THE UNIVERSITY OF ALBERTA

BLINDNESS-BASED TESTING FOR DOMAIN ERRORS

BY

© STEPHEN H. MA

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1988

ISBN 0-315-45642-6

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Stephen H. Ma

TITLE OF THESIS: Blindness-based Testing for Domain Errors

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1988

[illegible]

Permanent Address:

16511-98 St.

Edmonton, Alberta

Canada T5X 5L1

Date: June 1, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled Blindness-based Testing for Domain Errors submitted by Stephen H. Ma in partial fulfilment of the requirements for the degree of Master of Science.

Lee Hotutto
(Supervisor)

[illegible]

Jia-Huai You

J. E. Lewis

Date June 23, 1988

ABSTRACT

Program testing involves the execution of a program over a set of test data followed by the evaluation of the program over the set. In general, the problem of finding a finite set of paths to conduct testing is known to be unsolvable. For a certain class of programs which restrict the operations to linear functions, it is possible to characterize the set of errors which escape detection for a given path. This type of error is termed *blindness*. The blindness concept can be formulated as a path selection strategy to uncover *domain errors*, one class of principal errors in computer programs, which occur when specific input data follow the wrong paths due to errors in the control flow of the program. Two path selection algorithms based on the blindness concept are proposed to expose domain errors, one with a complexity of $O(n^2)$ and the other with a complexity of $O(n)$, where n is the number of program variables. The process of selecting test paths is analyzed, heuristics are summarized, and a stopping criterion is suggested.

Acknowledgements

I am indebted to my supervisor Dr. L.J. White for his guidance and encouragement. His advice is invaluable.

I am grateful to the members of my examining committee, Dr. P. Rudnicki, Dr. J.H. You, and Dr. J.E. Lewis for their helpful comments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Glenford J. Myers, an authority on software engineering, once wrote:

> Given the tremendous amount of time and money spent in testing computer programs, and given the serious consequences of program errors in computing systems, one would expect that software testing would be a highly developed and widely understood skill. Actually, this is not the case at all. Less seems to be known about software testing than about any other aspect of software development [9].

Program testing is the execution of a program over a set of test data in order to identify desired program functions or to reveal program errors. A *testing strategy* is a method to construct test data. A testing strategy is *reliable* for a program if any set of test data constructed by this strategy indicates errors whenever the program is incorrect.

The correctness of the program is determined by evaluating the output of test data. We assume the existence of an "oracle" which can always determine whether the output corresponding to test data is correct or not. The concept of a test oracle was originated by Howden [7]. Test oracles can take on different forms: tables, hand calculated values, simulated results, formulae, program specifications, etc.

The difficulty of program testing arises from the fact that there is no reliable testing strategy for programs in general. Howden [4] proves that the problem amounts to the determination of the equivalence of arbitrary primitive recursive functions. According to computation theory, this is unsolvable.

Although the problem of finding a general testing strategy for all programs is unsolvable, for certain classes of programs, however, it is possible to construct a finite test data set to uncover certain classes of errors. There are strategies, which involve the construction of tests to expose specific classes of errors, for example, "algebraic errors" [5] and "domain errors" [11] for certain classes of programs.

Howden [4] has classified two types of errors for computer programs: *domain errors* and *computation errors*. A *domain error* occurs when an error in the control flow causes some input to follow the wrong path. A *computation error* occurs when an error in some assignment causes some input to compute the wrong function although the correct path is followed.

Path analysis testing, a class of strategies based on the structure of the program to be tested, consists of two operations:

1. select a set of paths for testing,
2. select input data to execute the chosen paths.

Currently, the second operation appears to be better understood. The *Domain Testing Strategy* [11], for example, generates test points on or near the boundaries of an input domain, which can reliably detect if a domain error has occurred as one or more boundaries have shifted. A data selection method is *reliable* if test data generated by the method in the path domain causes incorrect output whenever there is an error in the program. The progress which has been made in the selection of input data allows us to concentrate on the first operation: selecting test paths. We assume the existence of test data selection methods which can utilize selected paths to explicitly expose errors.

Zeil [13] identified three types of blindness errors which occur in path testing: *assignment blindness*, *equality blindness* and *self-blindness*. Blindness errors are algebraic expressions which can be added to the correct expression of a program statement (assignment or predicate) without being detected by a given path through that statement. Namely, the execution of the program along the given path will not distinguish if that statement contains an erroneous expression or not. Blindness errors, however, can be exposed by the selection of different paths. Zeil applied the blindness concept to formulate a path selection strategy, which we refer to as *blindness-based testing*.

This study will focus on blindness-based testing for domain errors. Its theory and justification will be discussed; its applicability as a testing strategy will be explored. Problems encountered in theory and practice will be identified and analyzed; procedures to deal with these problems will be suggested and evaluated. The goal is to formulate a reliable path selection strategy for certain classes of programs which can effectively expose domain errors.

Chapter two gives a brief introduction to the results of Zeil. Chapter three presents a mathematical model for a simple programming environment. Based on this model, we study the blindness theory and its justification. Chapter four summarizes the blindness concept and proposes two blindness-based path selection algorithms. The complexity and effectiveness of these two algorithms are discussed. Chapter five deals with a problem encountered in blindness testing: *invariant expressions*, which are combinations of blindness errors and cannot be eliminated by the selection of different paths. Chapter six discusses test data selection and related issues. Chapter seven is a summary of this study. The strength of blindness-based testing as well as its limitations is evaluated, and future study is outlined.

CHAPTER TWO

ZEIL'S STUDY

2.1 Background

Before we introduce the results of Zeil's study, we will first provide some background of program testing. For convenience, we distinguish two types of variables in the program. Variables appearing in the input statements are termed *input variables*; other variables whose values are established by the right hand side of assignment statements are termed *program variables*.

A program can be represented by a control flow graph G = (N, A), where G is a directed graph, N is a finite set of *nodes*, and A is a set of *arcs* between nodes. Each node in N represents a statement in the program. An *entrance node* in the graph has no predecessors; an *exit node* in the graph has no successors. There is only one entrance node in the graph; there can be multiple exit nodes in the graph. The control flow graph defines the paths within a program. A *subpath* is a finite sequence of nodes in N $(n_1, n_2, \ldots, n_p)$ such that for all i, $1 <= i < p$, n_{i+1} is a successor of n_i and $(n_i, n_{i+1}) \in A$. An *initial subpath* is a subpath whose first node is the entrance node. A *complete path* is an initial subpath whose last node is an exit node. The word *path* is used to denote both complete paths and subpaths.

Assignment statements and predicates (conditional branch statements) are two major components of most programs. Assignment statements assign new values to program variables. Predicates determine the program control flow. When a predicate is evaluated along a certain path, a *predicate interpretation* is produced by replacing each program variable appearing in the predicate with its symbolic value calculated in terms of input variables along that path.

Predicates control the program flow and partition the input domain into a set of subdomains. Each subdomain corresponds to a particular path where input data from the subdomain will cause that path to be executed. Whenever an error occurs in the control flow, it will usually cause a shift in the subdomain boundary. As a result, some input data will follow a wrong path, which is a typical domain error.

Domain errors may occur due to an error in a predicate, or an error in an assignment statement which subsequently affects the interpretation of a later predicate. The former is termed a *predicate error*, and the latter is called an *assignment error*. However, an assignment error can cause a domain error, a computation error, or both. When a required predicate is missing in a program, it is called a *missing path error* [4]. No path testing methods can systematically detect this type of error.

2.2 Zeil's Results

As we mentioned in the first chapter, Zeil identified three types of blindness errors in path testing: assignment blindness, equality blindness and self-blindness. Blindness errors are associated with assignments and predicates. For instance, after the assignment "$y_i := f(X)$", the expression "$y_i - f(X)$" can be added to a later statement without detection. This behavior is called *assignment blindness* (Fig. 1). Similarly, if an equality restriction in terms of input variables has been imposed along a path, this expression can be added to a later statement without detection. This behavior is called *equality blindness* (Fig. 1). Finally, a testing predicate can never be distinguished from its multiples. This can hardly be considered an error. However, this expression, called *self-blindness*, can be combined with assignment or equality blindness to yield unexpected erroneous expressions (Fig. 1).

Assignment Blindness

```
Correct Code              Incorrect Code

 Y = X                     Y = X
  ...                       ...
 IF 2Y + 3 > 0             IF X + Y + 3 > 0
  ...                       ...
```

Equality Blindness

```
Correct Code              Incorrect Code

 IF X = 1 THEN             IF X = 1 THEN
  ..                        ...
 IF Y > 0 THEN             IF X + Y - 1 > 0 THEN
  ...                       ...
```

Self-blindness

```
Correct Code              Incorrect Code

 Y = X                     Y = X
  ...                       ...
 Y > 1 THEN                X + Y > 2 THEN
  ...                       ...
```

Figure 1 Assignment Blindness, Equality Blindness, and Self-Blindness

Blindnesses are algebraic expressions which can be added to the correct expression of a program statement without being detected for a given path. If a statement contains a blindness expression which is indistinguishable in the program execution for any possible path through that statement, this blindness will not cause any error. In this case, the program with the statement containing the blindness expression is equivalent to the original program. When we refer to blindness in the following discussion, we only concentrate on blindness which will actually cause errors.

The principle of the blindness concept is that a single path cannot guarantee the correctness of the program. This is due to the fact that a single path cannot distinguish blindness errors arising from assignments and predicates along the path. In order to exclude blindness errors, multiple paths are needed. The blindness concept can be applied to guide the selection of paths.

Zeil developed a vector space model for a certain class of programs termed *linearly domained programs* (they will be defined later in this section). In the vector space model, variables, assignments and predicates are defined in an m+n+1 dimensional space, where m is the number of input variables and n the number of program variables. An infinite number of blindness errors can be characterized by a finite number of vectors. A *blindness space* is a collection of blindness errors which can be added to a program statement without being detected. The blindness space concept is applied to a path and a program statement respectively. A blindness space for a path leading to a program statement consists of n assignment blindness vectors, no more than m equality blindness vectors (equality constraints along the path), and a self-blindness vector (for predicates only). A blindness space for a program statement initially consists of m+n+1 vectors. After a path is selected, the new blindness space for the program statement is the intersection of the previous blindness space and the blindness space for the selected path.

A set of test paths is considered *sufficient* for a program statement if the failure to detect an error using a reliable data selection method along these test paths implies that the error cannot be guaranteed detectable for any path through the statement.

Zeil's path selection strategy consists of the following three criteria:

1. Path selection criterion.

If a set of subpaths ending at some program statement has been previously tested, then a new subpath ending at the same statement will be selected if it can reduce the dimension of the blindness space for the program statement.

2. Stopping criterion.

A set of subpaths ending at some program statement is sufficient for testing the statement if the blindness space for the program statement is a null space (for an assignment) or contains a self-blindness vector only (for a predicate).

3. Minimal sufficient test set criterion.

A minimal set of subpaths sufficient for testing given program statements will contain at most m+n+1 subpaths (for a predicate) or n(m+n+1) subpaths (for a block of assignments).

The last criterion is based on the fact that the initial blindness space for a predicate has a dimension of m+n+1; since each selected path will reduce the dimension of the blindness space by at least one, a sufficient test set will consist of no more than m+n+1 paths. The initial blindness space for a block of assignments has a dimension of n(m+n+1) since the initial blindness space for each assignment has a dimension of m+n+1, and a block has up to n assignments. Therefore, a sufficient test set for a block of assignments will consist of no more than n(m+n+1) paths.

Zeil applied this strategy to deal with predicate errors and assignment errors respectively.

Programs under investigation in this study are termed *linearly domained programs* which are subject to the following assumptions:

1. missing path errors do not occur;
2. predicates are simple, not combined with logical operators (AND, OR);
3. the input space is continuous;
4. adjacent domains compute different functions; and
5. predicates and assignments are linear expressions in terms of input variables, and if they are incorrect, the correct ones are also linear when expressed in terms of input variables.

The first assumption is inherent to path analysis testing methods. Assumptions two to four are actually not limitations of this model, they are just for convenience to simplify the forms of predicates and the selection of test data. The last assumption assures that program computations and predicate interpretations are closed under vector addition and scalar multiplication.

2.3 Research Problems

Zeil's study has laid the foundation for blindness-based testing. The following problems are identified in order to continue the study in this area:

1. unification of testing assignments and predicates for domain errors, and
2. investigation of the stopping criterion and related issues.

As Zeil has applied the testing strategy to assignments and predicates separately, a logical conjecture is whether assignment testing and predicate testing can be combined. A related problem is the complexity of Zeil's approach in assignment testing, which is $O(n^2)$, where n is the number of program variables. This method appears too impractical to be implemented. In both theory and practice, we need a simplified scheme for assignment testing which can be easily combined with predicate testing to form a uniform testing strategy for domain errors.

The stopping criterion suggested by Zeil is inadequate to cover practical situations in testing. Sometimes there are vectors in the blindness space, in addition to self-blindness indicated by Zeil, which cannot be eliminated by the selection of different paths. This situation is first reported by Sahay in reference [10]. These vectors are termed *invariant expressions* in this paper (they will be defined in Chapter 5). If invariant expressions cannot be dealt with effectively, the path selection process may contradict our goal -- the search for a finite set of test paths may become endless.

The objective of this study is to develop a blindness-based testing method for linearly domained programs. This method will select a sufficient set of paths to test assignments and predicates for domain errors, and can be implemented practically.

CHAPTER THREE

A MATHEMATICAL MODEL FOR A SIMPLE PROGRAMMING ENVIRONMENT

In this chapter we will define a simple programming environment and introduce a vector space model mainly based on Zeil's study. We will give rigorous definitions for this model, and use this model to explore blindness-based testing. Some of Zeil's analytical methods have been adapted in the investigation. The analysis of predicate errors follows a similar approach to that of Zeil. The analysis of assignment errors comprises a different approach, and different conclusions are derived.

