The University of Alberta

An Automatic Test Path Generator for Path Oriented Testing Strategies

by

© Kwok-on Terence Lai

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1988

ISBN 0-315-45615-9

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR : **Kwok-on Terence Lai**

TITLE OF THESIS : **An Automatic Test Path Generator for Path Oriented Testing Strategies**

DEGREE FOR WHICH THIS THESIS WAS PRESENTED : **Master of Science**

YEAR THIS DEGREE GRANTED : **1988**

[illegible]

K. Terence Lai

Permanent Address:

2/Fl., 69A Fuk Wing Street
Shamshuipo, Kowloon
Hong Kong

Date August 11, 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 **An Automatic Test Path Generator for Path Oriented Testing Strategies** submitted by **Kwok-on Terence Lai** in partial fulfillment of the requirements for the degree of **Master of Science.**

[illegible]

Dr. L. J. White
Supervisor

[illegible]

Dr. M. Green

[illegible]

Dr. P. Rudnicki

Peter R. Winters

Dr. P. Winters
External

Date August 3, 1988

ABSTRACT

Path analysis testing is a white box testing approach which generates test data using the program structure to test computer programs. The process is divided into two operations: the selection of paths where testing is to be conducted and the selection of test data which execute the chosen paths. This research is concerned with the automation of a test path selection process which utilizes a path analysis testing strategy to detect predicate errors in computer programs. The approach is based on a vector space analytical model. A summary of the vector space analytical model, as well as an overview of SPTEST, a system that implements the model, are given. The problems of SPTEST are discussed, and ideas in solving those problems are proposed. The concepts of subpath extension are introduced and a new heuristic method based on those concepts for the automatic test path generation is provided. A new system called SPTEST II, which implements the heuristic method, has been developed and is presented as an improved version of SPTEST. Experimental results obtained from SPTEST II are discussed and analyzed. Some side issues related to path selection concerning invariant expressions and testing programs which utilize arrays are also discussed.

Acknowledgements

I would like to thank my supervisor, Dr. L. J. White, for many, many things. Without his suggestions and patience, I would not have completed this. I am grateful to the members of my examining committee, Dr. M. Green, Dr. P. Rudnicki, and Dr. P. Winters for their valuable comments.

I would also like to thank T. M. Koon, B. Lee, S. Chew, and D. Johnson who read rough drafts of the thesis and contributed help in many ways.

Special thanks to A. Collins, L. Gaetz, and K. Lochmanetz who provided me with technical help during the preparation of the thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

1.1 Program Testing

Program testing is the process of executing a program using selected test data with the aim of revealing errors in the program. Ideally, one would like to discover all possible errors in a program and correct them. Unfortunately, with the current state-of-the-art in software engineering, finding all possible errors in an arbitrary program is virtually impossible. Consequently, our requirement for programs must shift from the state of perfection to a level of being consistent and acceptable.

Testing is one of the practical approaches to reveal errors in a program, and it is by far the most popular and widely used approach in today's industries. In most software development projects, over 50% of the total cost is spent on testing. Based on this fact, one would expect that testing is a fully developed and well formalized process. On the contrary, to paraphrase Myers[Mye79], "........ less seems to be known about software testing than any other aspect of software development". Testing as done in industry is mostly ad hoc and human factors such as personal experience and intuition play a very important role. Thus, there is a need for more systematic and formal work on program testing. With the availability of a systematic testing strategy, the effect of human interaction on testing will be greatly reduced, thereby speeding up the process of testing.

This thesis is concerned with the automation of a test path generator which utilizes the path analysis testing strategy to detect predicate errors. Our main interest is the vector space model proposed by Zeil[ZeW81]. Based on this model, a computer program called SPTEST has been developed by Sahay[WhP85] to detect predicate errors for a particular predicate in a program. However, SPTEST is an interactive program and considerable user interaction is required when using the program. One of the major objectives of this thesis is to automate the process of test path selection. A review of the vector space model is given and some shortcomings of SPTEST are identified. The concepts of subpath extension are presented and a new heuristic method designed for the system automation are discussed. A new automatic system, SPTEST II, is developed as an extension of SPTEST. Issues involving invariant expressions in Zeil's model are examined and a problem is identified when applying the model to test programs which utilize arrays.

1.2 Predicate Errors and Missing Path Errors

According to Howden's definition[How76], errors in a program can be classified into two major categories: *domain errors* and *computation errors*.

A domain error occurs when incorrect output is generated due to the execution of the wrong path through a program. A computation error occurs when the correct path through the program is taken, but the output is incorrect because of errors in the computations along that path.

Domain errors can occur as a result of an error in the program predicate, or due to an error in some assignment statements which subsequently affect a later predicate; these will be identified as a *predicate error* and an *assignment error* , respectively. It is interesting to note that an assignment error may generate a domain error or a computation error or both.

Missing path errors constitute a subclass of domain errors, and they occur when the required conditional statements do not appear in the given program to be tested.

1.3 Path Analysis Testing

Testing strategies are usually classified into two classes: *black box testing* and *white box testing* . Black box testing approaches devise test data without any knowledge of the software under test or any aspect of its structure, whereas white box testing approaches explicitly use the program structure to develop test data. A white box testing strategy that is most commonly used is known as *path analysis testing* . Since path analysis testing strategies only deal with the existing program structure, no testing strategy in this category can detect missing path errors.

In path analysis testing , the process of testing a computer program consists of two main parts:

1. selection of a path or set of paths along which testing is to be conducted, and
2. selection of input data to serve as test cases which will cause the chosen paths to be executed.

This research focuses only on the first part of path analysis testing; therefore, the issue of test data selection will not be discussed. In addition, the availability of a reliable testing strategy is assumed. A testing strategy is *reliable* if it generates test data to indicate errors whenever the program being tested is incorrect. Since a program could have a potentially infinite number of paths, exhaustively testing all the paths in a program to reveal the errors is unreasonable and impractical. Furthermore, many test paths may not reveal errors, especially when taken in combination with other selected paths. A more reasonable approach is to test only those paths that will produce errors.

In [ZeW81], Zeil has developed a vector space analytical model which represents the set of all possible undetected predicate errors for a particular predicate. Based on this model, the *sufficient path testing strategy* was introduced to select a set of paths to sufficiently test a predicate. A path selection criterion was proposed such that every selected path is capable of revealing some testing information for that predicate. The details of the vector space model are reviewed in Chapter 2.

A computer system called SPTEST, which implements Zeil's model, was developed by Sahay[WhP85] to serve as a tool in the process of test path selection. The objective of SPTEST is to evaluate a given path and determine its usefulness in testing a predicate according to the path rejection criterion. With the experiments conducted on SPTEST, Sahay was able to give some guidelines for selecting test paths to reveal hidden predicate errors. Discussions of SPTEST and those guidelines are given in Chapter 2.

1.4 Extending SPTEST

The current version of SPTEST is only available as a tool to evaluate proposed paths to test program predicates. The user is responsible for providing the test paths which are to be evaluated. Consequently, since the system is running in an interactive mode, the user must spend a significant amount of time to manually conduct the path selection process. There are also some other problems which limit the applicability of SPTEST and are discussed in Chapter 2. Taking those problems into consideration, SPTEST II is developed as an extension of SPTEST in order to reduce the amount of human interaction so that the testing approach can be applied in a more effective way.

1.5 Thesis Objective and Organization

SPTEST is only available as an aid to evaluate the usefulness of proposed test paths in testing a predicate. It gives the user little assistance in generating the test paths. Therefore, a considerable amount of human interaction is required to generate a set of sufficient test paths to test a computer program. The objective of this thesis is focused on the development of a new computer program called SPTEST II, which is an extension of SPTEST, to automatically and systematically conduct the path selection process in order to reduce the amount of human interaction.

The rest of the thesis is organized in five chapters. Chapter 2 summarizes Zeil's model, and the contribution made by Sahay[WhP85] in putting the sufficient path testing strategy into practice with the system SPTEST. The problems of SPTEST are discussed and ideas to solve those problems are proposed. An overview of the extended system SPTEST II is given in Chapter 3. The concepts of subpath extension and the heuristic method for test path generation is introduced. The shortcomings of the heuristic method are discussed. Chapter 4 presents the results of experiments conducted on the extended system, together with an analysis of the performance of SPTEST II. Side issues related to path selection are discussed in Chapter 5. The discussion focuses on invariant expressions in the error space as well as testing programs which utilize arrays. Chapter 6 gives the conclusion of this research project and makes some suggestions for future research.

Chapter 2

Sufficient Path Testing and the System SPTEST

2.1 Background

There are two major classes of variable in a program: *program variables* and *input variables* . Input variables are variables which appear in READ statements. Program variables are variables whose values are obtained from assignment statements directly. The major components of most programs are *computations* and *predicates* , which correspond to assignment statements and logical constructs, respectively. Computations assign new values to program variables, whereas predicates determine the control flow of the execution. A predicate is called a *loop predicate* when it controls the loop iterations in the program.

A computer program can be represented by a *directed graph* (*digraph*) consisting of a set of nodes and arcs, where an arc is a directed line between two nodes. For the sake of simplicity, we will restrict the programs to have a single entry and a single exit. Moreover, we will remove most of the computational details from the program in order to place emphasis on the decision points of a program which determine the control flow.

The digraph representation of a program will contain a node for each occurrence of a predicate, and an arc for each possible flow of control between these nodes. In addition, the digraph should contain exactly one entry node E_n which has no incoming arcs, and should also contain exactly one exit node E_x

which has no arcs leaving it. Moreover, for every node in the digraph, there should exist a sequence of arcs such that this sequence can be traversed in the direction of the arcs from the entry node to that specified node. This sequence of arcs is called a *directed path* . Similarly, for every node in the digraph, there should also exist a directed path from that node to the exit node of the digraph.

A *path* in a computer program is defined to be a directed path from the entry node to the exit node of the digraph. A *subpath* is a directed path which begins at an arbitrary node P_i and ends at an arbitrary predicate P_j in the digraph such that $P_i \neq P_j \neq E_x$. A subpath S which ends at predicate P_i can *reach* predicate P_j if there exists a directed path between their corresponding nodes in the digraph such that $P_i \neq P_j$. Thus, subpath S is called an *extendible subpath* with respect to predicates P_i and P_j. For subpaths which begin at the entry of the program, the specification of the entry node is omitted from the notation in the context of this thesis. The following notations are used throughout this thesis:

Notation: Δ denotes *logical or* ; $|$ denotes *concatenation* ; $*$ denotes *repetitions;* and $==$ denotes *equivalent to*.

Notation: $E_n \mid (T_1 \Delta F_1) \mid (T_i \Delta F_i)^* \mid (T_n \Delta F_n) \mid E_x$ denotes a *path*, where n is the last predicate which leads to the exit of the program and $i = 2....n-1$.

Notation: $(T_i \Delta F_i) \mid (T_j \Delta F_j)^* \mid ?_k$ denotes a *subpath*, where i indicates the starting node of the subpath, k indicates the predicate number to which the subpath leads, and j represent the nodes traversed in the directed path such that $j = i+1 .... k-1$.

For easy reference, the predicates are numbered in the order of their appearance in the text of the program. Paths that consist of different numbers of loop iterations are considered as different paths. The maximum number of paths that exist in a program containing only *n* IF-THEN-ELSE constructs will be 2^n. When a WHILE-DO construct appears in a program, the upper bound on the number of paths cannot be obtained as there are potentially an infinite numbers of paths available. A path is *feasible* if there exists a set of data which causes execution along that path. An *infeasible path* occurs when no input points exist for which that path can be executed. For example, consider the following program:

Example 1: *Program 1.*

```
        READ X, Y
        R = 0
        A = 0
P1      WHILE ( R .NE. 2 ) DO
             A = A + (X - Y)
             R = R + 1
        END WHILE
        PRINT X, Y, A
        STOP
        END                       []
```

In this example, $E_nT_1T_1F_1E_x$ is a path and $T_1?_1$ is a subpath. The paths $E_nT_1F_1E_x$ and $E_nT_1T_1F_1E_x$ are considered as two different paths. The path $E_nT_1F_1E_x$ is an infeasible path as there exists no input data point to execute this path because of the contradiction in predicate P1.

