDEX(J), \forall J, 1 \leq J \leq BOUND;
23) INDEX(J) := I(J), \forall J, 1 \leq J \leq FREE;
24) C := OFFSET , C(Sort(INDEX),*);
25) end if;
26) return(C);
27) end SOLVE;

IV. Determining the Closed Form Equations.

The SOLVE procedure transforms a description of an affine space from closed form to spanning form. Frequently the inverse of this transformation is required, finding the closed form for a given set of spanning vectors. For the purposes of this work, vector spaces will
suffice rather than the more general affine spaces. The algorithm to determine the closed form is somewhat simpler than the SOLVE procedure.

The starting point for this algorithm is the spanning form of a vector space. The diagonalization procedure is applied to the columns of the matrix of spanning vectors to give

\[
\begin{bmatrix}
\bar{p} \\
\bar{x} \\
\bar{y}
\end{bmatrix}
\]

where \(\bar{x}\) represents those variables assigned by diagonalized rows and \(\bar{y}\) the remaining variables. Then the top set of equations are simply \(p_i = x_i\), so an equivalent form for this system is

\[
\begin{bmatrix}
\bar{p} \\
\bar{x} \\
\bar{y}
\end{bmatrix}
\]

Now the elements of \(\bar{p}\) are simply arbitrary multipliers whose values are of no interest. Consequently those equations which solve for \(\bar{p}\) can be dropped to yield

\[
\begin{bmatrix}
\bar{x} \\
\bar{y}
\end{bmatrix}
\]

Then to reach the closed form we need simply move the \(\bar{y}\) over to the left side of the system. Each equation is in the form \(f(\bar{x}) = y_i\), and the desired form is \(f(\bar{x}) - y_i = 0\). Multiply the matrix by \(-1\) and append an identity matrix to the right hand side to give the final form

\[
\begin{bmatrix}
\bar{x}
\end{bmatrix}
\]
This represents a closed form of the original vector space.

This algorithm is summarized in the following procedure.

1) procedure CLOSEDFORM(A);
2) \( C := (\text{DIAG}(A^T))^T; \)
3) \( \text{HIGH} := \text{Rows}(C); \)
4) \( \text{WIDE} := \text{Cols}(C); \)
5) if \( \text{HIGH} = \text{WIDE} \) then
   /* all possible vectors are in this space */
6) \( C := \text{empty matrix (0 by 0)}; \)
7) else
   /* not all possible vectors are in this space */
8) define \( \text{INDEX} \) as a vector of length \( \text{HIGH}; \)
9) \( \text{INDEX}(J) := \text{index of first nonzero element in } C(*,J), \)
    \( \forall J, 1 \leq J \leq \text{Cols}(C); \)
   /* drop top rows and append identity matrix */
10) drop row \( C(\text{INDEX}(J), *) \) from \( C \), \( \forall J, 1 \leq J \leq \text{HIGH}; \)
11) \( I := (\text{HIGH-WIDE}) \text{ by (HIGH-WIDE)} \text{ identity matrix}; \)
12) \( C := I, -C; \)
13) \( I := \text{vector of all positive integers } \leq \text{HIGH} \text{ not contained in } INDEX, \text{sorted into ascending order}; \)
14) \( \text{INDEX}(J+\text{WIDE-HIGH}) := \text{INDEX}(J), \quad \forall J, 1 \leq J \leq \text{HIGH}; \)
15) \( \text{INDEX}(J) := I(J), \quad \forall J, 1 \leq J \leq \text{WIDE-HIGH}; \)
16) \( C := C(\text{Sort(INDEX), *}); \)
17) end if
18) return(C);
19) end CLOSEDFORM;
V. Union and Intersection of Sets of Spanning Vectors.

Most of the preceeding chapters refer explicitly to the intersection of two or more vector spaces. In addition, there is an implicit requirement that the union of vector spaces be formed in order to combine the spaces generated by the various forms of blindness. In both cases, it is assumed that the vector spaces are represented by a set of spanning vectors.

Finding the union is a trivial operation. Since the union consists of any vectors in the span of either set of characteristic vectors, the union is characterized by combining both sets of spanning vectors into one set. The columns of the resulting matrix will span the union, but in general will not form a linearly independent set. Linear independence can be easily achieved by applying the DIAG procedure.

Finding the intersection is somewhat more difficult. A vector is in the intersection only if it is in both spaces. The key here is to think in terms of the closed form representation of a vector space. This form presents a set of equations which must be satisfied by any vector in the corresponding space. Consequently, a vector is in the intersection exactly when it satisfies both closed forms. If the closed form equations for the two spaces are combined to form a single system, the SOLVE procedure can be used to find the spanning vectors corresponding to the intersection.

1) procedure INTERSECT(A,B);
2) A := CLOSEDFORM(A);
3) B := CLOSEDFORM(B);
4) C := (A^T, B^T)^T;
5) C := SOLVE(C,U);
6) drop first column of C;
/* (offset vector will be zero) */

7) return(C);

8) end INTERSECT;
BIBLIOGRAPHY


137