3.1 A Simple Programming Environment

Let us consider a simple programming environment. There are four types of statements: input, output, assignment and predicate. These are major components of most computer programs. As in structured programming, we deal with two types of predicate statements: selection, commonly "IF-THEN-ELSE"; and iteration, commonly "WHILE" loop. There are two types of variables in the program: input and program variables. For the simplicity of discussion, we assume that the values of input variables remain unchanged throughout the program. This environment is linearly domained, i.e., assignments and predicates are restricted to linear expressions in terms of input variables.

3.2 A Vector Space Model

Linearly domained programs can be modeled by linear vector spaces using matrices to represent vector operations. The vector space model is defined in an m+n+1 dimensional space, where m and n are the numbers of input variables and program variables respectively. In the following, we will define program state, predicate, assignment, path, and predicate interpretations in this model.

3.2.1 Definitions

In the following discussion, a row vector is represented by $(a_0,\ldots,a_{m+n})$, and a column vector is represented by $(a_0,\ldots,a_{m+n})^t$, where t denotes a transpose. A *program state* $\bar{V} = (1,x_1,\ldots,x_m,y_1,\ldots,y_n)^t$ is an m+n+1 dimensional column vector, where 1 is a constant, $x_1,\ldots,x_m$ are input variables and $y_1,\ldots,y_n$ program variables. The program state vector represents the values of all variables at any point during the program execution. The constant "1" is just for notational convenience. The *initial state* vector is $\bar{V}_0 = (1,x_1,\ldots,x_m,0,\ldots,0)^t$, where $x_1,\ldots,x_m$ assume values from the input statement, and all program variables are initialized to zero before the first assignment. The program state is updated after every assignment. $\bar{V}_i$ denotes the program state after the i-th assignment. Since the values of input variables remain unchanged, program variables contained in $\bar{V}_i$ are always in terms of constants and input variables.

A *predicate* $a_0 + a_1x_1 + \ldots + a_mx_m + a_{m+1}y_1 + \ldots + a_{m+n}y_n$ ROP 0 is defined as an m+n+1 dimensional row vector $\bar{P} = (a_0,a_1,\ldots,a_m,a_{m+1},\ldots,a_{m+n})$, where a_0 is a constant, $a_1,\ldots,a_{m+n}$ are the coefficients of input and program variables, and ROP is a relational operator. All predicates in the program are labeled sequentially, e.g., $\bar{P}_k$ denotes the k-th predicate in the program.

An *assignment* $y_j = a_0 + a_1x_1 + \ldots + a_mx_m + a_{m+1}y_1 + \ldots + a_{m+n}y_n$

is defined as an (m+n+1)x(m+n+1) matrix

$$A = \begin{array}{c|cccccccc|c} & 1 & x_1 & \cdots & x_m & y_1 & \cdots & y_n & & \\ & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 & & 1 \\ & 0 & 1 & \cdots & 0 & 0 & \cdots & 0 & & x_1 \\ & & & \cdots & & & \cdots & & & \\ & 0 & 0 & \cdots & 1 & 0 & \cdots & 0 & & x_m \\ & 0 & 0 & \cdots & 0 & 1 & \cdots & 0 & & y_1 \\ & & & \cdots & & & \cdots & & & \\ & a_0 & a_1 & \cdots & a_m & a_{m+1} & \cdots & a_{m+n} & & y_j \\ & & & \cdots & & & \cdots & & & \\ & 0 & 0 & \cdots & 0 & 0 & \cdots & 1 & & y_n \end{array}$$

A is an identity matrix with the (m+j+1)-th row (corresponding to the j-th program variable) replaced by a constant and coefficients of input and program variables assigned to the program variable y_j. All assignments in the program are labeled sequentially, i.e., A_i denotes the i-th assignment in the program.

Since the values of input variables remain unchanged, it is obvious that the first m+1 rows of the assignment matrix always form an identity matrix followed by zeros from the (m+2)-th to (m+n+1)-th columns. The (m+j+1)-th row, which corresponds to the j-th program variable, consists of coefficients of the assignment; the rest of the rows from m+2 to m+n+1 are all zeros except for the elements on the diagonal line being one.

In this model, a path $A_p = A_{i_k} A_{i_{k-1}} \ldots A_{i_1}$ is represented by the multiplication of a finite sequence of assignment matrices, and the result is also a matrix. The path representation implies the Boolean values assumed by predicates along the path, even though the predicates do not appear in the path representation. A *feasible path* requires that all predicate expressions with implied Boolean values along the path be satisfiable. If not, the path is considered *infeasible*. Since the programming environment is linearly domained, the feasibility of any path can be decided by linear programming methods, say, the simplex method.

We have defined the program state and the initial state vector at the beginning, and now we can provide a dynamic view of the program state vector after each assignment. A new program state after the i-th assignment is $\overline{V}_i = A_p\overline{V}_0 = A_i \dots A_1\overline{V}_0$, namely, $\overline{V}_i$ is the multiplication of A_p which denotes an initial subpath (whose last assignment is A_i) and the initial state $\overline{V}_0$.

The *predicate interpretation* is the result of a scalar product, $\overline{P}_j\overline{V}_i = \overline{P}_jA_p\overline{V}_0 = \overline{P}_jA_i \dots A_1\overline{V}_0$, where $\overline{P}_j$ denotes the predicate, $\overline{V}_i$ the program state, and A_p the initial subpath leading to the predicate. The scalar product is then compared with zero to decide the Boolean value of the predicate.

3.2.2 An Illustration

In the following we use a sample program to illustrate the model (Fig. 2). We label assignment, predicate and output statements in sequence. We also list corresponding assignment matrices and predicate vectors.

```
          READ $x_1$, $x_2$
$A_1$         $y_1 = x_1$
$A_2$         $y_2 = x_2$
$P_1$         WHILE $y_1 - y_2 > 0$
$A_3$             $y_1 = y_1 - y_2$
$A_4$             $y_2 = y_2 + 1$
          END WHILE
$P_2$         IF $x_1 + 2x_2 - 8 = 0$ THEN
$A_5$             $y_1 = y_2 + x_2$
          ELSE
$A_6$             $y_2 = y_1 + 2x_1 + 2$
          END IF
$O_1$         PRINT $x_1$, $x_2$, $y_1$, $y_2$
```

$$A_1 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix} \quad A_2 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{vmatrix}$$

$$\overline{P}_1 = (0\ 0\ 0\ 1\ -1)$$

$$A_3 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix} \quad A_4 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{vmatrix}$$

$$\overline{P}_2 = (-8\ 1\ 2\ 0\ 0\ )$$

$$A_5 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix} \quad A_6 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2 & 2 & 0 & 1 & 0 \end{vmatrix}$$

Figure 2 Sample Program and Vector Space Model Representations

We will give some examples to demonstrate the vector and matrix representations. The subpaths A_p ending at the second assignment and A_q ending at the fourth assignment can be represented respectively as

$$A_p = A_2A_1 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{vmatrix} \qquad A_q = A_4A_3A_p = A_4A_3A_2A_1 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{vmatrix}$$

The program state after the second assignment is

$$\overline{V}_2 = A_p\overline{V}_0 = A_2A_1\overline{V}_0 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{vmatrix} \begin{vmatrix} 1 \\ x_1 \\ x_2 \\ 0 \\ 0 \end{vmatrix} = (1, x_1, x_2, x_1, x_2)^t$$

where program variables y_1 and y_2 are now expressed in terms of constants and input variables. The program state after the fourth assignment is

$$\overline{V}_4 = A_q\overline{V}_0 = A_4A_3A_2A_1\overline{V}_0 = (1,x_1,x_2,x_1-x_2,1+x_2)^t.$$

The subpath $A_q = A_4A_3A_2A_1$ implies the first predicate $\overline{P}_1$ encountered along the path being true, namely,

$$\overline{P}_1\overline{V}_2 = (0,0,0,1,-1)(1,x_1,x_2,x_1,x_2)^t = x_1 - x_2 > 0$$

where $\overline{P}_1$ denotes the predicate and $\overline{V}_2$ the program state before the predicate. The result is a scalar product.

All initial subpath matrices have the following characteristics:

1. The rows from m+2 to m+n+1 contain the actual values assigned to the corresponding program variables in terms of constants and input variables.
2. The last n columns are always zeros, since all program variables are initialized to zero at the beginning, and are always in terms of constants and input variables only.

3.3 Predicate Errors

An erroneous predicate $\overline{P}'$ is represented as:

$$\overline{P}' = \overline{P} + \overline{E} \qquad \overline{E} \neq \overline{0}$$

where $\overline{P}$ denotes the correct predicate vector, and $\overline{E}$ the error vector, an m+n+1 dimensional row vector. A predicate error will escape detection if an interpretation of the erroneous predicate is equivalent to a multiple of the correct one. That is

$$h\overline{P}'\overline{V} = \overline{P}\,\overline{V} = (\overline{P}' - \overline{E})\overline{V} = \overline{P}'\overline{V} - \overline{E}\,\overline{V}$$

where h is a non-zero constant and $\overline{V}$ the program state before the predicate. From the above equation, we have

$$[(h-1)\overline{P}' + \overline{E}]\overline{V} = \overline{0} \qquad (1)$$

Let us consider h = 1 first, name an interpretation of the erroneous predicate is exactly equal to the correct one, a relatively simple situation. When h = 1, the equation (1) reduces to

$$\bar{E}\,\bar{V} = \bar{E}A_p\bar{V}_0 = 0 \qquad (2)$$

where A_p denotes the sub ath leading to the predicate and $\bar{V}_0$ the initial state vector.

Two cases will cause $\bar{E}A_p\bar{V}_0$ equal to zero:

1. $\bar{E}A_p = \bar{0}$ (3)
2. $\bar{E}A_p\bar{V}_0 = 0$ where $\bar{E}A_p \neq \bar{0}$ (4)

In the following, we will find a solution set for $\bar{E}$, which satisfies equations (3) and (4).

Equation (3) means that the product of $\bar{E}$ and A_p is a null vector. Let us investigate the initial subpath matrix A_p. As defined previously,

$$A_p = A_i \ldots A_1 = \begin{array}{c|ccccccc|c} & 1 & x_1 & \ldots & x_m & y_1 & \ldots & y_n & \\ & 1 & 0 & \ldots & 0 & 0 & \ldots & 0 & 1 \\ & 0 & 1 & \ldots & 0 & 0 & \ldots & 0 & x_1 \\ & & & \ldots & & & \ldots & & \\ & 0 & 0 & \ldots & 1 & 0 & \ldots & 0 & x_m \\ & a_{10} & a_{11} & \ldots & a_{1m} & 0 & \ldots & 0 & y_1 \\ & & & \ldots & & & \ldots & & \\ & a_{j0} & a_{j1} & \ldots & a_{jm} & 0 & \ldots & 0 & y_j \\ & & & \ldots & & & \ldots & & \\ & a_{n0} & a_{n1} & \ldots & a_{nm} & 0 & \ldots & 0 & y_n \end{array}$$

where the last n columns of A_p are all zeros, the upper left corner is an identity matrix and the lower left corner a non-zero matrix. This non-zero matrix, as we discussed before, contains the values of corresponding program variables in terms of constants and the coefficients of input variables.

By observation, we can easily construct a solution set which satisfies $\bar{E}A_p = \bar{0}$, namely,

$$\bar{B}_1 = (a_{10}, a_{11}, \ldots, a_{1m}, -1, 0, \ \ldots\ldots \ , 0) \qquad (5)$$

$$\ldots\ldots$$

$$\bar{B}_j = (a_{j0}, a_{j1}, \ldots, a_{jm}, 0, 0, \ldots, -1, \ldots, 0)$$

$$\ldots\ldots$$

$$\bar{B}_n = (a_{n0}, a_{n1}, \ldots, a_{nm}, 0, 0, \ \ldots\ldots \ , -1)$$

The solution set consists of the last n rows of the matrix A_p with the (m+j+1)-th element (zero) replaced by -1, j = 1,...,n.

Using each vector to multiply the matrix A_p will result in zero; obviously, these vectors are the solutions to equation (3). This type of error is termed *assignment blindness*, which is solely caused by assignments along the path. Furthermore, we will prove that the solution set is unique, namely, any solution to equation (3) can be represented as a linear combination of these assignment blindness vectors.

By transposing the left hand side of equation (3), we have $A_p^t \bar{E}^t = \bar{0}$. A_p^t can b~ considered as a linear transformation matrix, an (m+n+1)-dimension to n-dimension transformation. Now we only have to prove that these n vectors $\bar{B}_j = (a_{j0}, a_{j1}, \ldots, a_{jm}, 0, \ldots, -1, \ldots, 0)$, j=1,...,n, are linearly independent. Let us just consider the last n elements alone for each vector. With a dimension of n, clearly these n vectors are linearly independent. Expanding each vector to m+n+1 dimensions, we still have n linearly independent vectors. Therefore, we conclude that any solution to equation (3) can be represented as a linear combination of ass;gnment blindness vectors, the above set (5) we derived.