To determine the feasibility of a subpath/path is the same as finding solutions to the set of inequalities that correspond to the predicates encountered along that subpath. Davis[Dav73] has shown that the problem of determining the existence of a solution to a system of inequalities is undecidable; thus, the path feasibility problem is undecidable. However, if the system of inequalities are shown to be linear, the techniques of linear programming can be used to obtain a solution to the system of inequalities.

The expression in a predicate often makes use of variables whose values have been determined by previous computations. It is therefore necessary to know the results of the preceding computations before a predicate can be evaluated. In general, predicates can be expressed in terms of both program variables and input variables. However, in generating input data to satisfy the path condition we must work with constraints in terms of input variables. If we replace each program variable appearing in the predicate by its symbolic value calculated in terms of input variables along that path, we get an equivalent constraint which is called the *predicate interpretation* . A single predicate can appear on many different paths. Since each of these paths will in general consist of a different sequence of assignment statements, a single predicate can have many different interpretations. Consider the following program segment:

```
        READ X, Y
        A = 0
        B = 0
        C = X - Y
        D = X - 2Y
P1      IF ( C .GT. 0 ) THEN
                C = C - D
                A = A + 1
        ELSE
                C = C + D
                B = B + 1
        END IF
P2      IF ( D - C .EQ. B ) THEN
        :
```

If the decision is true on predicate P1, then the predicate P2 is interpreted as follows:

$$\begin{aligned}(D - C \text{ .EQ. } B) &== ((X - 2Y) - ((X - Y) - (X - 2Y)) \text{ .EQ. } 0)\\ &== (X - 2Y) - (X - Y - X + 2Y) \text{ .EQ. } 0\\ &== X - 2Y - Y \text{ .EQ. } 0\\ &== X - 3Y \text{ .EQ. } 0\end{aligned}$$

So the predicate interpretation for (D - C .EQ. B) is (X - 3Y .EQ. 0). On the other hand, if the decision is false on predicate P1, the predicate interpretation for P2 will become the follows:

$$\begin{aligned}(D - C \text{ .EQ. } B) &== ((X - 2Y) - ((X + Y) + (X - 2Y)) \text{ .EQ. } 1)\\ &== (X - 2Y) - (X + Y + X - 2Y) \text{ .EQ. } 1\\ &== X - 2Y - 2X + Y \text{ .EQ. } 1\\ &== -X - Y \text{ .EQ. } 1\end{aligned}$$

For the above program segment, the number of predicate interpretations for predicate P2 is two.

2.2 The Vector Space Model

In [ZeW81], Zeil introduced the vector space model to represent a computer program by a finite-dimensional vector space. The dimension of the vector space is defined to be m+n+1, where m is the number of input variables and n is the number of the program variables used in the program. Expressions in a program can be expressed by vectors from the defined vector space. Each vector component corresponds to a unique variable used in the program. The value of a component represents the variable coefficient in an expression. The extra dimension in the error space is required to accommodate computations which involve constants.

Suppose a program uses two input variables, A and B, and two program variables, C and D. An expression, P1, "D = C - A + 1" can be rewritten to the form "D - C + A - 1 = 0". It is represented by a vector with five components:

$$\begin{array}{ccccc} A & B & C & D & \text{Constant} \\ (\;1 & 0 & -1 & 1 & -1\;) \end{array}$$

By the same token, another expression, P2, "D = C - B" can also be represented by the vector (0 1 -1 1 0). The difference of P1 and P2 is also an expression, namely P1 - P2 = "-A + B + 1 = 0", which can also be represented by a vector (-1 1 0 0 1). Thus, all expressions can be expressed in terms of vectors with a defined number of components.

A vector space provides a powerful way of describing and manipulating an infinitely large set of functions. Two significant properties of vector spaces should be reviewed here. Firstly, any finite-dimensional vector space can be described by specifying a finite set of basis vectors, such that any member of the vector space can be formed as a linear combination of these basis vectors. Secondly, a vector space is closed under the operations of addition and scalar multiplication. It is shown in a later stage that these two properties are useful to describe the undetected errors in predicates.

Suppose that some correct predicate T has been replaced by an erroneous form T'. The expression " T - T' " will then represent the error term in the incorrect predicate. Since the vector space containing T and T' is closed under subtraction, the error term " T - T' " must also be in the same vector space. An error could easily escape detection if the expression " T - T' " is equal to zero. If a reliable method of testing a given predicate is employed, then a predicate error is detectable whenever the interpretation of the incorrect predicate is not a multiple of the correct predicate's interpretation.

Along with this model, Zeil also identified three types of predicate error which are undetectable. He called them *Assignment Blindness* , *Equality Blindness* , and *Self-Blindness* . Furthermore, the property of closure in vector spaces reminds us that any linear combination of these blindness errors, for a particular predicate, is also undetectable. An assignment blindness occurs when an assignment statement " X = f(v) " is executed, where f(v) is the function on a set of variables v, the expression X - f (v) can be added to a predicate without detection because X - f(v) is equal to zero.

Example of Assignment Blindness:

```
Incorrect Segment               Correct Segment
:                               :
A = 1                           A = 1
:                               :
IF ( B + A  .GT. 1 ) THEN       IF ( B .GT. 0 )  THEN
```

An equality blindness occurs in a similar situation with an equality restriction on a selected path.

Example of Equality Blindness:

```
Incorrect Segment              Correct Segment

:                              :
IF ( A .EQ. 2 ) THEN           IF ( A .EQ. 2 ) THEN
:                              :
IF ( A + B .GT. 3 ) THEN       IF ( B .GT. 1 ) THEN
```

This assignment blindness or equality blindness may be removed if a second path is selected such that variable A assumes a different value. Self-blindness occurs in the predicate itself in that the incorrect predicate T' may be a multiple of the correct predicate T, and no mechanism can distinguish the difference because it does not produce a domain error. More concern has been placed on the errors formed by the combinations of the self-blindness and other blindness errors other than the self-blindness itself.

Example of Self Blindness:

```
Incorrect Segment              Correct Segment

:                              :
A = B                          A = B
IF ( A + B - 2 .GT. 0 ) THEN   IF ( A - 1 .GT. 0 ) THEN
```

Although the set of all blindness errors for a program can be infinitely large, it can be represented by a vector space of finite dimension. As we have mentioned above, the dimension of that finite vector space is m+n+1, where m is the number of input variables and n is the number of program variables. Since this vector space will include vectors corresponding to all possible blindness errors, we will call it the *error space* . For a particular predicate p in the program, a subpath s leading to p is selected and has an error space E_s . The error space E_s contains the blindness errors for p after s is used to test the predicate. The dimension of E_s is always (o+e+1) where o is the number of program variables used along p and e is the number of equality restrictions encountered along p. More precisely, the error space E_s has a vector for each program variable used along p, a vector for each equality restriction and also a vector for the self-blindness. The construction of the error space E_s is based on the computations of the subpath p such that the symbolic values of program variables, equality restrictions and self-blindness are expressed by vectors of (m+n+1) components as shown earlier.

Suppose that error space ES_i and ES_j are the respective error spaces after subpath S_i and S_j have been selected to test predicate P_a in a program. A vector V_i in ES_i but not in ES_j indicates that the blindness error, corresponding to V_i, is detectable by subpath S_j. After testing the predicate with both subpaths, the remaining blindness errors are those which correspond to existing vectors in both ES_i and ES_j. Thus, the blindness errors for P_a after using subpaths S_i and S_j correspond to the intersection of ES_i and ES_j.

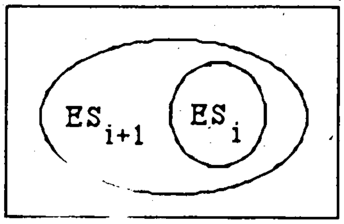
Zeil[ZeW81] also proposed a *path rejection criterion* such that a path is selected if it can reduce the dimension of the error space; otherwise the path is rejected. In other words, a path is only selected if it has the capability of eliminating some blindness errors. Suppose we are testing a predicate P in a program such that the initial error space ES_{init} for P has a dimension of (m+n+1). An initial subpath p_i is used to produce an error space ES_i of dimension k, such that $k < (m+n+1)$. Since the intersection of ES_i and ES_{init} has a smaller dimension than ES_{init} which is of dimension (m+n+1); therefore, p_i is accepted. If a subsequent subpath p_{i+1} is proposed, and it produces an error space ES_{i+1}, one of the following four cases will occur as shown in Figure 1.

It is impossible that the intersection between ES_i and ES_{i+1} is a null set because they always have a common self-blindness vector in their respective error spaces. Knowing that the initial dimension of the error space is (m+n+1) and each selected path will reduce the dimension of the error space at least by one dimension, it can be deduced that the maximum number of subpaths to form a sufficient set for a predicate will be m+n+1.

A set of subpaths is considered to be a *sufficient set* [ZeW81] for a predicate if the failure to detect some errors in that construct, using a reliable testing strategy to select test data for those paths, implies that this error would have gone undetected for any subpath leading to that predicate. *Sufficient path testing* is a path selection strategy to obtain a sufficient set of paths which could test the program predicates for predicate errors.

In sufficient path testing, an initial error space ES_{init} of dimension (m+n+1) is set up for a predicate under test. A current error space ES_{curr}, initially set to ES_{init}, is used to describe the remaining blindness errors after multiple subpaths are selected to test a predicate. A corresponding error space ES_i, with respect to the predicate under test, is constructed for each generated subpath. The error spaceES_{curr} of the same predicate is updated as new subpaths are selected . To determine whether or not a subpath is selected, the intersection ES_{new} of ES_i and ES_{curr} is computed. If the computed intersection ES_{new} has a dimension smaller than the current error space ES_{curr}, the subpath is selected and ES_{curr} will be

Case 1 :



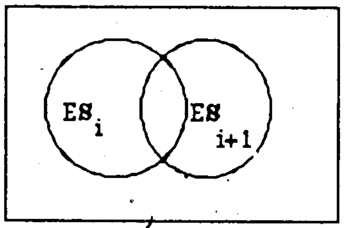
Error Space Intersection

Intersection of ES_i and ES_{i+1} is

equal to ES_i

Then subpath p_{i+1} is rejected

Case 2 :



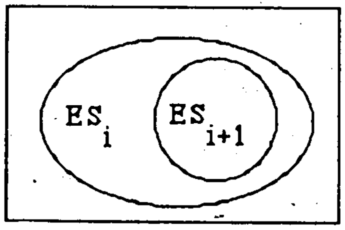
Error Space Intersection

Intersection of ES_i and ES_{i+1} is

not equal to ES_i nor ES_{i+1}

Then subpath p_{i+1} is selected

Case 3 :



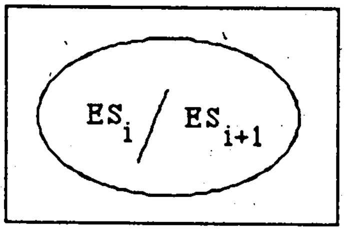
Error Space Intersection

Intersection of ES_i and ES_{i+1} is

equal to ES_{i+1}

Then subpath p_{i+1} is selected

Case 4 :



Error Space Intersection

ES_i is equal to ES_{i+1}

Then subpath p_{i+1} is rejected

Figure 1: Path Rejection / Acceptance Criterion .

replaced by ES_{new}. In other words, the subpath is selected if ES_{new} is a proper subspace of ES_{curr}. By successively replacing the error space ES_{curr} as additional subpaths are selected, the dimension of ES_{curr} will be *reduced* monotonically. This process will terminate when none of the available subpaths can reduce ES_{curr} to a lower dimension and we will call ES_{curr} the *irreducible error space* for that predicate.