Now let us study equation (4), $\bar{E}A_p\bar{V}_0 = 0$, where $\bar{E}A_p \neq \bar{0}$. Since

$\bar{E}A_p$ denotes a non-null vector, we use a vector $\bar{C}_k = (c_0,c_1,\ldots,c_m,0,\ldots,0)$ to represent $\bar{E}A_p$. The last n elements of $\bar{C}_k$ are all zeros; this is due to the fact that the last n columns of the initial subpath matrix A_p are all zeros. The product of $(\bar{E}A_p)$ and $\bar{V}_0$ is a scalar in terms of constants and input variables,

$$(\bar{E}A_p)\bar{V}_0 = (c_0,c_1,\ldots,c_m,0,\ldots,0)\ (1,x_1,\ldots,x_m,0,\ldots0)^t$$
$$= c_0 + c_1x_1 + \ldots + c_mx_m = 0. \qquad (6)$$

This type of error is termed *equality blindness*. Equality blindness can happen in the following three cases:

1. *equality predicates* encountered along the path, e.g., $x_1 = x_2$;
2. combinations of two or more inequality predicates, e.g., $x_1 >= x_2$ and $x_2 >= x_1$ imply $x_1 = x_2$; these are referred to as *coincidental equalities*;
3. equalities due to the selection of input values which form some identical relations, e.g., $x_1 = 2x_2$; these are referred to as *input equalities*.

The nature of input equalities is completely different from the first two cases which are *equality constraints* along the path. Equality constraints are path dependent, while input equalities are path independent. Since input equalities are caused by the selection of input values, and in earlier discussion, the availability of reliable data selection schemes is assumed; therefore, we can demonstrate that input equalities can be effectively excluded provided a reliable data selection scheme is adopted. Because input equalities directly relate to the selection of input data instead of the selection of paths, we will leave the discussion of input equalities to Chapter 6.

Theoretically, there are infinite $\bar{C}_k \in \bar{E}A_p$, which can cause equality blindness ($\bar{C}_k$'s are vectors with the last n elements zeros). However, for an input space of dimension m, there are no more than m linearly independent $\bar{C}_k \in \bar{E}A_p$ which satisfy equation (6). In other words, the number of linearly independent equality blindness vectors along any given path is no greater than m. The linear combinations of $\bar{C}_k$ (k=1,...,m) also satisfy equation (6). Therefore,

$$\bar{E}A_p = \sum_{k=1}^{m} d_k\bar{C}_k \qquad (7)$$

where d_k, k = 1,...m, are constants.

Since we want to find the solution for $\bar{E}$, let us set

$\bar{E} = (e_0, e_1, \ldots, e_m, e_{m+1}, \ldots, e_{m+n})$.

$$\bar{E}A_p = (e_0, e_1, \ldots, e_m, e_{m+1}, \ldots, e_{m+n}) \begin{vmatrix} 1 & 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 & 0 & \ldots & 0 \\ & & \ldots & & & \ldots & \\ 0 & 0 & \ldots & 1 & 0 & \ldots & 0 \\ a_{10} & a_{11} & \ldots & a_{1m} & 0 & \ldots & 0 \\ & & \ldots & & & \ldots & \\ a_{j0} & a_{j1} & \ldots & a_{jm} & 0 & \ldots & 0 \\ & & \ldots & & & \ldots & \\ a_{n0} & a_{n1} & \ldots & a_{nm} & 0 & \ldots & 0 \end{vmatrix}$$

$$= (e_0+e_{m+1}a_{10}+\ldots+e_{m+n}a_{n0},\ e_1+e_{m+1}a_{11}+\ldots+e_{m+n}a_{n1},\ \ldots,$$
$$e_m+e_{m+1}a_{1m}+\ldots+e_{m+n}a_{nm},\ 0,\ \ldots,\ 0)$$
$$= (e_0, e_1, \ldots, e_m, 0, \ldots, 0) + e_{m+1}(a_{10}, a_{11}, \ldots, a_{1m}, 0, \ldots, 0) + \ldots +$$
$$e_{m+n}(a_{n0}, a_{n1}, \ldots, a_{nm}, 0, \ldots, 0).$$

Using the solution set (5) we derived earlier,

$\bar{B}_j = (a_{j0}, a_{j1}, \ldots, a_{jm}, 0, \ldots, -1, \ldots, 0)$, j = 1,...,n,

and substituting $\bar{B}_j$ for $(a_{j0}, a_{j1}, \ldots, a_{jm}, 0, \ldots, 0)$, j = 1,...n, in the right hand side of the above equation, we have

$$\bar{E}A = (e_0, e_1, \ldots, e_m, 0, \ldots, 0) + e_{m+1}\bar{B}_1 + \ldots + e_{m+n}\bar{B}_n + (0, 0, \ldots, 0, e_{m+1}, \ldots, e_{m+n})$$
$$= \bar{E} + e_{m+1}\bar{B}_1 + \ldots + e_{m+n}\bar{B}_n.$$

Substituting the right hand side of equation (7) for EA_p and setting $g_j = -e_{m+j}$, $j = 1, \ldots, n$, we have

$$\sum_{k=1}^{m} d_k\overline{C}_k = \overline{E} - \sum_{j=1}^{n} g_j\overline{B}_j$$

$$\overline{E} = \sum_{j=1}^{n} g_j\overline{B}_j + \sum_{k=1}^{m} d_k\overline{C}_k \qquad (8)$$

Now we have a solution for equation (4). Here $\overline{B}_j$'s are assignment blindness vectors; $\overline{C}_k$'s are equality blindness vectors; g_j's and d_k's are constants. When the error term $\overline{E}$ can be represented as a linear combination of assignment blindness and equality blindness, the error will escape detection. This solution set consists of no more than m+n linearly independent vectors. When the second summation equals zero, solution set (8) also satisfies equation (3). Therefore, solution set (8) is a general solution for equation (2). We conclude the discussion for the case $h = 1$.

Now, consider $h \neq 1$, namely, an interpretation of the erroneous predicate is equal to a multiple of the correct one. We have to solve equation (1), $[(h-1)\overline{P}' + \overline{E}]\overline{V} = 0$, when $h \neq 1$.

Since we have already solved equation (2), $\overline{E}\,\overline{V} = 0$, with solution set (8), we can use the same result to solve equation (1). Similarly, we have

$$(h-1)\overline{P}' + \overline{E} = \sum_{j=1}^{n} g_j\overline{B}_j + \sum_{k=1}^{m} d_k\overline{C}_k.$$

Moving $\overline{P}'$, the *self-blindness* term, to the right hand side,

$$\overline{E} = \sum_{j=1}^{n} g_j\overline{B}_j + \sum_{k=1}^{m} d_k\overline{C}_k + (1-h)\overline{P}' \qquad (9)$$

Finally, we have derived a general solution to equation (1) for both cases $h = 1$ and $h \neq 1$. As compared with equation (8), the additional item in equation (9), a multiple of the predicate itself, is the result of self-blindness. When the error term can be represented as a linear combination of assignment blindness, equality blindness and self-blindness, the error mav escape detection. We can conclude that assignment blindness, equality blindness and self-blindness are the causes which make predicate errors escape detection.

3.4 Assignment Errors

An erroneous assignment A′ is represented as:

$$A' = A + A_e \qquad A_e \neq 0$$

where A denotes the correct assignment matrix, and A_e the error matrix. An assignment matrix is an identity matrix with its (m+j+1)-th row replaced by the coefficients of variables assigned to the j-th program variable. The error matrix A_e represents an error in the assignment; therefore, the (m+j+1)-th row of A_e contains the coefficients of the error term, with the rest of the rows being all zeros.

An assignment error will escape detection if a value assigned by the erroneous assignment is equivalent to the correct one. That is

$$A'\overline{V} = A\,\overline{V} = (A' - A_e)\overline{V} = A'\overline{V} - A_e\overline{V} \qquad (10)$$

where $\overline{V}$ is the program state before the assignment.

The necessary and sufficient condition for equation (10) to hold is

$$A_e\overline{V} = A_eA_p\overline{V}_0 = \overline{0} \qquad (11)$$

where A_p represents the subpath leading to the assignment and $\overline{V}_0$ the initial state vector. The three items in the left hand side of the equation can be expressed as

$$\left|\begin{array}{ccc} 0 & \cdots & 0 \\ & \cdots & \\ & \cdots & \\ 0 & \cdots & 0 \\ & \cdots & \\ e_0 & \cdots & e_{m+n} \\ & \cdots & \\ 0 & \cdots & 0 \end{array}\right| \left|\begin{array}{ccccccc} 1 & \cdots & 0 & 0 & \cdots & 0 \\ & \cdots & & & \cdots & \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ a_{10} & \cdots & a_{1m} & 0 & \cdots & 0 \\ & \cdots & & & \cdots & \\ a_{j0} & \cdots & a_{jm} & 0 & \cdots & 0 \\ & \cdots & & & \cdots & \\ a_{n0} & \cdots & a_{nm} & 0 & \cdots & 0 \end{array}\right| (1,x_1,\ldots,x_m,0,\ldots,0)^t$$

The solution to the above equation is surprisingly simple. Although A_e is a matrix, it has only one non-zero row vector, and behaves exactly like a vector. We can easily apply the solution set (8) to this equation. Using $\overline{E}_i$ to denote the non-zero row vector in A_e,

$$\overline{E}_i = \sum_{j=1}^{n} g_j\overline{B}_j + \sum_{k=1}^{m} d_k\overline{C}_k \qquad (12)$$

We can conclude that assignment blinoness and equality blindness are the causes which make assignment errors escape detection.

In the above discussion, we analyzed the situation where a subpath which occurs before an erroneous assignment may cancel out the error. Later, we will analyze a similar situation where a subpath which occurs after an erroneous assignment may cancel out the error as well.

An assignment error may cause an erroneous predicate interpretation, which leads to a domain error; it may also cause a computation error. In other words, an assignment error may reveal itself through a domain error or a computation error.

3.4.1 The Effect of Assignment Errors on a Predicate Interpretation

We will investigate how an assignment error affects the interpretation of a subsequent predicate. Consider an initial subpath $A_qA'A_p$ which leads to a predicate $\overline{P}$, where A_p represents an initial subpath before the erroneous assignment A' and A_q represents a subpath after the assignment A'. The predicate interpretation is $\overline{P}A_qA'A_p\overline{V}_0$, where $\overline{V}_0$ denotes the initial state. If the predicate $\overline{P}$ follows the assignment A' immediately, the subpath matrix A_q becomes an identity matrix.

The assignment error will not be revealed if the predicate interpretation is a multiple of the correct one, that is,

$$h\overline{P}A_qA'A_p\overline{V}_0 = \overline{P}A_qAA_p\overline{V}_0 = \overline{P}A_q(A' - A_e)A_p\overline{V}_0 = \overline{P}A_qA'A_p\overline{V}_0 - \overline{P}A_qA_eA_p\overline{V}_0$$

We can reorganize the equation as

$$\overline{P}A_q[(h-1)A' + A_e]A_p\overline{V}_0 = 0 \qquad (13)$$

Consider h = 1 first, namely, the erroneous predicate interpretation is exactly equal to the correct one. In this case, equation (13) reduces to

$$\overline{P}A_qA_eA_p\overline{V}_0 = 0 \qquad (14)$$

The product of the first three items $\overline{P}A_qA_e$ results in a row vector. In the previous section, we solved the equation $\overline{E}A_p\overline{V}_0 = 0$ with solution set (8). Now, substituting $\overline{P}A_qA_e$ for $\overline{E}$, we have

$$\overline{P}A_qA_e = \sum_{j=1}^{n} g_j\overline{B}_j + \sum_{k=1}^{m} d_k\overline{C}_k \qquad (15)$$

where $\overline{B}_j$'s and $\overline{C}_k$'s are vectors which represent assignment blindness and equality blindness, respectively.

$\overline{P}A_qA_e$, where $\overline{P}$ denotes the predicate, A_q the subpath and A_e the error term of the assignment, can be expressed as

$$\overline{P}A_qA_e = (p_0, p_1, \ldots, p_{m+n}) \begin{vmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ & & \cdots & & & \cdots & \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ a_{10} & a_{11} & \cdots & a_{1m} & a_{1(m+1)} & \cdots & a_{1(m+n)} \\ & & \cdots & & & \cdots & \\ a_{i0} & a_{i1} & \cdots & a_{im} & a_{i(m+1)} & \cdots & a_{i(m+n)} \\ & & \cdots & & & \cdots & \\ a_{n0} & a_{n1} & \cdots & a_{nm} & a_{n(m+1)} & \cdots & a_{n(m+n)} \end{vmatrix} \begin{vmatrix} 0 & \cdots & 0 \\ & \cdots & \\ 0 & \cdots & 0 \\ 0 & \cdots & 0 \\ & \cdots & \\ e_0 & \cdots & e_{m+n} \\ & \cdots & \\ 0 & \cdots & 0 \end{vmatrix}$$

The first two items result in a row vector,

$$\bar{P}A_q = (p_0+p_{m+1}a_{10}+\cdot\cdot+p_{m+n}a_{n0},\ \ldots,\ p_m+p_{m+1}a_{1m}+\cdot\cdot+p_{m+n}a_{nm},$$
$$p_{m+1}a_{1(m+1)}+\cdot\cdot+p_{m+n}a_{n(m+1)},\ \ldots,\ p_{m+1}a_{1(m+n)}+\cdot\cdot+p_{m+n}a_{n(m+n)})$$

therefore,

$$\bar{P}A_qA_e = (p_{m+1}a_{1(m+i)}+\cdot\cdot+p_{m+n}a_{n(m+i)})\ (e_0,e_1,\ldots,e_{m+n})$$

Using $\bar{E}_i$ to denote the non-zero vector of the matrix A_e, and b_i to denote the constant $p_{m+1}a_{1(m+i)}+\ldots+p_{m+n}a_{n(m+i)}$, and substituting $b_i\bar{E}_i$ for $\bar{P}A_qA_e$ in the left hand side of equation (15), we have

$$b_i\bar{E}_i = \sum_{j=1}^{n} g_j\bar{B}_j + \sum_{k=1}^{m} d_k\bar{C}_k \qquad (16)$$

First, this result confirms the conclusion derived early in this section that when $\bar{E}_i$ is a linear combination of assignment blindness and equality blindness, the assignment error will escape detection. In other words, the subpath before the erroneous assignment may nullify the effect of the error on a subsequent predicate interpretation. Second, if $b_i\bar{E}_i = 0$, the error will not be revealed either. In this situation, the constant $p_{m+1}a_{1(m+i)} + \ldots + p_{m+n}a_{n(m+i)}$ equals zero. This case indicates that, combined with the predicate, the subpath which occurs after the erroneous assignment may nullify the effect of the error with respect to the predicate.

Now consider $h \neq 1$, $\bar{P}A_q[(h-1)A'+ A_e]A_pV_0 = 0$. Since the first three items $\bar{P}A_q[(h-1)A'+ A_e]$ result in a vector, using solution set (8) again, we have

$$\bar{P}A_q[(h-1)A'+ A_e] = \sum_{j=1}^{n} g_j\bar{B}_j + \sum_{k=1}^{m} d_k\bar{C}_k$$

$$\bar{P}A_qA_e = \sum_{j=1}^{n} g_j\bar{B}_j + \sum_{k=1}^{m} d_k\bar{C}_k + (1-h)\bar{P}A_qA'$$

Similarly, we can derive the solution as in (16),

$$b_i\bar{E}_i = \sum_{j=1}^{n} g_j\bar{B}_j + \sum_{k=1}^{m} d_k\bar{C}_k + (1-h)\bar{P}A_qA' \qquad (17)$$

Solution set (17) is a general solution for equation (13). The third item $(1-h)\bar{P}A_qA'$ is similar to the self-blindness term in predicate testing. This is another case that combined with the predicate, a subpath which occurs after the erroneous assignment may nullify the effect of the error on the predicate interpretation.

3.4.2 The Effect of Assignment Errors on Computation

We just discussed how an assignment error affects a predicate interpretation. Similarly, we will discuss how an assignment error affects a computation. A computation error will be revealed when an incorrect output is produced. We can treat an output statement as a vector just as in the analysis of predicates, where the vector contains the coefficients of variables in the output expression. For instance, an output statement "PRINT $2x_1+x_2$" can be represented as $(0,2,1,0,\ldots)$; an output statement "PRINT y_1, y_2" can be considered two separate statements "PRINT y_1" and "PRINT y_2", and expressed in the vector representation accordingly.

We can use a similar procedure to discuss the effect of assignment errors on the output as that on the predicate. An assignment error will not be revealed if the output is equivalent to the correct one, that is,

$$\bar{O}A_qA'A_p\bar{V}_0 = \bar{O}A_qAA_p\bar{V}_0 = \bar{O}A_q(A'-A_e)A_p\bar{V}_0 = \bar{O}A_qA'A_p\bar{V}_0 - \bar{O}A_qA_eA_p\bar{V}_0 \quad (18)$$

where $\bar{V}_0$ denotes the initial state, A_i' the erroneous assignment, A the correct assignment, A_e the error term, A_p the initial subpath before the assignment, and A_q the subpath after the assignment.

The necessary and sufficient condition for equation (18) to hold is

$$\bar{O}A_qA_eA_p\bar{V}_0 = 0 \tag{19}$$

Using the result in the last section, we have

$$b_i\bar{E}_i = \sum_{j=1}^{n} g_j\bar{B}_j + \sum_{k=1}^{m} d_k\bar{C}_k \tag{20}$$

where $b_i = o_{m+1}a_{1(m+i)}+\ldots+o_{m+n}a_{n(m+i)}$.

First, when $\bar{E}_i$ is a linear combination of assignment blindness and equality blindness, the assignment error will escape detection. Second, when $b_i = 0$, combined with the output statement, a subpath which occurs after the erroneous assignment may cancel out the error with respect to the output statement.

3.5 Summary of Predicate and Assignment Errors

Assignment, equality and self-blindness cause predicate and assignment errors to escape detection for a given path. In other words, any error term in a predicate or assignment will be undetectable if the error is a linear combination of assignment, equality and self-blindness. In addition to blindness, a subpath which occurs after an erroneous assignment may nullify the effect of the assignment error on a subsequent predicate or output statement.

The exclusion of blindness will guarantee the exposure of predicate errors which may cause domain errors. If we not only exclude blindness, but also prevent potential assignment errors from being nullified by subsequent subpaths, we can guarantee the exposure of assignment errors which may cause domain errors. These conclusions form the foundation of blindness-based testing for linearly domained programs.

CHAPTER FOUR

BLINDNESS-BASED TESTING STRATEGY

4.1 Blindness Testing Concept

The objective of blindness-based testing is to expose blindness errors by selecting a set of reliable test paths. The term reliable means that any blindness error will be exposed by at least one of the selected paths. For each program statement (assignment or predicate), we select a set of paths leading to the statement, which can reliably test the statement for blindness errors. Combining all selected paths will form a set of reliable paths for the entire program.

4.1.1 Cancellation of Assignment Errors

In predicate testing, since blindness causes predicate errors to escape detection, the elimination of blindness will guarantee the exposure of predicate errors which may lead to domain errors. In other words, the selection of a set of reliable paths will expose potential predicate errors.

In assignment testing, however, the matter is not that simple. In addition to blindness, other factors may also cause assignment errors to escape detection. An assignment error may lead to a domain error if the affected program variable is involved either directly or indirectly in a subsequent predicate interpretation. If the affected program variable is not involved in a subsequent predicate, of course, the interpretation of the predicate will not reveal the assignment error. An assignment error is *canceled out* with respect to a subsequent predicate if the error is contained in a subpath leading to the predicate and the affected program variable is involved either directly or indirectly in the predicate interpretation, but the interpretation of the predicate is equal to the correct one as if there were no assignment error contained in the subpath. The cancellation may be caused by a subpath which occurs either before or after the erroneous assignment. Two examples are provided in Fig 3.

Example 1

Correct Code

$Y_1 = 5$
...
$Y_2 = 2Y_1 + 1$
...
IF $Y_1 + Y_2 > 0$
...

Incorrect Code

$Y_1 = 5$
...
$Y_2 = Y_1 + 6$ *
...
IF $Y_1 + Y_2 > 0$
...

Example 2

Correct Code

$Y_1 = 2X + 3$
...
$Y_2 = 1 + Y_1$
...
IF $4X - Y_1 + Y_2 > 0$
...

Incorrect Code

$Y_1 = 0$ *
...
$Y_2 = 1 + Y_1$
...
IF $4X - Y_1 + Y_2 > 0$
...

Figure 3 Cancellation of Assignment Errors

The incorrect assignments are marked with a star. The first example shows the effect of a subpath which occurs before the erroneous assignment, and cancels out the assignment error with respect to a subsequent predicate. The incorrect assignment assigns the same value to the variable Y_2 as the correct one. In fact this is an example of assignment blindness. It cancels out the error term for any subsequent predicate interpretation.

The second example shows the effect of a subpath which occurs after the erroneous assignment, and cancels out the assignment error with respect to a subsequent predicate. Combined with the predicate, the assignment $Y_2 = 1 + Y_1$ cancels out the error term for the predicate interpretation --

$$4X - Y_1 + (1 + Y_1) = 4X + 1$$

which is exactly the same as the predicate interpretation of the correct code.

In general, blindness errors are the only causes for the situation of example 1 (Fig. 3), where a subpath which occurs before an erroneous assignment cancels out the error with respect to subsequent predicates. However, we are not able to attribute the causes to a certain type of error for the situation of example 2, where a subpath which occurs after an erroneous assignment cancels out the error with respect to a subsequent predicate.

If an erroneous program variable is not involved either directly or indirectly in any subsequent predicates, we can conclude that it will not cause any domain error, though it may cause computation errors.

We say an assignment error is *nullified* with respect to a subsequent predicate if the error term is either canceled out or not involved in the predicate interpretation.

The concept of blindness-based testing is focused on the local environment. If we trace program statements step by step, the selection of paths based on the blindness concept will guarantee the exposure of blindness errors in any of the predicate or assignment statement to be tested. For an erroneous predicate, a different predicate interpretation as compared with the correct one will be produced by at least one of the selected paths. For an erroneous assignment, a different value as compared with the correct one will be assigned to the affected program variable by at least one of the selected paths. If a symbolic trace is performed, a different symbolic value will expose the erroneous predicate or assignment. However, a domain error will be detected only by an incorrect predicate interpretation. An assignment error may not lead to a domain error if the error is nullified. Therefore, the exposure of an assignment error in a local environment does not automatically lead to the exposure in a global environment.

The objective of blindness-based testing is to expose all blindness errors. The situation where an assignment error may be canceled out by a subpath which occurs before the erroneous assignment can be prevented completely. The situation where an assignment error may be canceled out by a subpath which occurs after the erroneous assignment is preventable, but the cost is quite high. Each selected subpath needs to be extended from the assignment towards a subsequent predicate.

In the following sections, we will present two blindness-based path selection algorithms for linearly domained programs. The first one emphasizes the completeness of testing, while the second one emphasizes the effectiveness of testing.

4.1.2 Blindness-Based Path Selection

The path selection algorithm which will be proposed in this section is based on the analysis of assignment and predicate errors from the last chapter. The aim is to select a reliable set of paths to test assignments and predicates for domain errors. Predicate testing is straightforward; the test will guarantee the exposure of all predicate errors characterized by blindness. In assignment testing, we have to expose all assignment errors characterized by blindness, and prevent the errors from being canceled out.

Recall our discussion on assignment testing in the last chapter, where the solution to the equation

$$\bar{P}A_q[(h-1)A'+A_e]A_p\bar{V}_0 = 0 \qquad (13)$$

is

$$b_i\bar{E}_i = \sum_{j=1}^{n} g_j\bar{B}_j + \sum_{k=1}^{m} d_k\bar{C}_k + (1-h)\bar{P}A_qA' \qquad (17)$$

where $\bar{P}$ denotes the predicate, A' the erroneous assignment, A_e the error term, A_q the subpath after the assignment, A_p the subpath before the assignment, $\bar{V}_0$ the initial state, $\bar{E}_i$ the non-zero row vector of A_e; $\bar{B}_j$'s and $\bar{C}_k$'s are assignment blindness vectors and equality blindness vectors respectively; g_j's, d_k's and h are constants; b_i equals

$$p_{m+1}a_{1(m+i)}+\cdots+p_{m+n}a_{n(m+i)},$$

a scalar product of the predicate $\bar{P}$ and the (m+i+1)-th column of the subpath matrix A_q.

When $\bar{E}_i$ is a linear combination of assignment, equality and self-blindness, this assignment error will escape detection. When $b_i = 0$, this assignment error will also escape detection with respect to the predicate $\bar{P}$.

Therefore, when we select a test path for an assignment based on the blindness concept, we also need to prevent the possible assignment error from being canceled out by a subpath which occurs after the assignment. In other words, we have to make sure that b_i is not zero.

The process can be described as follows.

For any assignment to be tested, whenever we select a subpath leading to the assignment, we extend this subpath to a subsequent predicate. If the potential assignment error is nullified with respect to this predicate, the subpath is extended to a next predicate or a different subpath is selected after that assignment. This process continues until a predicate is reached where the potential assignment error will not be nullified in the predicate interpretation, or it can be ascertained that no such predicate exists.

The following path selection algorithm consists of two major operations. It first selects a set of reliable subpaths for each program construct (assignment or predicate) according to the sequence of its appearance in the program, and then combines all selected paths to form a reliable set for the entire program. The first operation is carried out by four steps, and the second operation has one step. We will describe each step in detail later.

Blindness-Based Path Selection Algorithm I

For each program construct (assignment or predicate):

1. Select an initial subpath or extend a previously selected initial subpath leading to the program construct. Test its feasibility, and exclude any infeasible subpath.
2. Calculate the blindness space for the program construct. Reject the subpath if the dimension of the blindness space for the program construct cannot be reduced by this subpath.
3. If the program construct is an assignment and the subpath is not rejected, extend the subpath to a subsequent predicate. If

a potential assignment error will be nullified with respect to this predicate, extend the subpath to a next predicate or select a different subpath, until a predicate is reached where the potential assignment error will not be nullified or it is established that no such predicate exits.

4. When the blindness space for the program construct becomes a null space, or contains invariant expressions only, a reliable set for the construct has been obtained. Move to the next construct (GO TO 1).
5. After all constructs in the program are processed, combine any initial subpaths if one contains the other, and extend all initial subpaths to complete paths. These paths form a reliable set.

The first step simply selects an initial subpath by assigning a Boolean value to each predicate encountered along the subpath leading to the construct to be tested, e.g., $(P_1{:}t, P_2{:}f, \ldots)$, or extends a previously selected subpath leading to the construct. The feasibility test can be carried out by calling a linear programming routine; all predicates encountered in the subpath form the constraints.

In step two, the blindness space for each construct consists of assignment blindness vectors, equality blindness vectors (equality constraints along the subpath), and a self-blindness vector. In the blindness space for a predicate, the self-blindness vector is the predicate expression itself. In the blindness space for an assignment, the self-blindness vector is represented by a product of three items ($\overline{P}A_qA'$ in equation 17): the assignment, the subpath after the assignment, and the predicate. Each blindness space is represented in matrix form with each blindness vector as a row vector. The calculation for the intersection of two blindness spaces is based on a method appearing in the appendix of reference [13].

In the third step, in order to prevent a possible assignment error from being nullified, we can always check if

$p_{m+1}a_{1(m+i)}+\cdots+p_{m+n}a_{n(m+i)}$,

a scalar product of a subsequent predicate and the (m+i+1)-th column of a subpath matrix, equals zero or not. If the scalar product equals zero, we either extend the subpath to another predicate or select a different subpath.