Ideally one might assume the dimension of the irreducible error space for a particular predicate to be zero, as we would like to eliminate all the blindness errors in the program. However, we have mentioned earlier that self-blindness cannot be removed no matter how many subpaths are chosen to test the predicate. Thus, the dimension of the irreducible error space is at least one.

The process of subpath selection to obtain an irreducible error space can be looked upon as a search for the appropriate subpaths to form the sufficient set. Search problems are often expressed by trees and a search fails when an inappropriate branch is traversed in the tree. This implies that if an inappropriate selection is made, a global optimum cannot be reached. However, this is not true in subpath selection, as the characteristics of the vector space model give us the property that an irreducible error space can always be obtained if enough subpaths are examined. Suppose an initial subpath S_0 is selected and a current error space ES_{curr} is produced. Any subpath S_i which produces an error space ES_i can be selected into the sufficient set if the intersection of ES_{curr} and ES_i is a proper subspace of ES_{curr}. A subpath S_j which produces an error space ES_j is rejected when the intersection of ES_{curr} and ES_j is not a proper subspace of ES_{curr}. As more subpaths are selected, a current error space ES'_{curr} is obtained. The intersection of ES'_{curr} and ES_j can never be a proper subspace of ES'_{curr} because the intersection of ES_{curr} and ES_j is not a proper subspace of ES_{curr}, and ES'_{curr} is a proper subspace of ES_{curr}. It is clear that the order of subpath selection does not affect the dimension of the irreducible error space and an irreducible error space can always be obtained.

2.3 Basic Assumptions

In this research, we are only interested in a class of programs called *linearly domained programs*. Programs in this class must satisfy the following assumptions:

1. Missing path errors do not occur;
2. The input space is continuous;
3. Predicates are simple, not combined with AND, OR, or other logical operators;
4. Adjacent domains compute different functions;
5. Predicate interpretations are linear in the input variables for both the given predicate as well as the correct predicate.

The first assumption is inherent to the nature of path analysis testing because no testing strategy, based only on the program structure, can guarantee the detection of missing path errors. The second assumption of continuity permits the use of standard mathematical tools. The third assumption simplifies the functional forms of the predicates, but need not actually restrict the set of acceptable programs since any program can be easily transformed to eliminate compound predicates.

Assumption 4 ensures that if a domain error occurs, there must exist some point in at least one of the affected domains which produces incorrect output. The final assumption assures that operations are closed in an appropriate vector space. The assumption of linearity also simplifies the problem of path feasibility. If all the predicate constraints along a path can be shown to be linear in the input variables, then the feasibility problem will become decidable since the technique of linear programming can be employed to obtain a solution of the linear constraints.

2.4 The Sufficient Path Testing System SPTEST

SPTEST is a computer system, implemented by Sahay[WhP85], which obtains sufficient test paths for selected predicates. The purpose of SPTEST is to evaluate a proposed path for testing a given predicate by obtaining the reduction in the predicate error space due to that path. SPTEST does not depend upon any testing strategy, nor does it assume that a particular testing strategy will be used to test program predicates. It simply selects a set of test paths that is appropriate for testing predicates in a computer program. Once a set of paths is selected, any reliable testing strategy can be used to perform the actual testing along the selected paths.

An *invariant expression* is an algebraic expression which can be added to a program statement without being detected along all possible paths through that statement. An *unused variable* occurs when a program variable is initialized, but is not used in any of the paths. In [ZeW81], Zeil has pointed out that the set of undetectable predicate errors can be classified into three types of blindness errors: assignment blindness, equality blindness, and self-blindness. From the experimental results on SPTEST, Sahay has observed that it is not unusual for the irreducible error space to have a dimension well beyond one. His experiments also confirmed that the irreducible error space contains those blindness errors mentioned by Zeil, In addition, he has characterized that an irreducible error space contains vectors corresponding to unused variables, equality restrictions, invariant expressions and self-blindness. It is expected that vectors corresponding to unused variables and equality restrictions would appear in the error space as they are related to assignment blindness and equality blindness. In [ZeW81], Zeil stated that "..... one consequence of utilizing this testing criterion is that eventually all invariant linear relations for a program will be derived at each predicate." Thus, the presence of invariant expressions in the error space should not surprise us at all. Invariant expressions will be discussed further in Chapter 5.

Based on the results obtained from his experiments, Sahay[WhP85] has provided us with considerable insight into the process of path selection. He has proposed the following heuristics to assist us in selecting paths:

1. In testing any predicate, choose the first path such that it has the least equality restrictions.

2. By utilizing common subpaths for testing subsequent predicates and by first evaluating predicates which occur early in the program, the total number of paths required for testing all the predicates in a program can be substantially reduced.

3. If possible, select additional paths which employ unused program variables after selecting the first path.

4. For any loop in the program, test only those paths which execute that loop with no more than m+n iterations, where m is the number of input variables and n is the number of program variables.

2.5 Problems with SPTEST and Proposed Solutions

SPTEST runs in an interactive mode with user-specified decisions on all predicates encountered along the subpath. The usefulness of SPTEST is limited by the following major problems:

1. *Time-consuming Process* :

   During the selection of a test path, the user has to enter the predicate decisions, one at a time, at the terminal. Thus, the process is very tedious and time-consuming . It has been demonstrated in reference [WCL87] that the time spent on manually selecting paths is much more significant than the time spent on the process execution.

2. *Reducibility Detection* :

   SPTEST only informs the users that a sufficient set of subpaths has been obtained when the final error space is of dimension one. It makes no attempt to provide a termination indicator when the irreducible error space has been obtained, and has a dimension greater than one.

3. *Feasibility Detection* :

   The system is incapable of detecting infeasible paths. Since it makes no decisions in choosing subpaths, users have to be very familiar with the program being tested in order to specify feasible subpaths.

4. *Extensibility of Subpaths* :

   SPTEST provides us with sets of sufficient subpaths for each predicate in the program. To test the entire program, we have to use some scheme to collect these fragments and extend them to form test paths for all predicates in the program. There exists a close relationship between feasible paths and subpath extensions. It is clear that to extend a subpath to the next predicate, the resulting subpath must be feasible in order to be useful.

As a result, the system requires a lot of human intervention and insight to be effective, therefore, it serves only as an aid in selecting subpaths.

In an attempt to provide solutions to the above problems, the extended system SPTEST II will incorporate those heuristics proposed by Sahay, and automate the whole path selection process. The following approaches serve as guidelines for this research project as well as solutions to the problems mentioned above:

1. To automate the process, the system must be capable of identifying the terminating condition. The terminating condition occurs when the error space obtained is irreducible for each predicate in the program. Sahay has established the categorization of the vector components in an irreducible error space as invariant expressions, unused variables, equality restrictions

and self-blindness. In order to determine the reducibility of an error space, we must be able to recognize each vector in the error space according to these four categories. Vectors corresponding to unused variables, self-blindness and equality restrictions can be identified by keeping track of the computation process at the program statement level and all the predicates encountered along the subpaths. Since our knowledge concerning invariant expressions is limited, heuristics are developed to determine if an error space is irreducible.

2. In selecting subpaths, we have to make sure that the selected subpaths are feasible. Since the predicates in the program are linear, we can group all the predicates which are encountered along the subpath and feed this information into a linear programming problem solver. If the solver returns a solution to the corresponding linear inequality system, then the selected subpath is feasible; otherwise, the solver will indicate the subpath is infeasible and should not be used.

3. To form test paths to test all the predicates in the program, we can extend the subpaths to the next predicate. Eventually, we will reach the last predicate in the program, thus forming test paths for an entire program. However, this method does not work when extendible subpaths do not exist in the program. When a set of sufficient extendible subpaths cannot be obtained from the previous sufficient set, additional test paths are normally required to achieve the same error space reduction.

Chapter 3

The Automatic Test Path Generator

3.1 System Components of SPTEST II

SPTEST II is an automatic test path generator which generates test paths according to the sufficient path criterion for computer programs. An overall structure of the system SPTEST II is presented in Figure 2.

The input for this system is a linearly domained ANSI FORTRAN program. Since the objective of the system is to generate paths to detect predicate errors only, the input program is assumed to be syntax error free, and the system will not perform any syntax error checking on the input. SPTEST II is written in FORTRAN 77 and presently runs on UNIX.

SPTEST II consists of five phases:

1. Parse, Compile and Tokens Scanning
2. Path Generation
3. Symbolic Execution
4. Feasibility Test and Sufficient Path Testing
5. Subpath Collection and Completion

The phases for program parse and compilation, symbolic execution and sufficient path testing are the same as those in SPTEST. Minor modifications are made in order to change them into independent procedures to interact with other phases in SPTEST II.

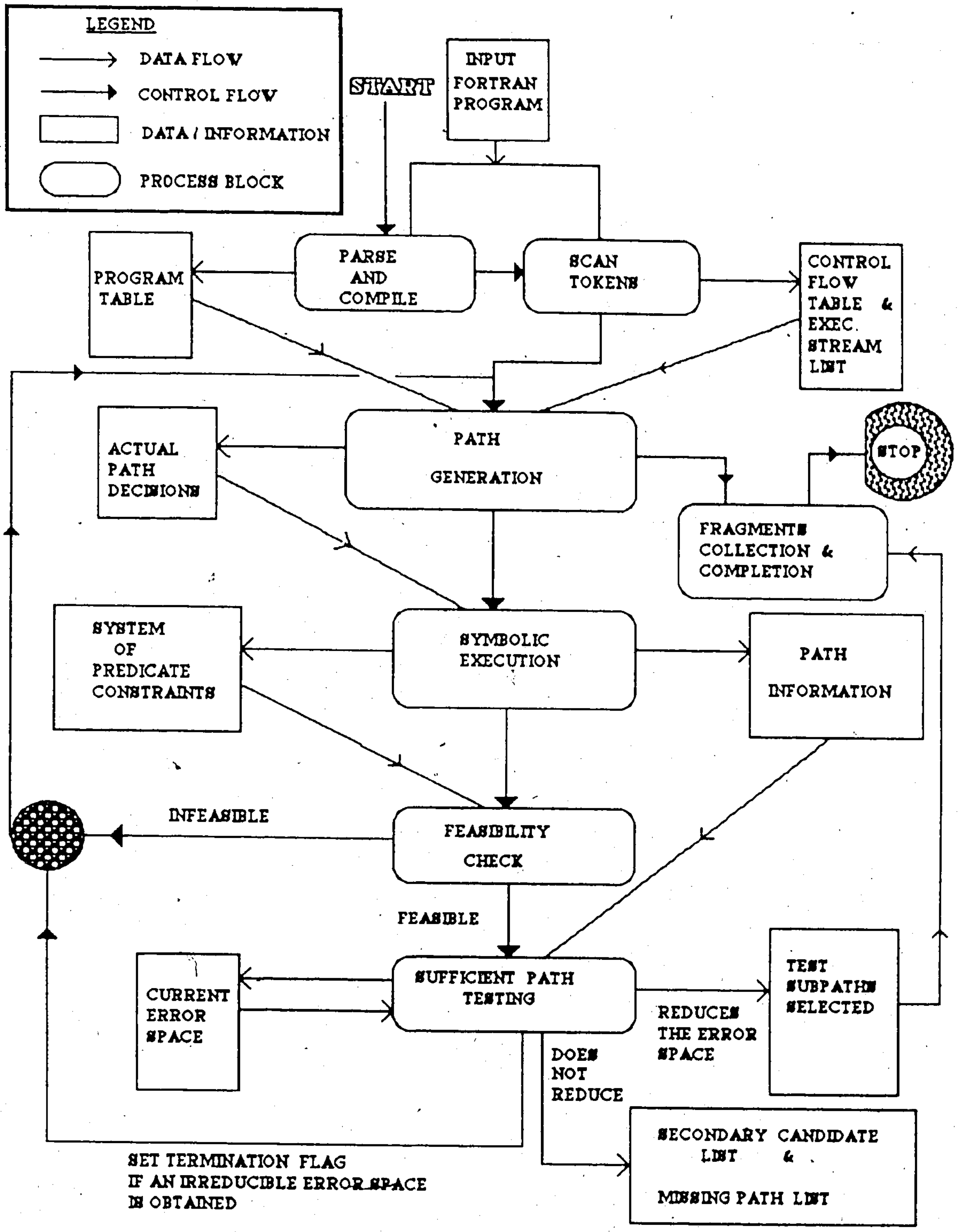


Figure 2: An Overview of the System SPTEST II

3.1.1 Program Parse, Compile and Token Scanning

The first phase of SPTEST II is a parse of the input program. It constructs the symbol table which differentiates input variables and program variables. Predicate lists and other control information is secured from this parse. Arithmetic expressions for either assignment statements or predicate expressions are stored as binary trees. The compilation program determines the linearity of all assignments and predicate expressions. The process of token scanning constructs a simple predicate control flow table which is used to determine the reachability of subpaths, and a list of decision options within each loop predicate. The decision option list is later used in generating subpaths which involve iterations.