Step four gives a stopping criterion, which signals the selection of a sufficient set of subpaths for the construct to be tested. Invariant expressions mentioned here are expressions in the blindness space which cannot be eliminated by the selection of different paths. Detailed discussion will be given in the next chapter.

The fifth step simply combines subpaths and extends subpaths to complete paths.

There are two assumptions made for this algorithm. First, for any assignment or predicate statement in the program, it is assumed that there exists an initial subpath which can reach that statement. Second, for any initial subpath, it is assumed that the subpath can be extended to a halt statement (which corresponds to an exit node in the control flow graph) in the program. These assumptions are not extraordinary requirements. There must be serious structural problems which will prevent a statement from being reached or a path from reaching the exit. In fact, these problems can be easily identified by static testing [6], a class of techniques to produce general information about a program, such as cross-reference, search for particular kinds of errors, etc. These two assumptions can be dropped completely if the capability of back tracking is included in the algorithm at the expense of an increased complexity.

The initial blindness space for each program construct has a dimension of m+n+1. Since every selected path will reduce the dimension of the blindness space by at least one, therefore, a set of reliable paths consists of at most m+n+1 paths.

One complication of this algorithm is the trace (step 3) from an assignment under test towards a subsequent predicate in order to prevent a potential assignment error from being canceled out. The trace will not end until a predicate is found such that a scalar product between the predicate and a column of a subpath matrix does not equal zero, or it is established that no such predicate exists. This trace is potentially endless. If we relax a bit with regard to theoretical completeness, such a costly trace may not even be necessary. This relaxation, however, is not done in an arbitrary fashion. In practice, the situation where an assignment error may escape detection can be reduced to a minimum by blindness-based testing. The reason is as follows.

When an assignment error is canceled out by a subpath which occurs before the erroneous assignment, the error will remain undetectable for any path which extends that subpath. When an assignment error is canceled out with respect to a subsequent predicate by a subpath which occurs after the erroneous assignment, the error may be revealed by the selection of another subpath after the assignment. For a given path, an assignment error will escape detection completely if the error is nullified with respect to every subsequent predicate along the path. This is a rather strong condition. For a set of paths, an assignment error will escape detection completely if the error is nullified with respect to *every subsequent predicate* along *all* paths in the set. This situation rarely happens. So long as the error is not nullified in one of the predicate interpretations along one of the paths, the error will be exposed.

Since our focus is on domain errors, we only concentrate on the effect of assignment errors on subsequent predicate interpretations. In fact an assignment error may lead to an incorrect computation. Therefore, when an assignment error is nullified with respect to every subsequent predicate along a path, the error may be revealed by an incorrect output. Due to this fact, the likelihood that an assignment error may completely escape detection is further reduced.

The same argument can be applied to the self-blindness term in assignment testing ($\bar{P}A_qA'$ in equation 17), where the error term of the assignment causes the interpretation of a subsequent predicate equal to a multiple of the correct interpretation. The chance that this situation happens for every subsequent predicate is very unlikely, since different subpaths are followed, and different predicates are encountered. The chance that this situation happens for every subsequent predicate along all selected paths is even more unlikely. So long as one of the interpretations is not equal to a multiple of the correct one, the error will be exposed. Therefore, we need not to be concerned with this error term in assignment testing.

Summarizing our discussion, the Blindness-Based Path Selection

Algorithm I can be simplified if we abandon the costly trace and concentrate on the situation where an assignment error may escape detection due to a subpath which occurs before the assignment. Since blindness-based testing can effectively minimize the cancellation of assignment errors, the reliability of testing will not be deteriorated by these simplifications. In the next section, we will introduce the concept of computation blocks, which can further simplify the complexity of the proposed algorithm.

4.2 Computation Blocks and Assignment Testing

In assignment testing, we can treat a block of consecutive assignments as if it were a single statement; this is termed a *computation block* in reference [13]. In the vector space model, an assignment is represented as a matrix; a computation block is represented using the multiplication of consecutive assignment matrices, which also can be represented as a matrix.

4.2.1 Computation Blocks

In the following example, there are two input variables x_1 and x_2, and two program variables y_1 and y_2. A_1 and A_2 are two consecutive assignments, they form a computation block A_2A_1.

A_1: $y_1 = 1 + 2x_1 - x_2 + 3y_1 + y_2$

A_2: $y_2 = 2 + y_1$

$$A_1 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & -1 & 3 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix} \quad A_2 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 & 0 \end{vmatrix} \quad A_2A_1 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & -1 & 3 & 1 \\ 3 & 2 & -1 & 3 & 1 \end{vmatrix}$$

The computation block in fact transforms the two assignments into

$$y_1 = 1 + 2x_1 - x_2 + 3y_1' + y_2'$$

$$y_2 = 3 + 2x_1 - x_2 + 3y_1' + y_2'$$

where y_1' and y_2' are in terms of the program state prior to the block. These two assignments are equivalent to the original ones.

This transformation is an important feature of the computation block. Thus a block of consecutive assignments can be treated as a single assignment. The whole block is updated simultaneously instead of sequentially as in conventional programming. Because of this powerful feature, we need not to deal with the individual assignment within the block. Instead, we treat the block as a whole. As far as testing is concerned, a computation block is considered as a single entity. We are no longer concerned how the first assignment will affect the second one within the block; we are only concerned how the block is affected by the previous program state and how the following program state is affected by the block.

The introduction of computation blocks has no effect on predicate testing; only assignment testing will be affected. A computation block assigns program variables simultaneously in a parallel fashion. The advantage of using computation blocks is that it will greatly reduce the complexity of testing. Instead of testing each assignment, now we only have to test each computation block, which amounts to a reduction of test paths from a maximum of n(m+n+1) down to m+n+1. The blindness space for a computation block contains n vectors of assignment blindness and no more than m vectors of equality blindness; therefore, no more than m+n+1 paths are needed to test a computation block. This compares with the proposed algorithm where a maximum of m+n+1 paths is needed to test an assignment, and a maximum of n(m+n+1) paths to test all assignments in a computation block.

Most of the analysis of assignment testing discussed previously can be extended to computation blocks without any problem. Blindness-based testing still guarantees the exposure of all blindness errors up to the computation block to be tested. For any erroneous assignment within the block, the affected program variable will be assigned a different value as compared with the correct one by at least one of the selected paths. Testing each individual assignment or a block of assignments as a whole makes no difference in this respect.

The following facts still hold for a computation block: for a given path, an assignment error will escape detection completely if the error is nullified with respect to every subsequent predicate along the path; for a set of paths, an assignment error will escape detection completely if the error is nullified with respect to every subsequent predicate along all paths in the set.

The disadvantage of using computation blocks is that we cannot completely prevent an assignment error from being canceled out by a subpath which occurs after the erroneous assignment. For a single assignment, we can extend a selected subpath to a subsequent predicate, and we are able to determine whether a potential assignment error would be nullified with respect to that predicate. For a block of assignments, we are not able to perform such a trace to prevent potential assignment errors from being canceled out. The reason is that we are not able to resolve the combination effect of several program variables on subsequent predicates. Therefore, we are not able to determine whether potential assignment errors would be nullified with respect to these predicates.

4.2.2 Simplified Path Selection Algorithm

This following algorithm is a simplification of the Blindness-Based Path Selection Algorithm I. It tests each computation block instead of each assignment. It does not perform the trace from an assignment under test towards a subsequent predicate.

Blindness-Based Path Selection Algorithm II

For each program construct (computation block or predicate):

1. Select an initial subpath or extend a previously selected initial subpath leading to the program construct. Test its feasibility, and exclude any infeasible subpath.
2. Calculate the blindness space for the program construct. Reject the subpath if the dimension of the blindness space for the program construct cannot be reduced by this subpath.
3. When the blindness space for the program construct becomes a null space, or contains invariant expressions only, a reliable set for the construct has been obtained. Move to the next construct (GO TO 1).
4. After all constructs in the program are processed, combine any initial subpaths if one contains the other, and extend all initial subpaths to complete paths. These paths form a reliable set.

The reachability and halting assumptions we made for the first algorithm also apply to this algorithm. The blindness space for predicates contains assignment, equality and self-blindness. The blindness space for computation blocks contains only assignment and equality blindness.

The complexity of this algorithm is m+n+l for each computation block and predicate. The upper bound for the whole program will be k(m+n+l), where k is the total number of computation blocks and predicates in the program. In fact, the actual number of paths needed to test the whole program is far less than k(m+n+l), because many paths are shared, namely, one path can be used to test different predicates and computation blocks. We will demonstrate this fact later.

4.3 Comparison Between the Two Algorithms

The difference between these two algorithms involves assignment testing. The Blindness-Based Path Selection Algorithm I tests each assignment, and attempts to prevent possible assignment errors from being canceled out in subsequent predicate interpretations. The algorithm traces from the assignment under test towards a subsequent predicate, which will force any potential assignment error to be exposed in the predicate interpretation. Therefore, this algorithm will guarantee the exposure of all assignment errors which are characterized by blindness, and may cause domain errors.

However, the complexity of the Blindness-Based Path Selection Algorithm I is high; it needs m+n+l paths to test each assignment, and up to n(m+n+l) paths to test each computation block. The operation of this algorithm is complicated too. The trace from an assignment under test towards a subsequent predicate is a complex task, which needs expensive matrix operations, and the trace is potentially endless. The question is whether the benefit can justify the cost, or whether it is worthwhile to pay the excessive cost for the capacity to prevent the situation where an assignment error may escape detection completely, which rarely happens.

The approach taken by the Blindness-Based Path Selection Algorithm II seems more effective. There are several advantages with such an approach. First, the complexity of this algorithm is low; it needs only m+n+1 paths to test a computation block. Second, the operation of this algorithm is relatively simple. The trace which extends from the assignment to a subsequent predicate is no longer performed. Third, this approach in assignment testing is consistent with that of predicate testing. Therefore, it is possible to formulate a uniform strategy to deal with both predicate and assignment errors.

The disadvantage of the Blindness-Based Path Selection Algorithm II is that in theory the exposure of all assignment errors which may cause domain errors is not guaranteed. Although the test will force the exposure of assignment errors in a local environment, it cannot ensure the exposure in a global environment. For any assignment, the selection of a set of reliable paths guarantees that if there is an error in the assignment, a different value as compared with the correct one will be assigned to the affected program variable by at least one of the selected paths. However, it does not guarantee that the assignment error will be exposed in a subsequent predicate interpretation. In other words, the assignment error may be nullified. As we discussed earlier, the likelihood that an assignment error may completely escape detection is rare. We [illegible] the advantages far outweigh the disadvantages. Most impc tantl . this is the algorithm which can be implemented practically.

4.4 Blindness-Based Testing and Branch Coverage

There are other path selection criteria; the most common ones are statement coverage, branch coverage, and path coverage. Statement coverage will ensure the execution of every statement in the program, which requires every program statement to appear in at least one of the test paths. Branch coverage will ensure the execution of every possible branch in the program, which requires every predicate to be tested using both true and false values. Branch coverage subsumes statement coverage. Path coverage will ensure the execution of every possible path in the program, which requires the evaluation of all feasible combinations of predicates. Path coverage subsumes branch coverage.

For each statement or branch in the program, the determination of a feasible path which contains the statement or branch is in general undecidable. Nevertheless, statement coverage and branch coverage are practical criteria used in program testing. In all remaining discussion, the existence of feasible paths for both statement coverage and branch coverage is assumed.

Though path coverage is the most complete and desirable measure, in reality, it is usually impossible or impractical. A program with a "WHILE" loop can contain infinitely many paths. Disregarding loops, a program with just 20 "IF-THEN-ELSE" branches, can contain over a million possible paths.

Blindness-based path selection algorithms imply statement coverage for the program, since each predicate and assignment (including the case of computation blocks) will [illegible] tested. Blindness-based testing also implies branch coverage for the program, although it does not indicate the coverage explicitly. In testing an "IF-THEN" branch, the blindness-based algorithm does not explicitly force the Boolean value "FALSE" to be taken, though it guarantees the Boolean value "TRUE" to be taken by testing the assignments immediately after the predicate. There is no similar problem for an "IF-THEN-ELSE" branch, since the Boolean value "FALSE" is guaranteed to be taken by testing the assignments after the "ELSE" statement. However, the tester should have no difficulty to conform to branch coverage by simply taking "TRUE" and "FALSE" at least once for every "IF-THEN" branch. This does not necessarily mean that additional test paths are required. Usually several paths are needed to test a program construct. The tester can assign "TRUE" and "FALSE" to different paths.

4.5 An Example

We will give a demonstration of Blindness-Based Path Selection Algorithm II, which is to select test paths to expose domain errors (Fig. 4).

We label all assignments and predicates in the program in sequence. There are two predicates P_1 and P_2, and five computation blocks A_2A_1, A_4A_3, A_5, A_6 and A_7. We have defined "program state" in the last chapter to represent the values of all variables at any point in the program execution. We will use the notation $(P_i:t|f)$ to indicate the Boolean value (TRUE or FALSE) of the i-th predicate, and notations $(P_i:?)$ and $(A_j:?)$ to indicate the predicate and assignment under test respectively. Thus a subpath leading to a predicate or assignment to be tested can be represented as a string of $(P_i:t,P_j:f,\ldots,P_m:?)$ or $(P_i:t,P_j:f,\ldots,A_n:?)$.

```
       READ $x_1$, $x_2$
A1     $y_1 = x_1$
A2     $y_2 = x_2$
P1     WHILE $y_1 - y_2 > 0$
A3         $y_1 = y_1 - y_2$
A4         $y_2 = y_2 + 1$
       END WHILE
P2     IF $x_1 + 2x_2 - 8 >= 0$ THEN
A5         $y_1 = y_2 + x_2$
       ELSE
A6         $y_2 = y_1 + 2x_1 + 2$
       END IF
A7     $y_2 = x_1 + x_2 + y_1 + y_2$
       PRINT $x_1$, $x_2$, $y_1$, $y_2$
```

Figure 4 Sample Program for Path Selection

In the following, we will select a set of reliable paths for each program construct (predicate or computation block) according to the sequence of its appearance in the program.