3.1.2 Path Generation

Path generation is the second phase of SPTEST II. For each predicate in the input, it generates test subpaths to test the predicate until the predicate is sufficiently tested or when the system limits are exceeded. The path generation process operates under the concepts of subpath extension, information provided by the control flow table, and the loop decision options list. Further discussion on path generation will be presented in subsequent sections.

3.1.3 Symbolic Execution

The symbolic execution of a subpath is carried out by evaluating the sequence of computations and predicates encountered along the subpath. Predicate interpretations for all the predicates that are encountered along the subpath are obtained. These interpretations are stored in a file which serves as input for the next phase. To demonstrate the process of symbolic execution, let us consider the following example with a subpath $T_1T_2T_3T_4F_4?_3$:

Example 2: *Program 2.*

```
C
C  A program to find the remainder of X divided by Y
C
       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             []
```

The following substitutions will be made during the process of symbolic execution:

Line	Symbolic Execution
2	R = 0
3	A = 0
4	(X .GE. 0)
5	(Y .GT. 0)
6	R = X
7	(X .GE. Y)
8	A = Y
9	(X .GE. Y)
10	R = X - Y
11	A = Y + Y
9	(X - Y .LT. 2*Y)
7	?

At this point, variables R and A have symbolic values of "X-Y" and "2*Y", respectively. The predicate interpretations on the subpath are:

(X .GE. 0)
(Y .GT. 0)
(X .GE. Y)
(X .GE. Y)
(X - Y .LT. 2*Y).

The predicate interpretations will serve as the linear constraints specified by the subpath $T_1T_2T_3T_4F_4?_3$.

3.1.4 Feasibility Test and Sufficient Path Testing

The feasibility of a subpath is determined in this phase, and the sufficient path testing is conducted after the subpath is found to be feasible. The overall objective of this phase is to determine the usefulness of a test subpath according to the path acceptance/rejection criterion. A subpath is selected to test a predicate if it is feasible and reduces the error space of that predicate to a smaller dimension. The constraint set of a subpath can be obtained by performing a predicate interpretation on the predicates along the subpath. By utilizing a linear programming problem solver (LP solver) to obtain a solution to the constraint set, we can determine the feasibility of the subpath. If a subpath is found to be infeasible, it will be rejected, and sufficient path testing will not be performed on that subpath. Among the restrictions specified by the package, two of them affect the performance of SPTEST II:

1. the inequality constraints must be in the form of either .LE. or .EQ., and
2. all the unknowns must have non-negative values.

We often encounter program predicates with .NE. constraints. In addition, there is no guarantee that input to a program will be non-negative. The first mentioned restriction affects the system in terms of efficiency because any non-equality constraint must be converted into two strong inequalities. The unspecified interpretations of non-equality constraints force us to examine all the combinations of replacement by strong inequalities before we can determine whether a subpath is infeasible. The worst case to conclude that a subpath is infeasible is $O(2^n)$, where n is the number of non-equality constraints in the constraint set.

Suppose we have a constraint set as follows:

A .NE. B
A + 1 .GT. B
B .LE. 5
A - B .NE. C
C .GT. 2

To determine if the constraint set is infeasible, we will have to look at all four possible interpretations that the non-equality constraints in the set can assume.

1.	A .GT. B A + 1 .GT. B B .LE. 5 A - B .GT. C C .GT. 2	2.	A .LT. B A + 1 .GT. B B .LE. 5 A - B .GT. C C .GT. 2
3.	A .GT. B A + 1 .GT. B B .LE. 5 A - B .LT. C C .GT. 2	4.	A .LT. B A + 1 .GT. B B .LE. 5 A - B .LT. C C .GT. 2

In addition, the strong inequalities have to be modified to become weak inequalities .LE., and these extra computations turn out to be a significant factor which affects the processing time.

The second restriction mentioned will affect the process of path selection. The reliability of the computation of the LP solver is not applicable when unknowns in the constraint set are forced to have negative values by the generated subpath. Therefore, the constraint sets are checked before it is passed to the LP solver. Any occurrence of unknowns having negative values in a constraint set will be considered as an infeasible set. Thus, it reduces the number of potential test paths in the path generation process.

Sufficient path testing is performed on feasible subpaths only. As mentioned earlier, since the self-blindness for a predicate can never be removed from its error space, we decided to exclude the self-blindness vector from the error space calculation in order to reduce the processing time. Subpaths which do not reduce the current error space will be stored in the system for possible reference in a later stage.

3.1.5 Subpath Collection and Completion

This is the last phase of the system SPTEST II. After we have obtained a set of sufficient test subpaths for the last predicate in the input program, we have to collect all the subpaths which are relevant for testing the predicates in the program, and complete them to form test paths. The set of test paths is stored in a file for the use of test data generation.

3.2 Concepts of Subpath Extension

One of the major problems associated with SPTEST is the extensibility of subpaths to form test paths. SPTEST does not have to deal with this problem directly since it is an interactive system which accepts subpaths given by the user, and conducts sufficient path testing. The attempt to generate the minimum sufficient set of paths to test a program using SPTEST brings us the idea of "re-using the subpaths". The idea described here can only apply to *structured programs* , and as we have mentioned in Section 2.1, the predicates are numbered in the order of their appearance in the text of the program.

The concept of subpath extension is based on the fact that information is "re-useable" in testing predicates because of the relationship held by consecutive predicates in the program. The predominant idea is to extend subpaths that end at predicate i, which are sufficient to test predicate i in the input program, to test predicate i+1. With this approach, we are able to formulate a heuristic method for test path generation. Prior to the layout of the heuristic method, several definitions shall be introduced.

3.2.1 Definitions and Basic Concepts

An *illegal flow error* occurs when an extended subpath does not reach the desired predicate. Instead, it reaches some other predicate and creates no conflict in terms of symbolic execution. There are two types of illegal flow errors, *illegal flow type 1 error* and *illegal flow type 2 error* . An illegal flow type 1 error occurs when the subpath reaches an untested predicate, while an illegal flow type 2 error occurs when the subpath reaches a previously tested predicate.

A *candidate list* contains a set of subpaths which are generated from the subpath extension method for testing the predicate in a program. There are three kinds of candidate lists: the *prime candidate list* , the *secondary candidate list* , and the *missing path candidate list* . Whenever a subpath is generated, it has to go through the tests on reachability and feasibility, and must satisfy the path acceptance criterion in order to determine its usefulness. The three candidate lists reflect the usefulness of the subpaths constructed by the method. The prime candidate list contains subpaths which reduce the dimension of the error space for the current predicate being tested. When a subpath produces an illegal flow type 1 error and is part of the sufficient test path set for some previous predicate, it is stored in the secondary candidate list. Subpaths in the secondary candidate list will not be needed to test the current predicate but are required to test some previous predicate in the program; they may also be needed to test some subsequent predicates. In order to reduce the number of test paths in the final test set, each subpath in these two categories has a *status word* to indicate its applicability.

Prime candidate list entries having a status word of 1 are subpaths which are needed to test the current predicate; subpaths which are carried over from the previous tested predicate will have a status word of 2. A status word of 1 in the secondary candidate list indicates a subpath has committed an illegal flow error when testing predicate i, but is needed to test some other predicate(s) prior to predicate i. Eventually, it will be needed in the final test path set. A subpath in this category having a status word of 2 indicates it produces an illegal flow error when testing predicate i, but one of its extended versions is already kep[illegible] the prime candidate list. If these subpaths do not contrib[illegible] to the error space reduction at a later stage, they can be discarded. The reason for retaining them is to provide entry points to some future predicates in case these subpaths provide the only feasible way to get to subsequent predicates. Infeasible paths will not be stored in any of the candidate lists.

A missing path candidate list contains subpaths which do not reduce the error space at the time they were generated. They will serve as the final alternatives to test some future predicates when needed. Once a subpath is entered into the missing path candidate list, it will be extended to the current testing predicate after the current predicate has been sufficiently tested. Subpaths in the missing path candidate list may generate infeasible paths after extension because we only check their feasibilities when it's required. When an extended subpath in the missing path candidate list produces an illegal flow error, that subpath will be deleted from the list.

A *loop path list* is a list where we keep all subpaths which have ended at a loop predicate, and these subpaths will be used for loop iteration extensions. A *work space* is a working area which we use to transfer subpaths to and from the collection. The *final collection* provides storage for the subpaths when they reach the exit of the program instead of any other predicates during the extension. They will constitute the test paths for the program.

A *control flow table* contains entries which indicate the destinations of both true and false options for each predicate. With this table, a simple table-look-up method can be used to distinguish paths which produce illegal flow errors.

```
READ M, N
A = M
B = N
C = M - N
D = 0
IF ( A .GT. 9 ) THEN                T1
        D = 8
ELSE
        D = M
END IF
WHILE (.......) DO                  T2
        A = B - 1
        C = C - 1
        IF (.....) THEN DO          T3
                D = ......
        ELSE DO
                E = ......
                END IF
        END WHILE
:
:
STOP
END
```

For example, the program structure above will have a control flow table as follows:

Control Flow Table:

Predicate	**True**	**False**
1	2	2
2	3	0
3	2	2

Note : a zero entry in the table represents the exit of the program.

A single loop in a test program is sufficient to create a potentially infinite number of paths for us to examine. If we were allowed to test all the possible paths in a program, the process would never end as there are an infinite number of paths for us to examine. Therefore, loop predicates are entitled to special attention in this heuristic method. A *loop pattern* is defined to be a directed path which starts at node i where node i corresponds to a loop predicate and ends at node j where node j has an arc leading to node i. Every loop predicate in the program has its own set of loop patterns which is formed by making all possible combinations of decisions for the embedded predicates. In other words, the set of loop patterns contains all the possible directed paths in a single iteration from node i to node j. Consider the following program segment:

```
          :
P3      WHILE ( ....... ) DO
P4              IF ( ..... ) THEN
                        :
                ELSE
P5                      IF ( ..... ) THEN
                                :
                        ELSE
                                :
                        END IF
                END IF
        END WHILE
          :
```

In this program, F_4T_5 is one of the loop patterns in the loop for predicate P3.

There can be many different loop patterns within a loop but not all of them are feasible. In the heuristic method, a subpath to test a loop predicate is extended by one iteration at a time. This will give us the advantage of forming a relatively small set of loop patterns before the subpath extension begins. To perform the extension with one extra iteration, the original subpath is used as the source and appended by different loop patterns to form a set of new extended subpaths. This idea is easily illustrated by the following example:

Example 3 : *Program 3.*

```
    :
    WHILE (........) DO                 T3
            IF (........)  THEN         T4
                    :
            ELSE
                    :
            END IF
            IF (......) THEN            T5
                    IF (...........) THEN  T6
                            :
                    END IF
            END IF
            END WHILE
    :                                   []
```

In this example, the set of loop patterns for predicate T3 are :

$T_3T_4T_5T_6$
$T_3T_4T_5F_6$
$T_3T_4F_5$
$T_3F_4T_5T_6$
$T_3F_4T_5F_6$
$T_3F_4F_5$

Assuming that the original subpath is $T_1F_2?_3$, then the set of new extended subpaths are shown as follows:

$T_1F_2T_3T_4T_5T_6?_3$
$T_1F_2T_3T_4T_5F_6?_3$
$T_1F_2T_3T_4F_5?_3$
$T_1F_2T_3F_4T_5T_6?_3$
$T_1F_2T_3F_4T_5F_6?_3$
$T_1F_2T_3F_4F_5?_3$

This idea is important to us in terms of generating paths through iterations. We can extend the subpath, which ended at a loop predicate, with one of the loop patterns without making arbitrary decisions on the predicates encountered within the loop. When nested loop predicates are found, a zero iteration is placed on the nested loop predicates and then the nested loop is iterated once every time until it reaches the system limits. The following is a more complicated example of setting up the loop patterns which involves a nested loop:

Example 4: *Program 4*.

```
:
WHILE (.......) DO                              T3
      IF (........) THEN                        T4
                 :
      ELSE
                 :
      END IF
      WHILE (.........) DO                      T5
            IF (...........) THEN               T6
                       :
            END IF
            END WHILE
      END WHILE
:
```

[]

In the above example, the loop patterns for predicate T3 (with no iteration on the inner loop) are:

$T_3T_4F_5$
$T_3F_4F_5$

and the loop patterns for predicate T5 are:

T_5T_6
T_5F_6.

Again if we assume that the original subpath is $T_1F_2?_3$, the set of new extended subpaths for predicate T3 with no inner loop iteration and one nested loop iteration will look as follows:

Subpaths with no inner loop iteration :	$T_1F_2T_3T_4F_5?_3$
	$T_1F_2T_3F_4F_5?_3$.
Subpaths with one inner loop iteration :	$T_1F_2T_3T_4T_5T_6F_5?_3$
	$T_1F_2T_3T_4T_5F_6F_5?_3$
	$T_1F_2T_3F_4T_5T_6F_5?_3$
	$T_1F_2T_3F_4T_5F_6F_5?_3$.

It is interesting to note that not only can the loop predicates be extended by using the loop patterns, but any predicate that is embedded in loop predicates can also be extended as well. Consider the following example:

Example 5: *Program 5.*

```
:
WHILE ( .......) DO                  T3
      IF (.......) THEN              T4
            :
      END IF
      IF (......) THEN               T5
            :
      END IF
      END WHILE
:
```

[]

The loop patterns for T3 are: $T_3T_4T_5$
$T_3T_4F_5$
$T_3F_4T_5$
$T_3F_4F_5$.

However, these loop patterns can be wrapped around to form loop patterns for T4 as shown below:

$T_4T_5T_3$
$T_4F_5T_3$
$F_4T_5T_3$
$F_4F_5T_3$.

Any subpath which tests predicate T4 can be extended by one of these loop patterns whenever extra iterations are needed. This special property adds a certain degree of versatility to loop extensions. The loop patterns for each loop predicate in the program can be constructed and stored before the process of test path generation. When a subpath requires loop extensions, a simple concatenation of the loop patterns to the current subpath is performed, and a new set of potential test subpaths are available for examination.

3.2.2 Heuristic Method for Test Path Generation

Subpath extension and loop extension are the two major operations in the heuristic method. An initial subpath is generated to test the first predicate and loop extension is performed when the predicate is a loop predicate. Subpaths are stored in different lists according to their usages. When an irreducible error space has been obtained, the method will proceed to test subsequent predicates. For each subsequent predicate, subpath extension is performed on subpaths from the prime candidate list, the secondary candidate list and the missing path candidate list until an irreducible error space is obtained, and loop extension is considered when necessary. After all the predicates are sufficiently tested, subpaths from the sufficient set are extended to form test paths. The method will stop generating subpaths to test a particular predicate when previous information is insufficient to obtain the irreducible error space by loop extension and subpath extension. The process will continue to test subsequent predicates until the last predicate in the program has been tested.

The heuristic method can be described in seven steps. For each predicate in the input program, apply step one to step six until termination. After applying these six steps to the last predicate in the input program, the method terminates with step seven. The path generation process on a predicate may stop at any step when the corresponding error space becomes irreducible. The details of the heuristic method is presented in a psuedo code format as follows:

Step 1 :

```
if the prime candidate list is empty then
{it must be the first subpath to be selected}
  generate subpath "?" and calculate the error space;
  add this subpath into the prime candidate list;
  goto Step 2;
else
  goto Step 3;
end if
```

Step 2 :

```
if the current predicate is a loop predicate then
  do a loop extension on the subpath;
  goto Step 5;
else
  {done with this predicate}
  if this is the last predicate in the input program then
    goto Step 7;
  else
    goto Step 1 and test the next predicate;
  end if
end if
```

Step 3 :

```
copy the prime candidate list to the work space;
empty the prime candidate list;
for each subpath i in the work space do
  replace the "?" in subpath i by both "T?" and "F?" to form subpaths a and b,
                    respectively;
  select action according to the action table shown in Table 1;
end for
If the dimension of the error space is not zero then
  empty the work space;
  select subpaths from the secondary candidate list to the work space if they can
                    reach the testing predicate ( according to the control flow table );
  for every subpath i in the work space do
    append a "?" to subpath i and assign it to a;
    assign a null path to b;
    select action according to the action table shown in Table 1;
    exit the FOR loop when error space is of dimension zero;
  end for
end if
  goto Step 4;
```

b \ a	FEASIBLE & REDUCES	FEASIBLE BUT NOT REDUCES	INFEASIBLE	ILLEGAL FLOW TYPE 1	ILLEGAL FLOW TYPE 2	EXIT
FEASIBLE & REDUCES	1 (1) & 2 (1)	2 (1) & 5	2 (1)	2 (1) & 3 (2)	2 (1)	2 (1)
FEASIBLE BUT NOT REDUCES	1 (1) & 6	1 (2) & 6	2 (2)	3 (1)	2 (2)	2 (2)
INFEASIBLE	1 (1)	1 (2)		3 (1)	3 (1)	7
ILLEGAL FLOW TYPE 1	1 (1) & 4 (2)	4 (1)	4 (1)	3 (1) & 4 (2)	4 (1)	7
ILLEGAL FLOW TYPE 2	1 (1)	1 (2)	4 (1)	4 (1)	10	7
EXIT	1 (1)	1 (2)	8	8	8	7
NULL	1 (1)	1 : 3 (1) 2 : 5	9	1 : 3 (1) 2 : 5	1 : 3 (1) 2 : 5	

Notations :

n (i) means action n should be taken and status i is assigned to the path
n means action n is taken and no status is assigned
i : n (j) means if the previous status is i then action n is taken and status j is assigned to the path
i : n means if the previous status is i then action n is taken

Action Code:

1: add subpath "a" to the prime candidate list
2: add subpath "b" to the prime candidate list
3: add subpath "a" to the secondary candidate list without the "?"
4: add subpath "b" to the secondary candidate list without the "?"
5: add subpath "a" to the missing path candidate list without the "?"
6: add subpath "b" to the missing path candidate list without the "?"
7: add subpath "a" to the final collection without the "?"
8: add subpath "b" to the final collection without the "?"
9: ignore the subpath
10:change "?" in both subpaths to "F?" and repeat the

Table 1 : Ac Table for Path Selection.

In Table 1, there are two cases that cannot occur:

1) both subpaths are infeasible when extended,

2) subpath *a* extended to EXIT and subpath *b* is a null subpath.

Case 2 cannot happen because when subpath *b* is a null path, we know that subpath *a* must be selected from the secondary candidate list. However, for subpaths which have been extended to EXIT and are used to test some predicates in the program, they will be selected and stored in either the prime candidate list or the final collection. Therefore it is impossible to have the existence of case 2.

It is true for both loop predicates and non-loop predicates that case 1 cannot occur. This can be explained by the following lemma:

lemma :

Given a structured program and a feasible subpath p which ends at predicate i. Let the true branch of i go to predicate j_1, thereby forming subpath p_1; let the false branch of i go to predicate j_2, thus forming subpath p_2. Either j_1 and j_2 could be the exit of the program. Then either p_1 or p_2 must be feasible.

proof :

We know that p is feasible and the constraint set p_c along p must also be feasible. Assume p_1 is infeasible, and the true branch we made on predicate i which adds the constraint i_c to p_c to form the constraint set p_{1c} along p_1 must have created the infeasibility. Therefore p_2, which took the false branch, must be feasible because the constraint set p_{2c} along p_2 contains p_c and the inverse of i_c which are both feasible. Similarly, if p_2 is infeasible, p_1 must be feasible. []

Step 4:

```
if the dimension of the error space is not zero then
  if the testing predicate is within a loop and not a loop predicate then
    find the first outer loop predicate j in which the current predicate is embedded;
    find the loop patterns for loop predicate j;
    form wrap-around loop patterns for the current predicate;
    copy the prime candidate list to the loop path list;
*** for each subpath i in the loop path list do
      form all possible extension with the wrap-around loop patterns using subpath
          i as the source and put them in the work space;
      end for
    empty the loop path list;
    for each subpath i in the work space do
      perform feasibility test and sufficient path test on subpath i to determine its
                        usefulness;
      if subpath i is selected then
        add subpath i to the prime candidate list;
        add subpath i to the loop path list;
        exit the FOR loop if error space is of dimension zero;
      else if subpath i is feasible then
              add subpath i to the missing path candidate list;
              add subpath i to the loop path list;
          end if
      end if
    end for
    if the dimension of the error space is not zero and number of
              iteration performed is not greater than the system limit then
      goto ***;
    else
      goto Step 6;
    end if
  else
    if the testing predicate is a loop predicate then
```

```
        goto Step 5;
      end if
    end if
  end if
  goto Step 6;
```

Step 5 :

```
    empty the work space;
    copy the prime candidate list to the loop path list;
    find the loop patterns for the predicate;
*** for each subpath i in the loop path list do
      form all possible extensions with the loop patterns using subpath i as the source
           and put them in the work space;
    end for
    empty the loop path list;
    for each subpath i in the work space do
      perform feasibility test and sufficient path test on subpath i;
      if subpath i is selected then
        add subpath i to the prime candidate list;
        add subpath i to the loop path list;
      else if subpath i is feasible then
             add subpath i to the missing path candidate list ;
             add subpath i to the loop path list;
           end if
      end if
    exit the FOR loop whenever the error space is of dimension zero;
    end for
    if the dimension of the error space is not zero and number of iterations performed
           is not greater than the system limit then
      goto ***;
    else
      goto Step 6;
    end if
```

Step 6 :

```
if the dimension of the error space is zero then
  if the testing predicate is the last predicate in the program then
    goto Step 7;
  else
    goto Step 1 and test the next predicate;
  end if
else
  copy the missing path candidate list to the work space;
  empty the missing path candidate list;
  for each subpath i in the work space do
    if subpath i leads to the testing predicate then
      if subpath i is feasible then
        if subpath i reduces the error space then
          add subpath i to the prime candidate list;
          if the dimension of the error space is zero then
            if the current predicate is the last predicate in the program then
                goto Step 7;
            else
                goto Step 1 and test the next predicate;
            end if
          end if
        else
          add subpath i to the missing path candidate list;
        end if
      end if
    else
      add subpath i to the missing path candidate list;
    end if
  end for
end if
```

```
    { At this point, the system has explored all the available alternatives in hand
including subpaths in the missing path candidate list, no matter what the error space
looks like, the system has run out of subpaths to examine. }
if the current predicate is the last predicate in the input program then
   goto Step 7;
else
   goto Step 1 and test the next predicate;
end if
```

Step 7 :

```
empty the work space;
transfer all the subpaths in the prime candidate list and subpaths having status 1 in
         the secondary candidate list to the work space;
for each subpath in the work space do
   complete the subpath until it reaches EXIT with appropriate feasible predicate
              decisions;
   add the test path to the final collection;
end for
```

We will demonstrate the process of the heuristic method for test path generation briefly by the following example which finds the Euclid GCF:

Example 6: *Program 6.*

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

The Control Flow Table for this program looks like the following:

Predicate	True	False
1	2	3
2	1	1
3	0	0

The loop patterns for predicate 1 are:

T_1T_2
T_1F_2

Initially, the Prime Candidate List (P), Secondary Candidate List (S), Missing Path Candidate List (M), Loop Path List (L) and the Final Collection (F) are empty. The original error space (ES) is of dimension 6 and the work space (WS) is also empty.