1. Computation block A_2A_1: There is only one subpath. The program state after A_2 is $y_1 = x_1$, $y_2 = x_2$.

2. Predicate P_1: First we select subpath (P_1:?). The program state after A_2 is $y_1 = x_1$, $y_2 = x_2$. Its blindness space contains a self-blindness vector and two assignment blindness vectors. Second we select another subpath (P_1:t,P_1:?). The program state after A_4 is $y_1 = x_1 - x_2$, $y_2 = x_2 + 1$. Its blindness space also contains a self-blindness vector and two assignment blindness vectors. The

intersection of these two blindness spaces contains only the self-blindness vector, which cannot be eliminated. Therefore a reliable set for this predicate consists of two subpaths $(P_1:?)$ and $(P_1:t,P_1:?)$.

3. Computation block A_4A_3: We extend selected subpath $(P_1:?)$ to $(P_1:t,A_3:?)$ and another selected subpath $(P_1:t,P_1:?)$ to $(P_1:t,P_1:t,A_3:?)$. The program state of variables has not changed, and no calculation is needed.

4. Predicate P_2: We extend selected subpath $(P_1:?)$ to $(P_1:f,P_2:?)$. Its program state is $y_1 = x_1$, $y_2 = x_2$. We extend another selected subpath $(P_1:t,P_1:?)$ to $(P_1:t,P_1:f,P_2:?)$. Its program state is $y_1 = x_1 - x_2$, $y_2 = x_2 + 1$. Each blindness space contains a self-blindness vector and two assignment blindness vectors. The intersection of these two spaces contains only the self-blindness vector. As discussed previously, these two subpaths are sufficient to test this predicate.

5. Assignment A_5: We extend selected subpath $(P_1:f,P_2:?)$ to $(P_1:f,P_2:t,A_5:?)$ and another selected subpath $(P_1:t,P_1:f,P_2:?)$ to $(P_1:t,P_1:f,P_2:t,A_5:?)$. This assignment shares the same blindness space with P_2 (no change of the program state).

6. Assignment A_6: We again extend selected subpath $(P_1:f,P_2:?)$ to $(P_1:f,P_2:f,A_6:?)$ and another selected subpath $(P_1:t,P_1:f,P_2:?)$ to $(P_1:t,P_1:f,P_2:f,A_6:?)$. This assignment shares the same blindness space with P_2 (no change of the program state).

7. Assignment A_7: We extend the subpath to $(P_1:f,P_2:t,A_7:?)$. Its program state after assignment A_5 is $y_1 = 2x_2$, $y_2 = x_2$. We extend another subpath to $(P_1:t,P_1:f,P_2:f,A_7:?)$. Its program state after

assignment A_6 is $y_1 = x_1 - x_2$, $y_2 = x_1 - x_2 + 2$. Each blindness space contains two assignment blindness vectors. The intersection of these two blindness spaces is the null space.

Finally, we have four paths $(P_1{:}f,P_2{:}t)$, $(P_1{:}t,P_1{:}f,P_2{:}t)$, $(P_1{:}f,P_2{:}f)$ and $(P_1{:}t,P_1{:}f,P_2{:}f)$. In the above description of the path selection process, we omitted the feasibility test, which is required at each decision point (predicate) to extend the selected path. This test can be carried out easily by a linear programming method. In fact all these selected paths are feasible. This set of paths also conforms to branch coverage.

From this example we can see an important feature of blindness-based testing: although the algorithm specifies the selection of a set of paths for every predicate and computation block, only a few of them involve the actual calculation of the intersection of blindness spaces. In this example, it only happens in selecting paths for predicates P_1 and P_2, and assignment A_7. In selecting paths for computation block A_4A_3 and assignments A_5 and A_6, no calculation of the blindness space is needed. They share the result of P_1 or P_2. Only the feasibility test is needed. Thus, the actual calculation required for this algorithm is much less than it might appear. In this example, a reliable test set consists of only four paths, far less than the theoretical upper bound $k(m+n+1) = 30$ (six predicates and computation blocks, four input and program variables). This illustrates the fact that many paths may be shared for testing different predicates and computation blocks.

4.6 Implementation

In the last chapter, we have introduced matrix operations in the vector space model. The implementation of this model, however, need not follow its development exactly. Symbolic execution [3], a simple method to execute the symbolic representation of path domains and computations, will be used to implement these operations. All operations defined in this model can be carried out equivalently by symbolic execution. The vector space model is useful to analyze the validity and reliability of program testing. When it comes to implementation, cost and efficiency are our major concern. Matrix operations are very inefficient, especially matrix multiplications.

4.7 Comparison of Different Methods

The Blindness-Based Path Selection Algorithm II has been implemented in a computer system called BBTEST. An experiment has been conducted to compare different methods. Four programs (see Appendix 1) are selected from [10]. The data is compiled in Table 1. The results from BBTEST are compared with the ones from another computer system named SPTEST, which implements Zeil's algorithm in predicate testing. Both methods are blindness-based. SPTEST tests predicates only, while BBTEST tests both predicates and assignments. Also included is the data which conforms to branch coverage when the Blindness-Based Path Selection Algorithm II is applied.

No attempt is made to optimize the number of selected paths in any of the methods. Although these results are far from conclusive, they do provide some insight into blindness-based testing. First, it again confirms the fact that the number of required paths to test the whole program is far less than its theoretical upper bound. Second, the number of paths needed to test assignments in addition to predicates is marginal. In two of the examples, there is no increase. This situation agrees with the fact that many paths are shared. Third, the number of paths needed to provide branch coverage is also marginal. In two of the examples, there is no increase. Although all selected programs are

Program	Number of Input and Program Variab.	Number of Predicates	Number of Computation Blocks	Number of Paths Required		
				SPTEST	BBTEST	BBTEST plus Branch Coverage
1	4	3	2	3	4	4
2	4	2	3	2	2	3
3	4	4	3	4	4	6
4	5	3	2	4	5	5

Note: 1. The first computation block at the beginning of the program is not counted for the number of computation blocks, since it has no effect on path selection.
2. SPTEST does not assure branch coverage.

Table 1 Experiment Results for Different Methods

relatively small in size, there is no reason to doubt these facts will not hold for programs of larger size.

One distinction between BBTEST and SPTEST is that BBTEST implies statement coverage, while SPTEST does not. A sufficient set of test paths generated by SPTEST only guarantees the exposure of all predicate errors which are characterized by blindness, and may cause domain errors. It cannot guarantee the same for assignment errors. In the following, we provide an example (Fig. 5) in which SPTEST may fail to include a test path that may cause a domain error due to an erroneous assignment.

```
         READ x, y
A1       w = x
P1       IF w + x >= 0 THEN
P2            IF w > 0 THEN
A2                 w = x + y
              ELSE
A3                 w = x - y
              END IF
         END IF
P3       IF w + x - 2y < 0 THEN
         ...
```

Figure 5. Sample Program for Two Path Selection Methods

In SPTEST, subpaths $(P_1{:}f, P_3{:}?)$ and $(P_1{:}t, P_2{:}t, P_3{:}?)$ will constitute a sufficient set for predicate P_3. This set will guarantee the exposure of all possible predicate errors in P_3, which are characterized by blindness, and may cause domain errors. However, none of these two paths traverses assignment A_3. If there is an error in A_3, it can certainly cause an incorrect interpretation of predicate P_3. As a result, domain errors may occur. SPTEST cannot guarantee the exposure of such errors. Since BBTEST will test each assignment in addition to predicates, these errors will be revealed.

CHAPTER FIVE

INVARIANT EXPRESSIONS

5.1 Definitions

One problem common to path analysis testing is the selection of a finite set of paths from the set of all possible paths in the program. Although the path selection algorithm proposed in the last chapter does provide the path selection criterion and the stopping criterion based on the dimension and components of the blindness space, it is still up to the tester to decide how to select paths and when to stop.

In this chapter, we will analyze the components of the blindness space, and summarize the heuristics and procedures which will guide the selection of paths and the completion of the selection process.

The path selection criterion and the stopping criterion indicate: A path will be selected if it can reduce the dimension of the blindness space for the program construct, until the blindness space becomes a null space or contains invariant expressions only.

What is an invariant expression? An invariant expression is an algebraic expression which can be added to the correct expression of a program statement (assignment or predicate) without being detected along *all* possible paths through that statement. Namely, the execution of the program along all paths will not distinguish whether the statement contains an invariant expression or not. Therefore, invariant expressions cannot be eliminated from the blindness space by the selection of multiple paths.

Invariant expressions are in fact combinations of assignment, equality and self-blindness. These three types of blindness are originally defined for a single path, while invariant expressions are defined for multiple paths. In other words, a blindness term for a single path may not be a blindness term for multiple paths. A self-blindness will always become an invariant expression; an assignment blindness and an equality blindness may or may not become invariant expressions depending on program constructs.

The goal of the path selection process is to reduce the dimension of the blindness space by selecting different paths until a null space is obtained or the space contains invariant expressions only.

Sahay [10] reported, based on his experiments with SPTEST, a computer system implementing Zeil's model to test predicates, that there are unused variables, equality blindness, self-blindness, and invariant expressions in the blindness space. These vectors form an *irreducible error space* which cannot be eliminated by the selection of different paths. We will demonstrate later that all components in the irreducible error space fall under the definition of invariant expressions.

Since invariant expressions cannot be eliminated from the selection of different paths, early identification of invariant expressions can avoid a useless search for new paths, which may be endless when loops are involved.

An invariant expression, from its definition, is invariant with respect to a program construct along all possible paths leading to the construct. If an expression is invariant with respect to a program construct along certain paths but not all possible path this expression is not an invariant expression, and can be eliminated from the blindness space by the proper selection of multiple paths.

The difficulty arises from the determination as to whether an expression is invariant with respect to a program construct along certain or all paths. Theoretically, this problem is in general undecidable. In practical situations, nevertheless, with the help of heuristics we are able to deal with most programs. In the following we will study some characteristics of invariant expressions, which will be useful for the identification of these expressions.

5.2 Characteristics of Invariant Expressions

Invariant expressions can be classified based on their appearance in the program. There are two types of invariant expressions: one can be traced to a single statement (assignment or predicate) in the program, the other can be traced to multiple statements in the program.

As we mentioned earlier, invariant expressions are combinations of the three types of blindness. Examining the value of an expression in the blindness space, which can be obtained by substituting constants and input variables for program variables in the expression, can help to identify assignment, equality and self-blindness which comprise the expression.

A self-blindness vector only appears in the blindness space for predicates. Since self-blindness has no effect on any assignment in general, it will not be a component in the blindness space for assignments. Because every blindness space for a subpath leading to a predicate contains a self-blindness vector, the self-blindness vector cannot be eliminated from the intersection. Therefore, a self-blindness vector is always an invariant expression.

An equality blindness vector in the blindness space is either in the original form of an equality predicate, or consists of constants and input variables only (after program variables are replaced), namely, the vector contains all zeros in its last n elements (n is the number of program variables).

An equality blindness vector will not become an invariant expression unless an equality constraint is in effect along all possible paths leading to the program construct to be tested. The equality constraint can arise from an equality predicate, a non-equality predicate, or multiple inequality predicates (a coincidental equality).

When a program variable remains constant in a local or global environment, it will become an invariant expression. An unused variable is a special case of constant program variables, where the program variable has been initialized but not used. However, the situation where an unused variable becomes an invariant expression has nothing to do with the usage of the variable. It is decided by the assignment (definition). As long as the variable remains constant, whether it has been used or not, it becomes an invariant expression.

Since the stopping criterion for the path selection algorithm depends on the identification of invariant expressions, we will impose a rigid criterion for the identification. An invariant expression should not be admitted until it can be proved that it is indeed invariant along all possible paths. Therefore, the identification process must be supported by program analysis. It is easy to verify an invariant expression which arises from a single statement. For an invariant expression arising from multiple statements, analytical methods (induction, for instance) used in program verification are needed. We will analyze an example to demonstrate how to reduce the dimension of the blindness space and how to identify invariant expressions.

5.3 An Example

The following program (Fig. 6) is from [10], which performs integer round-up.

```
          READ n
A1        i = 0
A2        j = n
A3        r = 0
P1        WHILE j >= 1
A4             i = i + 1
A5             j = n - i
          END WHILE
A6        r = n - i
P2        IF r >= .5 THEN
A7             i = i + 1
          END IF
          PRINT n, i
```

Figure 6 Sample Program for Invariant Expressions

In this program, there are three program variables i, j and r, and one input variable n.