Testing Predicate 1:

P	S	M	L	ES	E
---	---	---	---	6	---

[WS:

$?_1$ reduces the ES to 4, add "$?_1$" to P]

P	S	M	L	ES	E
$?_1$	---	---	---	4	---

[WS:

Predicate 1 is a loop predicate, thus do a loop extension.
Copy P to L
Fetch the loop patterns for Predicate 1
Formed subpaths: $T_1F_2?_1$
$T_1T_2?_1$

$T_1F_2?_1$ reduces the ES to 3, add "$T_1F_2?_1$" to P

$T_1T_2?_1$ reduces the [illegible] to 1, add "$T_1T_2?_1$" to P

Current ES contains the self-blindness vector only, thus sufficient paths obtained for Predicate 1]

P	S	M	L	ES	E
$?_1$ $T_1F_2?_1$ $T_1T_2?_1$	---	---	$?_1$	1	---

Testing Predicate 2:

P	S	M	L	ES	E
$?_1$ $T_1T_2?_1$ $T_1F_2?_1$	---	---	---	6	---

[WS:

Copy P to WS
{ $?_1$

$T_1T_2?_1$
$T_1F_2?_1$ }

$?_1 \Rightarrow$ $F_1?_2$
$T_1?_2$

$F_1?_2$ $T_1?_2$	an illegal flow type 1 error occurs reduces the [illegible] add "$T_1?_2$" to P and add "F_1" to S
$T_1F_2?_1 \Rightarrow$	$T_1T_2F_1?_2$ $T_1T_2T_1?_2$
$T_1T_2F_1?_2$ $T_1T_2T_1?_2$	an illegal flow type 1 error occurs reduces the ES to 3 add "$T_1T_2T_1?_2$" to P and add "$T_1T_2F_1$" to S
$T_1F_2?_1 \Rightarrow$	$T_1F_2F_1?_2$ $T_1F_2T_1?_2$
$T_1F_2F_1?_2$ $T_1F_2T_1?_2$	infeasible path reduces the ES to 1 add "$T_1F_2T_1?_2$" to P

Current ES contains the self-blindness vector only, thus sufficient paths obtained for Predicate 2]

P	S	M	L	ES	F
$T_1?_2$	F_1	---	---	1	---
$T_1T_2T_1?_1$	$T_1T_2F_1$				
$T_1F_2T_1?_2$					

Testing Predicate 3:

P	S	M	L	ES	F
$T_1?_2$	F_1	---	---	6	---
$T_1T_2T_1?_1$	$T_1T_2F_1$				
$T_1F_2T_1?_2$					

[WS:

Copy P to WS
{ $T_1?_2$

$T_1T_2T_1?_2$
$T_1F_2T_1?_2$ }

$T_1?_2$=> $T_1F_2?_3$
$T_1T_2?_3$

$T_1F_2?_3$	an illegal flow type 2 error occurs
$T_1T_2?_3$	an illegal flow type 2 error occurs

change "?" to "F?"

$T_1F_2F_1?_3$	infeasible path
$T_1T_2F_1?_3$	reduces the ES to 5, add "$T_1T_2F_1?_3$" to P

$T_1T_2T_1?_2$=> $T_1T_2T_1F_2?_3$
$T_1T_2T_1T_2?_3$

$T_1T_2T_1F_2?_3$	an illegal flow type 2 error occurs
$T_1T_2T_1T_2?_3$	an illegal flow type 2 error occurs

change "?" to "F?"

$T_1T_2T_1F_2F_1?_3$	infeasible path
$T_1T_2T_1T_2F_1?_3$	reduces the ES to 4, add "$T_1T_2T_1T_2F_1?_3$" to P

$T_1F_2T_1?_2$=> $T_1F_2T_1F_2?_3$
$T_1F_2T_1T_2?_3$

$T_1F_2T_1F_2?_3$	an illegal flow type 2 error occurs
$T_1F_2T_1T_2?_3$	an illegal flow type 2 error occurs

change "?" to "F?"

$T_1F_2T_1F_2F_1?_3$	infeasible path
$T_1F_2T_1T_2F_1?_3$	reduces the ES to 3, add "$T_1F_2T_1T_2F_1?_3$" to P]

P	S	M	L	ES	E
$T_1F_2F_1?_3$ $T_1T_2T_1T_2F_1?_3$ $T_1F_2T_1T_2F_1?_3$	F_1 $T_1T_2F_1$	---	---	3	---

[WS:

Copy S to WS and empty S

{ F_1
$T_1T_2F_1$ }

$F_1?_3$	does not reduce ES, add "F_1" to M
$T_1T_2F_1?_3$	does not reduce ES, add "$T_1T_2F_1$" to M

At this point, we have run out of subpaths to examine. Proceed to Step 7 for subpath completion]

P	S	M	L	ES	F
$T_1F_2F_1?_3$	---	F_1	---	3	---
$T_1T_2T_1T_2F_1?_3$		$T_1T_2F_1$			
$T_1F_2T_1T_2F_1?_3$					

[WS:

Copy P to WS
{ $T_1F_2F_1?_3$
$T_1T_2T_1T_2F_1?_3$
$T_1F_2T_1T_2F_1?_3$ }

$E_nT_1F_2F_1T_3E_x$	feasible path, add to F
$E_nT_1T_2T_1T_2F_1T_3E_x$	feasible path, add to F
$E_nT_1F_2T_1T_2F_1T_3E_x$	feasible path, add to F]

Now we have obtained three paths in F which are sufficient for testing the program.

3.2.3 Shortcomings of the Heuristic Method

The heuristic method has been demonstrated to be effective in finding the sufficient test path set on most programs examined. However, the method is not completely flawless; it has shortcomings which could lead to a failure in obtaining a sufficient path set. The heuristic method will keep generating subpaths to test a particular predicate until a set of sufficient subpaths has been obtained or until the system limit has been exceeded. When one of these conditions occurs, the method will terminate as if a sufficient set of subpaths has been obtained and proceed to test the next predicate. Although it gives us no explicit indications of failure when testing a particular predicate, we can examine the obtained subpaths and the corresponding error space for that predicate. As a matter of fact, any method for test path generation which limits the number of iterations of loops could fail to obtain the sufficient path set. This can be illustrated in the following example:

```
:
:
I = 0
WHILE (........) DO              T3
      :
      I = I + 1
      :
END WHILE
IF ( I .GT. K) THEN              T4
      IF (.........) THEN        T5
                  :
      END IF
END IF
:
:
```

If the number of iterations is limited to K, then any method for test path generation will fail to obtain feasible paths to test predicate T5 as long as the loop on T3 is iterated less than K+1 times.

Chapter 4

Experiments with SPTEST II

4.1 Input Programs and Experimental Results

With SPTEST, Sahay[WhP85] was able to perform experiments on a set of linearly domained programs in order to obtain greater-insight into path-oriented testing methods. In addition to the experiments on SPTEST, Sahay also spent a lot of time in choosing the optimum number of paths to test the programs. The set of linearly domained programs has been shown to be diversified and non-trivial. He has also computed Table 2 to illustrate the complexity of the programs.

Pgm.	Function	Lines	# of Input Variables	# of Program Variables	Dimension of Error Space	# of Predicates	Iteration Loops		McCabe Complexity Measure
							#	Nesting Level	
1	Euclid GCD	13	2	2	5	3	3	1	4
2	Integer Round-up	15	1	3	5	2	1		3
3		15	2	1	4	3	0		4
4	Integer Division Remainder	18	2	2	5	4	2	1	5
5	Euclid GCF	19	2	3	6	3	1		4
6	Conditional Series Summation	27	2	5	8	4	1		5
7	Sorted Set Intersection	31	8	5	14	6	2	1	6
8	Binary Search	42	11	5	17	7	1		9

Table 2: The Set of Programs Used for Experiments on SPTEST II.

In order to demonstrate the capability of the new system SPTEST II, experiments are conducted on SPTEST II with the same set of programs. The experiments are conducted at night on a VAX 780, and the computation time requires to perform the path generation process on a particular program is at most half-an-hour. The results obtained from SPTEST II are shown after the presentation of the results from SPTEST in the following tables. Both results are compiled to exclude the self-blindness vector from the error space.

Program	Predicate	Number of Subpaths Obtained	Final Dimension of Error Space	Error Classification	Total Number of Full Test Paths for Program
1	1	3	0		
	2	3	0		
	3	3	0		3
2	1	2	2	1 Unused Variable 1 Invariant Expression	
	2	2	2	2 Invariant Expression	2
3	1	1	1	1 Unused Variable	
	2	2	0		
	3	2	0		2
4	1	1	2	2 Unused Variables	
	2	1	2	2 Unused Variables	
	3	3	0		
	4	3	0		4

Table 3: Results of Experiments Using SPTEST.

Program	Predicate	Number of Subpaths Obtained	Final Dimension of Error Space	Error Classification	Total Number of Full Test Paths for Program
5	1	4	0		
	2	4	0		
	3	4	1	1 Equality Restriction	4
6	1	1	5	5 Unused Variables	
	2	2	4	3 Unused Variables 1 Invariant Expression	
	3	4	2	1 Unused Variable 1 Invariant Expression	
	4	4	2	1 Unused Variable 1 Invariant Expressions	4
7	1	4	0		
	2	5	0		
	3	3	0		
	4	3	1	1 Equality Restriction	
	5	3	1	1 Equality Restriction	5

Table 3: Results of Experiments Using SPTEST (Continued).

Program	Predicate	Number of Subpaths Obtained	Final Dimension of Error Space	Error Classification	Total Number of Full Test Paths for Program
8	1	1	5	5 Unused Variables	
	2	1	5	5 Unused Variables	
	3	4	1	1 Invariant Expression	
	4	3	2	1 Unused Variable 1 Invariant Expression	
	5	3	2	1 Unused Variable 1 Invariant Expression	
	6	3	2	1 Invariant Expression 1 Unused Variable	
	7	4	2	1 Unused Variable 1 Invariant Expression	4

Table 3: Results of Experiments Using SPTEST (Continued).

Program	Predicate	Number of Subpaths Obtained	Final Dimension of Error Space	Error Classification	Total Number of Full Test Paths for Program
1	1	3	0		
	2	3	0		
	3	3	0		3
2	1	2	2	1 Unused Variable 1 Invariant Expression	
	2	2	2	1 Unused Variable 1 Invariant Expression	2
3	1	1	1	1 Unused Variable	
	2	2	0		
	3	2	0		2
4	1	1	2	2 Unused Variables	
	2	1	2	2 Unused Variables	
	3	3	0		
	4	3	0		2

Table 4: Results of Experiments Using SPTEST II.

Program	Predicate	Number of Subpaths Obtained	Final Dimension of Error Space	Error Classification	Total Number of Full Test Paths for Program
5	1	3	0		
	2	3	0		
	3	3	2	1 Equality Restriction 1 Invariant Expression	3
6	1	1	5	5 Unused Variables	
	2	2	4	3 Unused Variables 1 Invariant Expression	
	3	4	2	1 Unused Variable 1 Invariant Expression	
	4	3	3	1 Unused Variable 2 Invariant Expressions	3
7	1	4	1	1 Invariant Expression	
	2	4	1	1 Invariant Expression	
	3	5	0		
	4	3	1	1 Invariant Expression	
	5	4	0		6

Table 4: Results of Experiments Using SPTEST II (Continued).

Program	Predicate	Number of Subpaths Obtained	Final Dimension of Error Space	Error Classification	Total Number of Full Test Paths for Program
8	1	1	5	5 Unused Variables	
	2	1	5	5 Unused Variables	
	3	5	1	1 Invariant Expression	
	4	3	2	1 Unused Variable 1 Invariant Expression	
	5	3	2	1 Unused Variable 1 Invariant Expression	
	6	4	1	1 Invariant Expression	
	7	3	3	3 Invariant Expressions	6

Table 4: Results of Experiments Using SPTEST II (Continued).

Program	Predicate	Number of Subpaths Obtained	
		SPTEST	SPTEST II
5	1	4	3
	2	4	3
	3	4	3
6	1	1	1
	2	2	2
	3	4	4
	4	4	3
7	1	4	4
	2	5	4
	3	3	5
	4	3	3
	5	3	4
8	1	1	1
	2	1	1
	3	4	5
	4	3	3
	5	3	3
	6	3	4
	7	4	3

Table 5: Comparison of the Number of Subpaths Obtained to Test the Predicates for Programs 5-8.

4.2 Analysis of the Results

For Programs 1, 2, 3, and 4, the sufficient subpaths obtained for each predicate are identical for both systems. Minor differences occur in the number of subpaths and the contents of the error space for Programs 5, 6, 7, and 8. As shown in Table 5, the maximum difference in the number of subpaths obtained for predicates is within the range of two, while the dimensions of the error space differed by no more than one.

In testing predicate 3 and predicate 5 for Program 7 and also predicate 3 and predicate 6 for Program 8, the increased number of subpaths obtained is a result of the trade-off between the loop extension and path feasibility. The heuristic method examines loop paths with one single iteration extended each time such that no subpaths would have multiple loop patterns appended in a single extension. It turns out that subpaths which employ multiple loop patterns in a single extension can reduce the error space more effectively. However, subpaths with multiple loop patterns are longer paths which have bigger constraint sets. Bigger constraint sets require more time to determine their feasibility. Also the larger the constraint set, the greater the chance of being infeasible. As we have mentioned earlier, the cost of feasibility determination is much higher than any other subprocesses in the system. This issue has been taken into account during the development of the heuristic method and we have decided to select shorter paths in order to reduce the processing time as well as the chance of generating infeasible paths. Incidentally, the perspective that longer paths can reveal errors more effectively than shorter paths has been confirmed in a recent research result using loop analysis for the domain testing strategy by White and Wiszniewski [WhW88] .

The differences in the size of the error spaces are mostly due to the occurrences of invariant expressions. Although we have a better understanding of how invariant expressions are formed, we cannot identify or remove them in general.

Program	Number of Test Paths Obtained	
	SPTEST	SPTEST II
1	3	3
2	2	2
3	2	2
4	4	2
5	4	3
6	4	3
7	5	6
8	4	6

Table 6: Comparison of the Number of Test Paths Obtained to Test Programs 1-8.

One interesting result from SPTEST II concerning the number of test paths should be mentioned. In Program 4, the maximum number of subpaths needed to test the predicate is three; therefore, we would expect that a minimum of three test paths are needed to test the program. Surprisingly, we could only obtain two different test paths from those three subpaths because one of the subpaths is identical to the other after the process of subpath completion. This occurs when a shorter path is forced to have additional iterations in order to produce a feasible path. In all testing strategies, it has been a well accepted fact that all programs should be tested with multiple paths. Ironically, the above observation gives us the insight that it may be possible to sufficiently test some programs with only a single path.

As we can see, the results obtained from SPTEST II are comparable to the results obtained from SPTEST. Although we cannot claim that the new system is superior to the old system, at least this study indicates that the heuristic method is very close to using human insight and intuition in selecting test paths. We should remind ourselves that the results obtained from SPTEST were conducted manually by Sahay, which required an extensive amount of time and intuition. Now SPTEST II can obtain a comparable result by just a few keystrokes on the keyboard. Without any doubt, the objective of this research has been achieved at least for this restricted set of programs.

Chapter 5

Side Issues Related to Sufficient Path Testing

5.1 Invariant Expressions and the Irreducible Error Space

We have defined an invariant expression as an algebraic expression which can be added to a program statement (assignment or predicate) without being detected along any possible path through that statement. An unused variable is an initialized program variable but is not used along any of the paths, so it also forms an invariant expression in the error space. In [7], Sahay has characterized that an irreducible error space contains vectors that correspond to unused variables, equality restrictions, the self-blindness vector and invariant expressions. The following example illustrates that all subpaths which stop at predicate T3 have unused variables I and J. These are also the invariants among the subpaths since variable I has a value of zero and variable J has a value of one.

Example 7: *Program 7.*

```
READ M, N
I = 0
J = 1
K = 0
IF ( M .LE. 5) THEN            T1
      K = N
ELSE
      K = M
END IF
IF ( K .GT. M-N ) THEN         T2
      PRINT K
END IF
IF  (..........)  THEN         T3
      :
      :
      :                        []
```

Although Zeil has anticipated the occurrence of invariant expressions in the error space, little has been done to identify or eliminate em. The occurrence of invariant expressions in an error space introduces a complication in the determination of an irreducible error space. In an attempt tc e greater insight into this problem, we have classified the invariant expression intc wo categories:

1. *Explicit Invariant Expression* (*EIE*)
2. *Implicit Invariant Expression* (*IIE*)

An EIE is an invariant which is caused by a single program statement, so an unused variable is an EIE. An IIE is an invariant which is caused by multiple program statements, and arises from:

1. the reconstruction of the original error space after some algebraic operations; or
2. the intersection of two or more error spaces.

To determine an irreducible error space, we have to identify the basis vectors in the error space. A trace on the program text is needed to identify unused variables, equality restrictions, self-blindness vectors and EIEs because they have a one-to-one correspondence with the program statements. In order to identify IIEs, some additional vector manipulations on the error space are required.

Since invariant expressions cannot be eliminated from the error space, they do not affect the choice of path selections. However, the existence of IIEs causes a major problem in the determination of irreducible error space. It is desirable to have an error space such that every basis vector in the error space expresses a program statement without any additional transformations. We will call such an error space a *recognizable error space* .

A suggested method to obtain a recognizable error space is to find out the corresponding program statement for each basis vector in the new error space after the calculation of the error space intersection.

Given subpaths P_a and P_b, such that each has an independent error space ES_a and ES_b, respectively , and $ES_a = \{v_1, v_2, ..., v_x\}$, $ES_b = \{w_1, w_2, ..., w_y\}$, the resulting error space after selecting these two paths is the intersection of ES_a and ES_b, say ES_{ab}, such that $ES_{ab} = \{u_1, u_2, ..., u_z\}$. To construct a recognizable error space R, initially empty, if we can express any basis vector (v_i or w_j) in ES_a and ES_b using a linear combination of $\{u_k\}$ in ES_{ab}, then that particular v_i or w_j is added to R. Thus, we have preserved the original vector components in R and we can determine whether R is reducible by conducting a trace on the text of the input program. Consider the following example:

Example 8: *Program 8*.

```
READ M, N
I = 0
J = 0
K = M + 2*N
IF ( M-N .GE. 1 ) THEN                 T1
       J = M
ELSE DO
       I = M
END IF
I = I + 1
IF ( I + J .GE. 3) THEN                T2
       K = K - 1
END IF
PRINT N, I, K
STOP
END                                    []
```

For the above example, two test paths are used to test predicate T2, and they produce two error spaces {v} and {w}, respectively.

1) $T_1?_2$

$v_1 = (I - 1)$ ==> I = I + 1

$v_2 = (J - M)$ ==> J = M

$v_3 = (K - M - 2N)$ ==> K = M + 2N

$v_4 = (I + J - 3)$ ==> IF (I + J .GE. 3) THEN

2) $F_1?_2$

$w_1 = (I - M - 1)$ ==> I = M + 1

$w_2 = (J)$ ==> J = 0

$w_3 = (K - M - 2N)$ ==> K = M + 2N

$w_4 = (I + J - 3)$ ==> IF (I + J .GE. 3) THEN

The final intersection U is

$u_1 = (M - 2)$ ==> M = 2

$u_2 = (I + J - 3)$ ==> IF (I + J .GE. 3) THEN

$u_3 = (K - 2N - 2)$ ==> K = 2N + 2.

Notice that only the basis vector u_2 occurs explicitly in the program and both u_1 and u_3 are IIEs. If we proceed to construct the recognizable error space R, we will have the following basis vectors in R:

$$r_1 = u_3 - u_1 = v_3 = w_3 = (K - M - 2N)$$
$$r_2 = u_2 = (I + J - 3).$$

The basis vectors v_1, v_2, w_1, and w_2 cannot be expressed in terms of the basis of the error space intersection U; there are only two unique explicit vectors expressible by U. Vector r_1 is the EIE ($K = M + 2N$) and vector r_2 is the self-blindness vector. A trace of the program text shows that all the subpaths which stop at predicate T2 have to go through the statement K = M + 2*N, and variable K does not receive a new value before reaching predicate T2. Thus, the value for variable K remains unchanged and [illegible]rms an invariant expression in the error space. In addition, knowing that [illegible]indness vector cannot be removed at any cost, it can be concluded [illegible]ducible error space for predicate T2 contains an invariant expression and self-blindness.

5.2 Sufficient Path Testing with Array Programs

The sufficient path testing strategy requires us to identify the size of the initial error space before the testing approach could apply. In order to generate different paths to test a program, we must allow a certain degree of flexibility in the values for the input variables. A hidden problem is revealed when the testing approach is applied to programs utilizing arrays, and the lengths of the arrays are specified by some other input variables. To determine the initial error space, we need to know the number of input variables. Each element in an array variable is considered to be a unique input variable. Since the length of an array is determined by one of the input variables, we have to know the actual value of that particular variable. Thus, that particular variable can only assume one value, the same value which defines the length of the array. In this case, we have lost all the value flexibility for that particular variable. This does not really create problems because it is justifiable to put a bound on the amount of memory that a program requires, as no computer can guarantee the successful execution of a program if it requires an arbitrary amount of memory.

The real problem arises, in terms of path generation, when a bound is put on the length of an array at the beginning of the program to determine the size of the initial error space, but due to the value flexibility, the bound is ignored afterwards. The path generation process will utilize the value flexibility of the input variables as if the bound does not exist at all. As a result, the inconsistency of these two processes can create a large number of infeasible paths. The following example will illustrate this idea:

Example 9: *Program 9.*

```
          READ M, N
          DO 10  I = 1, M                    T1
                 READ S(I)
   10     CONTINUE
          J = 1
          WHILE ( J .LE. N ) DO              T2
                 IF ( S(J) .GT. 0 ) THEN     T3
                        S(J) = S(J) + 1
                 END IF
                 J = J + 1
          END WHILE
          WHILE ( J .LE. M ) DO              T4
                 PRINT S(J)
                 J = J + 1
          END WHILE
          :
          :                         []
```

In the above example, M, N, S(1),..., S(M) are input variables. We need to know the value of M such that the number of input variables is M + 2. Suppose we know that M is 3, N is less than M, and the number of loop iterations is limited to two. A subpath which iterates the loop at T2 twice would give variable J a value of 3. The same subpath which iterates the loop at T4 twice would give variable J a value of 4, and reference is made to S(4) which does not exist. Thus, any subpath extended along this pattern will be infeasible. One of the alternatives to handle this problem is to modify the input program to replace all the occurrences of variable M with a constant value of 3 as follows:

```
           READ N
           DO 10  I = 1, 3                         T1
                 READ, S(I)
    10     CONTINUE
           J = 1
           WHILE ( J .LE. N ) DO                   T2
                 IF ( S(J) .GT. 0 ) THEN           T3
                       S(J) = S(J) + 1
                 END IF
                 J = J + 1
           END WHILE
           WHILE ( J .LE. 3 ) DO                   T4
                 PRINT S(J)
                 J = J + 1
           END WHILE
           :
```

Since SPTEST II is unable to handle this problem, it is suggested that such array programs should be modified before SPTEST II is applied.