For predicate P_1, we select subpath (P_1:?). Its program state is

$i = 0$, $j = n$, $r = 0$. The blindness space along this subpath is

$$\begin{array}{|cccc|c}
\multicolumn{1}{c}{i} & j & r & \multicolumn{1}{c}{} & \\
0 & 0 & 0 & -1 & 1 \\
0 & 1 & 0 & 0 & n \\
-1 & 0 & 0 & 0 & i \\
0 & -1 & 0 & 1 & j \\
0 & 0 & -1 & 0 & r
\end{array}$$

It consists of four vectors, the first three are assignment blindness and the fourth is self-blindness. Now we select another subpath $(P_1{:}t, P_1{:}?)$. The purpose of selecting this path is that at least two program variables (i and j) are assigned different values. We hope it will reduce the dimension of the blindness space. Its program state is $i = 1$, $j = n - 1$, $r = 0$. The blindness space along this subpath is

$$\begin{array}{|cccc|c}
1 & -1 & 0 & -1 & 1 \\
0 & 1 & 0 & 0 & n \\
-1 & 0 & 0 & 0 & i \\
0 & -1 & 0 & 1 & j \\
0 & 0 & -1 & 0 & r
\end{array}$$

It also consists of four vectors. The intersection of these two spaces does reduce the dimensionality as we expected, which is

$$\begin{array}{|ccc|c}
1 & -1 & 0 & 1 \\
-1 & 0 & 0 & n \\
1 & 0 & 0 & i \\
0 & 1 & 0 & j \\
0 & 0 & 1 & r
\end{array}$$

The intersection consists of three vectors. Since our objective is to eliminate as many vectors as possible, we select a third subpath $(P_1{:}t, P_1{:}t, P_1{:}?)$, hoping it will reduce the dimensionality further. Its program state is $i = 2$, $j = n - 2$, $r = 0$. The blindness space along this subpath is

$$\begin{array}{|cccc|c}
2 & -2 & 0 & -1 & 1 \\
0 & 1 & 0 & 0 & n \\
-1 & 0 & 0 & 0 & i \\
0 & -1 & 0 & 1 & j \\
0 & 0 & -1 & 0 & r
\end{array}$$

The intersection of the above three blindness spaces, to our surprise, is still

$$\left|\begin{array}{rrr|c} 1 & 0 & -1 & 1 \\ -1 & 0 & 0 & n \\ 1 & 0 & 0 & i \\ 0 & 0 & 1 & j \\ 0 & 1 & 0 & r \end{array}\right|$$

According to the path selection criterion, the third subpath is rejected since it cannot reduce the dimension of the intersection of blindness spaces. If we try to select different subpaths, namely taking more iterations, all results are just the same: none of the subpaths can reduce the dimension of the intersection. Why does this happen? We will analyze these three remaining vectors in the blindness space.

The third vector $(-1\ 0\ 0\ 1\ 0)^t$ is a self-blindness vector, which is, of course, an invariant expression. The second vector $(0\ 0\ 0\ 0\ 1)^t$ is a constant program variable, where the variable r is initialized to zero at the beginning, and not being assigned any new value in this environment. This is also an invariant expression.

The first vector $(1\ -1\ 1\ 0\ 0)^t$ is a bit tricky -- its expression "1 - n + i" seems unrelated to the program. We can check its value. Taking one subpath, say, the first one, and substituting the values of program variables (i = 0, j = n, r = 0), we have "1 - n" (since i = 0). The value of the expression is equal to the interpretation of the predicate, which indicates self-blindness is one of the factors that comprise the expression.

In order to eliminate the effect of self-blindness, we can add the self-blindness vector to the first vector, which results in $(0\ -1\ 1\ 1\ 0)^t$. Its expression is "-n + i + j". Now we have restored its original form -- this expression is derived from assignment

A_5: j = n - i. It is the last assignment before exiting the loop. No matter how many iterations are taken, it is always the last assignment before the loop ends. Therefore we have the expression "-n + i + j", which equals zero. For the subpath without entering the loop, the second and third assignments (j = n and r = 0) imply that the expression "-n + i + j" also equals zero. Since subpaths leading to P_1 either enter or skip the loop, having exhausted all possible paths, we have proved that the expression "-n + i + j" is indeed an invariant expression. Now we have identified all three vectors remaining in the blindness space as invariant expressions. According to the path selection algorithm, no more paths are needed for this predicate (P_1).

In the above discussion, "j = n - i" is actually a loop invariant. It is true on entry to the loop, and remains true after each iteration (including the final exit from the loop).

For predicate P_2 we extend the subpath to (P_1:f,P_2:?). Its program state is i = 0, j = n, r = n. The blindness space along this subpath is

$$\left|\begin{array}{cccc} 0 & 0 & 0 & -.5 \\ 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{array}\right| \begin{array}{c} 1 \\ n \\ i \\ j \\ r \end{array}$$

It contains four vectors, where the first three are assignment blindness and the fourth is self-blindness. We extend another subpath to (P_1:t,P_1:f,P_2:?). Its program state is i = 1, j = n - 1, r = n - 1. The blindness space along this subpath is

$$\left|\begin{array}{cccc} 1 & -1 & -1 & -.5 \\ 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{array}\right| \begin{array}{c} 1 \\ n \\ i \\ j \\ r \end{array}$$

The intersection of these two spaces is

$$\left|\begin{array}{ccc} .5 & -.5 & -.5 \\ -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right| \quad \begin{array}{c} 1 \\ n \\ i \\ j \\ r \end{array}$$

We can prove that if we select other subpaths with more iterations, none of them will reduce the dimension of the intersection of blindness spaces. In fact these three vectors are all invariant expressions.

Let us start with the easiest, the third vector, which is a self-blindness vector "r - .5". We can trace the origins of the other two vectors by adding the first vector to the second, and the first vector to the third. The results are $(0\ -1\ 1\ 1\ 0)^t$ and $(0\ -1\ 1\ 0\ 1)^t$. The former "-n + i + j" appears in the previous blindness space for predicate P_1. The only assignment (A_6) between these two states (P_1 and P_2) involves the assignment of program variable r, which has no effect on the expression "-n + i + j". Therefore, it remains an invariant expression. The latter "-n + i + r" is a new expression, which is the direct result of assignment A_6: r = n - i. Since every subpath leading to predicate P_2 passes through A_6, and P_2 follows A_6 immediately, as a result, "r = n - i" holds for all possible paths. Hence, it is also an invariant expression.

5.4 Summary of Procedures

From above discussion, it is unlikely that there exists a general method to deal with invariant expressions due to the undecidability of program testing and the complication of invariant expressions. However, invariant expressions are often identifiable by careful inspection and analysis of the program. There appears to be an effective way to deal with invariant expressions.

The following procedures can be taken during the testing process:

1. Eliminate vectors in the blindness space which are variant.

Vectors which are variant along different paths can be eliminated by the proper selection of multiple paths. In order to eliminate assignment blindness vectors, multiple paths should be selected where program variables are assigned different values. In order to eliminate equality blindness vectors, multiple paths should be selected where different Boolean values are assumed, or different predicate interpretations are produced.

2. Identify the type of invariant expressions.

It is easy to identify self-blindness and other invariant expressions which can be traced to a single program statement, such as constant program variables and equality blindness. As a result, complicated invariant expressions which cannot be traced to a single statement are isolated for later investigation.

3. Check the value of the expression in the blindness space.

Substituting the values of program variables in the expression will provide information how the expression is composed by assignment blindness, equality blindness, and self-blindness.

4. Use composition and decomposition methods.

The process to add or subtract an identified assignment, equality, or self-blindness vector to another vector in the blindness space under investigation will help to trace to the origin of the expression and restore the assignment, equality and self-blindness which comprise the origin of the expression. Therefore, we can determine if the expression is indeed invariant or not. The analysis should be based on program statements along different paths, associated program states of variables, associated blindness spaces, and intersections of blindness spaces.

The above procedures are inter-related, and should be taken in an integrated manner. We do not know if an expression is indeed invariant until it is either eliminated by the selection of multiple paths or proven as an invariant expression. For all vectors in the blindness space, the first choice is trying to eliminate them by selecting different paths; if not successful, the second choice is trying to prove them as invariant expressions; there is no other choice.

5.5 Heuristics in Selecting Paths

Although the existence of any method which will guarantee an optimal test set appears unlikely, some heuristics can certainly reduce the amount of calculation and the number of paths needed.

The most effective way to reduce the number of paths for blindness testing is to select different paths where as many program variables as possible are assigned different values along different paths. In this way, the dimension of the blindness space can be reduced quickly. This criterion is especially effective when nested loops and branches are involved. Testers usually select short paths (this is a good strategy), but overlook maximizing different values along different paths. If a program variable has the same value in two different paths, the variable will certainly appear in the intersection of these two blindness spaces. Our objective is to eliminate as many vectors in the blindness space as possible in a single path.

The other criterion is to identify invariant expressions as early as possible. If some expressions cannot be eliminated from the blindness space along several paths, program inspection and analysis should be made to rule out the possibility of invariant expressions. Any attempt to eliminate invariant expressions will result in a usele search for new paths, which is potentially endless when loops are involved.*

CHAPTER SIX

TEST DATA SELECTION AND RELATED ISSUES

6.1 Test Data Selection

This study, as we stated before, is focused on path selection based on the blindness concept. Blindness-based testing is independent of any test data selection schemes. We suggest, however, the Domain Testing Strategy (mentioned in Chapter 1), is a suitable test data selection scheme to carry out the blindness testing. In Chapter 3, we have mentioned input equalities, another source of equality blindness. The effective exclusion of input equalities depends on a reliable data selection method. In this chapter we will give detailed discussion of input equalities and explain why domain testing can effectively prevent input equalities. In the following, we will first give a brief introduction to domain testing.

6.2 Domain Testing Strategy

Domain testing constructs test data for selected paths in order to expose domain errors. This method falls into the category of path analysis testing. All feasible paths in the program partition the input space into subdomains. Input points in each subdomain cause the execution of statements along a certain path, which calculates the function associated with the path. The boundary of each subdomain is formed by predicates along the path. Domain testing analyzes the boundaries of subdomains to detect domain errors. Since domain errors are manifested by a shift in part of the subdomain boundary or a change in the corresponding relational operator, domain testing selects points on or near the subdomain boundary. This selection is based on the fact that points near the boundary are most sensitive to domain errors.

The segments of the subdomain boundary are determined by predicates along the path, which are termed *borders*. There are two types of test points, defined by their positions with respect to the border. An ON test point lies on the border; an OFF test point lies on the open side of the border, which is in the adjacent subdomain. In a two-dimensional case, domain testing selects two ON test points on the border, and one OFF test point with a small distance from the testing border. In an m-dimensional case, domain testing selects m linearly independent ON test points on the border, and one OFF test point in the adjacent subdomain whose projection on the given border is a convex combination of these m points.

There are some limitations with domain testing for certain types of errors. missing path errors, for instance, which are in fact common to path analysis testing methods. Nevertheless, domain testing can reliably detect any border shift or change in the corresponding relational operator within a tolerance limit.

6.3 Input Equality

The derivation of input equalities is from the following equation which appears in Chapter 3,

$$\bar{E}A_p\bar{V}_0 = 0, \quad (\bar{E}A_p \neq \bar{0}) \qquad (4)$$

where $\bar{E}$ is the error term in the predicate, A_p the subpath leading to the predicate, and $\bar{V}_0$ the initial state $(1, x_1, \ldots, x_m, 0, \ldots, 0)$.

We mentioned in Chapter 3, Zeil has identified two types of equality blindness: equality predicates and coincidental equalities.

These two types of equality blindness form equality constraints along the path. Whenever the error term $\bar{E}$ is a linear combination of equality constraints, the error will escape detection along the path.

Now let us consider all solutions to equation (4). Apparently, in addition to the error term $\bar{E}$, the input $\bar{V}_0$ also plays a role in this equation. This equation can be satisfied even without the presence of equality constraints. This situation arises when the error term coincides with input values. This is referred to as an *input equality*. Here is an example.

Correct Code	Incorrect Code
...	...
$x_1 + 3x_2 - 6 > 0$	$2x_1 + x_2 - 6 > 0$
...	...

The error term for the incorrect code is $x_1 - 2x_2$. If the input data selection happens to be x_1 as two times as x_2, say, $x_1 = 2$ and $x_2 = 1$, then the error term $x_1 - 2x_2$ will result in zero. In general, any selection satisfying $x_1 = 2x_2$ will nullify the error.

Input equalities have two constituents: an error term and the input values which match the error term. For each error term in a statement, there can be infinite sets of input values which match the error term. On the other hand, given one set of input values, it can match infinitely many potential error terms.

A typical predicate statement is

$$a_0+a_1x_1+\ldots+a_mx_m+a_{m+1}y_1+\ldots+a_{m+n+1}y_n+(b_0+b_1x_1+\ldots\ldots+b_{m+n+1}y_n) \text{ ROP } 0$$

where ROP stands for a relational operator, and items in the parenthesis form the error term. Since the interpretation of the error term is comprised of constants and input variables only, where program variables are being replaced, geometrically the error term of an input equality is strictly contained on a line (2-dimensional), a plane (3-dimensional), or a hyperplane (m-dimensional) in the input space.

Because the error term is unknown, we do not know which set of input values might cause input equality. We can expose the potential error term, however, by a careful selection of input data. In a two-dimensional space, if we select one point (a pair of input values), it can match infinitely many potential error terms; if we select two independent points, it can only match a unique error term since two points determine a line; if we select three independent points, no error term can be matched, namely, the error term will be revealed by at least one of the test points. Therefore, input equalities will be detected.

We can easily expand the above observation. Since three independent points will not lie on a line in a two-dimensional space, thus m+1 independent points will not lie on a hyperplane in an m-dimensional space. Hence, m+1 independent points will be the minimum number which guarantees the exclusion of input equalities in any statement. The simple explanation is that at least one of the m+1 test points will assure the error term being non-zero, and, as a result, the error will be exposed. There is a noticeable similarity between this result and the data selection method proposed by domain testing: both require m+1 test points.

The understanding of input equalities will assist us in data selection along chosen paths. With m input variables, if less than m+1 data points are selected to test a predicate, the test will not be considered reliable for its vulnerability to input equalities. Using this criterion we can conclude that domain testing is reliable to detect input equalities. Since domain testing selects m linearly independent ON points and one OFF point, these m+1 points are guaranteed to lie on different hyperplanes. If there [illegible] an error in the predicate, a domain error will be detected.