Chapter 6

Conclusions

6.1 Summary

As the demand for reliable software increases, the role of software testing becomes increasingly important. Testing strategies can be classified into two categories, white box testing and black box testing. We are interested in one of the subclasses in the white box testing approach called path analysis testing.

A computer system called SPTEST, which implements a vector space model proposed by Zeil, is available as an aid to select paths to test program predicates. This research focuses on an extension of SPTEST such that the path selection process is done automatically and systematically. To automate the process, several related issues of path selection, including the determination of path feasibility, iteration of loops, collection of subpath fragments and the determination of an irreducible error space have to be examined and resolved. As a result, an extended system called SPTEST II is implemented on UNIX using FORTRAN 77. SPTEST II adopts the concepts of subpath extension to produce a set of test paths instead of subpath fragments, and can be used to conduct the actual testing of the program using any reliable testing strategy. This research has made the following contributions to the field of program testing:

1. the development of a heuristic method which generates test paths systematically;

2. an automated computer system, which implements the above heuristic, as an extension of SPTEST for generating a sufficient set of test paths in order to test a computer program;

3. a set of experiments to show that the new system can generate results comparable to those obtained by SPTEST which is utilized manually by someone with considerable experience and intuition;

4. a proposed method to obtain the recognizable error space which helps to identify an irreducible error space.

In summary, the attempt to extend SPTEST to an automatic system which produces test paths, instead of subpath fragments, has been successful. The [illegible]ousness of running SPTEST to obtain sufficient paths, and the huge amount of time spent in the operation and human intuition required to select paths are all too demanding to be practical. With the new system SPTEST II, a set of sufficient test paths will be available to the user by using a few simple commands. Now a simple driver routine can easily be written to bring the two processes, namely test path generation and test data generation, together to perform testing on programs.

6.2 Future Work

Although satisfactory preliminary experimental results suggest that the performance of the heuristic method is comparable to using human intuition to generate test paths for the selected programs, further refinement of the heuristic method is needed to relax the restrictions, to accommodate more complicated programs and to provide system integration to speed up the process.

To enhance the performance of the system, a better linear programming solver package should replace the current package such that it could accept all possible values for the unknowns in the constraint sets. As a result, a bigger collection of feasible paths will be available for selection, and a sufficient set of test paths can be obtained in a short period of time.

At present SPTEST II can only be used to detect predicate errors on linearly domained programs. It would be desirable to expand the system such that it could detect assignment errors and computation errors. In [Ma88], a strategy to unify the testing of assignment errors and predicate errors is proposed. The strategy is called *blindness based testing* and is implemented into an interactive system called *bbtest*.. It would be interesting to see how well the heuristic method could utilize bbtest to detect predicate errors and assignment errors. In addition, more research need to be done on testing non-linear predicates in programs. What kind of practical approach could be used to determine the feasibility of paths in non-linear programs?

It has been observed in our experiments that most irreducible error spaces for predicates contain invariant expressions; it would be interesting to look at invariant expressions to see if they could be used to reflect program correctness. To what extent are invariant expressions related to program specifications?

The analysis of the experimental results has shown that it is not unusual to have a set of subpaths to test a predicate which extended to form a smaller set of paths due to the problem of path feasibility. An interesting branch that we could pursue is the possibility of testing a given program using a single path. What class of programs can this be applied to and how practical are that class of programs in general?

The problem in testing programs which utilize arrays has been identified in this research. This class of programs and their introduced complications need to be examined further. More practical ways must be developed to test this class of programs so that the sufficient path testing strategy can be applied more effectively.

As for the applicability of paths in testing programs, it would be interesting to know which paths give the greatest information for testing a particular program. This approach can be used as a rating mechanism to sort a finite set of sufficient paths according to their importance for testing. A sequential re-examination of the sorted list using the path rejection criterion would allow us to converge to the optimum number of paths to test a program.

REFERENCES

[Bei83] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold Company, 1983.

[Bor87] A. Borning, "Computer System Reliability and Nuclear War", *Communications of ACM*, Vol. 30, No. 2, February 1987, pp. 112-131.

[Dav73] M. Davis. "Hilbert's Tenth Problem is Unsolvable", *American Math. Mon.*, 80, 1973, pp. 233-269.

[GoG75] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 156-173.

[How76] W. Howden, "Reliability of Path Analysis Testing Strategy", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, Sept. 1976, pp. 208-215.

[Ma88] S. Ma, "Blindness-based Testing for Domain Errors", Master Thesis, University of Alberta, June 1988.

[Mye79] G. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.

[Whi86] L. White, "Software Testing and Verification", *Advances in Computers*, Vol. 26, Academic Press, Inc., 1987, pp. 335-391.

[WCC78] L. White, E. Cohen, B. Chandrasekaran, *A Domain Strategy for Computer Program Testing*, Technical Report Series, OSU-CISRC-TR-78-4, The Ohio State University, Columbus, Ohio, 1978.

[WCL87] L. White, W. Chiu, T. Lai, "Experiments to Select Test Points for Domain Testing", *CIPS Congress 87*, May, 1987, pp. 265-271.

[WhP85] L. White, P. Sahay, "Experiments determining best paths for testing computer program predicates", *Proc. IEEE Int. Conf. on Software Engineering, 8th,* 1985, pp. 238-243.

[WhW88] L. White, B. Wiszniewski, "Complexity of Testing Iterated Borders for Structured Programs", *Second Workshop on Software Testing, Verification, and Analysis*, July 19-21, 1988, to appear.

[ZeW81] S. Zeil, L. White, "Sufficient Test Sets for Path Analysis Testing Strategies", *Proc. IEEE Int. Conf. on Software Engineering, 5th,* 1981, pp. 184-191.

Appendix 1

Input Programs

The input programs used in this research project are listed in this appendix.

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
```

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. 0.50) THEN DO
        I = I + 1
END IF
PRINT, N, I
STOP
END
```

Program 3 :

```
READ, X, Y
A = 0
IF (Y .GT. X) THEN DO
        A = X
ELSE DO
        A = Y
END IF
IF (A .EQ. 0) THEN DO
        PRINT, A
END IF
IF (Y .GT. 0) THEN DO
        PRINT, Y
END IF
STOP
END
```

Program 4 : 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. Y) DO
                                R = R - A
                                A = A + A
                        END WHILE
                END WHILE
        END IF
END IF
PRINT, R, X, Y
STOP
END
```

Program 5 : 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
        PRINT, A, B
ELSE DO
        PRINT, A, B, S
END IF
STOP
END
```

Program 6 : Conditional Series Summation

```
READ, A, B
C = 0
D = 0
E = 0
F = 0
I = 0
IF (A .GT. B) THEN DO
        C = B + 1
ELSE DO
        C = B - 1
END IF
D = 2 * A + B
IF (C .GT. 0) THEN DO
        I = 1
        WHILE (I .LE. B) DO
                E = E + 2 * I
                I = I + 1
        END WHILE
END IF
IF (D .LE. 2) THEN DO
        F = E + A
ELSE DO
        F = E - A
END IF
PRINT, F
STOP
END
```

Program 7 : Sorted Set Intersection

```
      READ, M, N
      DO 10 I = 1, M
            READ, S1(I)
10    CONTINUE
      DO 20 I = 1, N
            READ, S2(I)
20    CONTINUE
      DONE = 0
      I = 1
      J = 0
      X = 0
      Y = 0
      WHILE (I .LE. M) DO
            IF (N .GT. 0) THEN DO
                  J = 1
                  DONE = 0
                  WHILE (DONE .EQ. 0) DO
                        X = S1(I)
                        Y = S2(J)
                        IF (X .EQ. Y) THEN DO
                              PRINT, S1(I)
                              DONE = 1
                        END IF
                        J = J + 1
                        IF (J .EQ. N) THEN DO
                              DONE = 1
                        END IF
                  END WHILE
            ELSE DO
                  I = M + 1
            END IF
            I = I + 1
      END WHILE
      STOP
      END
```

Program 8 : Binary Search

```
        READ, N
        DO 10 I = 1, N
              READ, B(I)
 10     CONTINUE
        READ, A
        I = 0
        HIGH = 0
        LOW = 0
        MID = 0
        TEMP = 0
        IF (N .GT. 0) THEN DO
              HIGH = N + 1
              LOW = 1
              MID = (HIGH + LOW) / 2
              I = 0
              TEMP = B(MID)
              IF (A .EQ. TEMP) THEN DO
                    I = 1
              END IF
              WHILE ( I .NE. 1) DO
                    TEMP = B(MID)
                    IF (A .EQ. TEMP) THEN DO
                          I = 1
                    ELSE DO
                          IF (A .LT. TEMP) THEN DO
                                HIGH = MID
                          ELSE DO
                                LOW = MID
                          END IF
                          MID = (HIGH + LOW) / 2
                    END IF
                    IF (LOW .EQ. MID) THEN DO
                          I = 1
                    END IF
              END WHILE
              TEMP = B(MID)
              IF (A .EQ. TEMP) THEN DO
                    PRINT, A
              END IF
        END IF
        STOP
        END
```

Appendix 2

Sufficient Paths Generated For Programs In Appendix 1

Program	Paths Generated
1	$T_1F_2T_3F_3F_1$
	$T_1T_2F_2F_3F_1$
	$T_1F_2T_3T_1T_2F_2T_3T_3F_3F_1$
2	F_1T_2
	$T_1F_1T_2$
3	$F_1F_2T_3$
	$T_1F_2T_3$
4	$T_1T_2T_3T_4F_4T_3F_4F_4F_3$
	$T_1T_2T_3T_4F_4F_3$
5	$T_1T_2F_1T_3$
	$T_1T_2T_1T_2F_1T_3$
	$T_1F_2T_1T_2F_1T_3$
6	$T_1T_2F_3T_4$
	$F_1T_2T_3T_3F_3T_4$
	$F_1T_2T_3F_3T_4$
7	$T_1T_2T_3F_4T_5F_3F_1$
	$T_1T_2T_3T_4T_5F_3F_1$
	$T_1T_2T_3F_4F_5T_3F_4T_5F_3F_1$
	$T_1T_2T_3T_4T_5F_3T_1T_2T_3F_4T_5F_3F_1$
	$T_1T_2T_3F_4T_5F_3T_1T_2T_3F_4T_5F_3F_1$
	$T_1T_2T_3T_4T_5F_3T_1T_2T_3T_4T_5F_3T_1T_2T_3F_4T_5F_3F_1$

Program	Paths Generated
8	$T_1T_2F_3T_7$
	$T_1F_2T_3F_4T_5F_6T_3T_4F_6F_3T_7$
	$T_1F_2T_3F_4F_5F_6T_3T_4F_6F_3T_7$
	$T_1F_2T_3F_4T_5F_6T_3F_4F_5F_6T_3T_4F_6F_3T_7$
	$T_1F_2T_3F_4F_5F_6T_3F_4F_5F_6T_3T_4F_6F_3T_7$
	$T_1F_2T_3F_4F_5F_6T_3F_4T_5F_6T_3T_4F_6F_3T_7$