The existence of input equalities will not inval[illegible]e the path selection algorithms based on the blindness concept as long as a reliable data selection method is adopted. Assignment blindness and equality constraints are usually path dependent, namely, the errors may escape detection for certain paths but not for others; the exclusion of these errors requires a set of paths. Input equalities are path independent, namely, the errors may escape detection for certain input values instead of certain paths; the exclusion of these errors does not necessarily require a set of paths, but a set of test points.

Because of the path independence, the number of test points necessary to detect all errors due to input equalities in the program will be far less than the number of points necessary to detect other blindness errors which are path dependent. Just considering predicate errors, if a path traverses all predicates in the program, we may only need to select data to test this path alone, which will expose input equalities for all predicates in the program. This is certainly not true for path dependent blindness errors, where a set of paths are usually required.

We have demonstrated that m+1 independent test points will detect input equalities for any given program statement. The distribution of these test points is not as strict as that of domain testing, where m ON points and one OFF point are required. The exclusion of input equalities only requires that the m+l points be independent, in other words, lie on different hyperplanes. Since the Domain Testing Strategy is an established data selection method, which can reliably detect domain errors, and also satisfy the criterion to detect input equalities; it can be a desiraole tool to carry out blindness-based testing.

CHAPTER SEVEN

SUMMARY

7.1 Blindness-based Testing

The philosophy of program testing is to assure the quality of software. Program testing involves the execution of a program over a set of test data. Since there is no general testing method for arbitrary programs, researchers attempt to circumvent this problem by concentrating on certain classes of programs over certain classes of errors. Path analysis testing, a class of testing strategies, involves the selection of test paths and the selection of test data for the chosen paths. Zeil's study focuses on the selection of test paths. He has identified three types of blindness errors which will escape detection along a given path for linearly domained programs, a class of programs defined in Chapter 2, and proposed a strategy to expose blindness errors by selecting a set of test paths.

White and Cohen [11] proposed the Domain Testing Strategy, a simple and effective method to detect domain errors by selecting test data on or near the boundary of a path domain. The remaining problem, general to path oriented methods, is how to deal with the number of paths in the program, which is potentially infinite. Zeil has applied the blindness concept to select a finite set of paths to test assignments and predicates separately. The question is whether these two tests can be combined and simplified.

This research attempts to select a finite set of paths for domain testing. The objective is to develop a blindness-based testing method for linearly domained programs, which will test assignments and predicates for domain errors, and can be implemented practically.

The main contributions of this research are the analysis of assignment testing, the proposal of a uniform strategy to combine assignment and predicate testing, the analysis of invariant expressions, the summary of procedures to deal with invariant expressions, and the computer implementation of the proposed testing strategy.

Zeil's model has been refined. Input equalities, another potential source of blindness, have been analyzed.

Two blindness-based path selection algorithms have been proposed. The Blindness-Based Path Selection Algorithm I tests each assignment and predicate with a complexity of $O(n^2)$, where n is the number of program variables. This complexity is consistent with the upper bound suggested by Zeil. This algorithm will guarantee the exposure of all assignment and predicate errors which are characterized by blindness and may cause domain errors. The Blindness-Based Path Selection Algorithm II tests each block of assignments and predicate with a complexity of $O(n)$. In theory, this proposed algorithm cannot guarantee the exposure of all assignment errors which are characterized by blindness. Under certain circumstances, an assignment error may not manifest itself in subsequent predicates following specific paths, and the error will escape detection. In practice, however, these circumstances are rare; therefore, the Path Selection Algorithm II can be as effective as the Path Selection Algorithm I.

Due to the existence of invariant expressions, a combination of blindness errors, the search for a finite set of test paths may become endless. Therefore, the understanding of invariant expressions is crucial to blindness-based testing. This research formally defines invariant expressions, and classifies them according to their appearance in the program. Procedures to identify invariant expressions are summarized, and a stopping criterion for testing as well as heuristics for selecting paths is suggested.

7.2 Future Research

Although linearly domained programs represent a large class of data processing programs, this class remains rather restricted. Extending blindness-based testing to non-linearly domained programs will be a major challenge. The main problem is that blindness-based testing requires the closure of program operations under a vector space, which most non-linearly domained programs cannot satisfy. Many features associated with linearly domained programs, e.g., the feasibility of a test path, the reliability of blindness testing, will become uncertain for non-linearly domained programs.

In order to extend the testing to non-linearly domained programs, new theory needs to address the above problems; new strategies need to trim the number of paths, which will, conceivably, grow exponentially. Zeil attempted to address the non-linear class of programs by introducing a method called "perturbation testing" [14, 15], which can be further explored.

The other problem associated with non-linearly domained programs is the selection of test data along chosen paths. Domain testing, the method we recommended to perform the selection of test data, is mainly applicable to linearly domained programs. The selection of test data is inseparable from the selection of test paths. How to select reliable test data is another challenge facing non-linearly domained programs.

REFERENCES

1. T.A. Budd, R.DeMillo, R.J. Lipton and F.G. Sayward, "The Design of a Prototype Mutation System for Program Testing", *Proc. 1978 NCC*, Anaheim, CA, 1978.

2. L.A. Clarke, J. Hassell, and D.J. Richardson, "A Close Look at Domain Testing", *IEEE TSE*, Vol. SE-8, No. 4, Jul. 1982, pp. 380-390.

3. L.A. Clarke and D.J. Richarson, "Symbolic Evaluation Methods -- Implementations and Applications", in *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, Eds. Amsterdam, The Netherlands: North-Holland, 1981, pp. 65-102.

4. W.E. Howden, "Reliability of the Path Analysis Testing Strategy", *IEEE TSE*, Vol. SE-2, Sept. 1976, pp. 208-215.

5. W.E. Howden, "Algebraic Program Testing", *ACTA Informatica*, Vol. 10, 1978, pp. 53-66.

6. W.E. Howden, "A Survey of Static Analysis Methods", in *Tutorial: Software Testing and Validation Techniques*, E. Miller and W.E. Howden, Eds. IEEE Computer Society, Washington, D.C. 1981, pp. 101-115.

7. W.E. Howden, "A Survey of Dynamic Analysis Methods", in *Tutorial: Software Testing and Validation Techniques*, E. Miller and W.E. Howden, Eds. IEEE Computer Society, Washington, D.C. 1981, pp. 209-231.

8. W.E. Howden, "Weak Mutation Testing and Completeness of Program Sets", *IEEE TSE*, Vol. SE-8, Jul. 1982, pp. 371-379.

9. G.J. Myers, *The Arts of Software Testing*, 1979, John Wiley & Sons.

10. P.N. Sahay, *Heuristics for Selecting Best Paths for Testing Computer Programs*, M.Sc. thesis, University of Alberta, 19[illegible]

11. L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing", *IEEE TSE*, Vol. SE-6, No. 3, May 1980, pp. 247-257.

12. M.R. Woodward, D. Hedley, and M.A. Hennell, "Experience with Path Analysis and Testing of Program", *IEEE TSE*, Vol. SE-6, No. 3, May 1980, pp. 278-286.

13. S.J. Zeil, *Selecting Sufficient Sets of Test Paths for Program Testing*, Ph.D. dissertation, Ohio State University, 1981.

14. S.J. Zeil, "Testing for Perturbations of Program Statements", *IEEE TSE*, Vol. SE-9, No. 3, May 1983, pp. 335-346.

15. S.J. Zeil, "Perturbation Testing for Computation Errors", *Seventh International conference on Software Engineering*, March 1984, pp. 257-265.

APPENDIX 1

TESTING PROGRAMS

Program 1: Euclid GCD

```
READ, X, Y
A = X
B = Y
WHILE (A .NE. B) DO
     WHILE (A .GT. B) DO
          A = A - B
     END WHILE
     WHILE (B .GT. A) DO
          B = B - A
     END WHILE
END WHILE
PRINT, X, Y, A
STOP
END
```

Required Paths for the Second Algorithm:

1. $(P_1{:}t, P_2{:}t, P_2{:}t, P_2{:}f, P_3{:}t, P_3{:}f, P_1{:}f)$
2. $(P_1{:}t, P_2{:}t, P_2{:}f, P_3{:}f, P_1{:}f)$
3. $(P_1{:}t, P_2{:}f, P_3{:}t, P_3{:}f, P_1{:}t, P_2{:}t, P_2{:}f, P_3{:}f, P_1{:}f)$
4. $(P_1{:}t, P_2{:}f, P_3{:}t, P_3{:}t, P_3{:}f, P_1{:}f)$

Program 2: Integer Round-up

```
READ, N
I = 0
J = N
R = 0
WHILE (J .GE. 1.0) DO
     I = I + 1
     J = N - I
END WHILE
R = N - I
IF (R .GE. .5) THEN DO
     I = I + 1
END IF
PRINT, N, I
STOP
END
```

Required Paths for the Second Algorithm:

1. $(P_1{:}t, P_1{:}f, P_2{:}t)$

2. $(P_1{:}t, P_1{:}t, P_1{:}f, P_2{:}t)$

Program 3: Integer Division Remainder

```
READ, X, Y
R = 0
A = 0
IF (X .GE. 0) THEN DO
     IF (Y .GT. 0) THEN DO
          R = X
          WHILE (R .GE. Y) DO
               A = Y
               WHILE (R .GE. A) DO
                    R = R - A
                    A = A + A
               END WHILE
          END WHILE
     END IF
END IF
PRINT, R, X, Y
STOP
END
```

Required Paths for the Second Algorithm:

1. $(P_1:t,P_2:t,P_3:f)$
2. $(P_1:t,P_2:t,P_3:t,P_4:t,P_4:f,P_3:t,P_4:t,P_4:f,P_3:f)$
3. $(P_1:t,P_2:t,P_3:t,P_4:t,P_4:t,P_4:f,P_3:t,P_4:t,P_4:f,P_3:f)$
4. $(P_1:t,P_2:t,P_3:t,P_4:t,P_4:t,P_4:t,P_4:f,P_3:t,P_4:t,P_4:f,P_3:f)$

Program 4: Euclid GCF

```
READ, A, B
S = A
T = B
U = 0
WHILE (S .NE. T) DO
     IF (S .GT. T) THEN DO
          S = S - T
     ELSE DO
          U = S
          S = T
          T = U
     END IF
END WHILE
IF (S .EQ. 1) THEN DO
     PRINT, A, B
ELSE
     PRINT, A, B, S
END IF
STOP
END
```

Required Paths for the Second Algorithm:

1. $(P_1{:}t, P_2{:}t, P_1{:}f, P_3{:}f)$

2. $(P_1{:}t, P_2{:}f, P_1{:}t, P_2{:}t, P_1{:}f, P_3{:}t)$

3. $(P_1{:}t, P_2{:}f, P_1{:}t, P_2{:}t, P_1{:}t, P_2{:}t, P_1{:}f, P_3{:}f)$

4. $(P_1{:}t, P_2{:}t, P_1{:}t, P_2{:}t, P_1{:}f, P_3{:}f)$

5. $(P_1{:}t, P_2{:}t, P_1{:}t, P_2{:}f, P_1{:}t, P_2{:}t, P_1{:}t, P_2{:}f, P_1{:}t, P_2{:}t, P_1{:}f, P_3{:}t)$

APPENDIX 2

BLINDNESS-BASED TESTING SYSTEM

The Blindness-based Testing System (BBTEST) has implemented the second path selection algorithm presented in chapter four. The implementation modified the Sufficient Path Testing System (SPTEST), a system implementing Zeil's algorithm in predicate testing, to incorporate the second path selection algorithm. The system is located on Pembina of the UNIX operating system at the Computing Science Department, the University of Alberta. The programming language used is FORTRAN. The source code is stored under the directory /ul/prof/leew/program.testing/bbtest/src.

The operation of the system can be divided into three phases. The first phase is the compilation of the input program; the second phase is the selection of test paths through an interactive symbolic execution; the third phase is the determination as to whether to accept or reject the selected path based on the path selection algorithm. To the end the user will be informed whether a sufficient set of test paths has been achieved or not.

In order to execute the system, the user should change the directory to /ul/prof/leew/program.testing/bbtest/bin. There are three commands: takea, takeb, and takeab. Takea will compile the input program. Takeb will select test paths. Takeab will compile the program and select paths. A program only needs to be compiled once. However, there is no limit to the number of paths to be selected. After the program is compiled (through takea or takeab), the command takeb can be executed repeatedly. The syntax of command lines:

```
takea <input program name>
takeab <input program name>
takeb
```

During the execution, the system generates some information which is stored under the directory /ul/prof/leew/program.testing/Files. The following are some of the files:

program.lst: the list of the input program;

program.prd: the list of predicates along the path(s);

span.vector: the blindness space(s) after selected path(s).

The system will not operate correctly if an input program contains syntax errors or statements unrecognized by the system.

The following is a list of FORTRAN statements recognized by the system:

ACCEPT	END WHILE
ASSIGNMENT STATMENT	ELSE DO
AT END DO	FORMAT
CASE	GO TO
COMPUTED GO TO	IF (...) <EXE STATEMENT>
CONTINUE	IF (...) THEN DO
DO CASE	PRINT
DO	READ (FORMATTED OR UNFORMATTED)
END	STOP
END CASE	WHILE (...) DO
END IF	WRITE

Some restrictions are set for the input program:

Maximum length (lines)	150
Arithmetic Statements	100

Assignment Statements	50
Do Loops	25
Input Variables	20
Output Variables	20
Labels	40
Predicate Statements	100
Read Statements	20
Write Statements	20