INFORMATION TO USERS

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

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

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

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

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

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR
COHEN, EDWARD IRA
A FINITE DOMAIN-TESTING STRATEGY FOR COMPUTER PROGRAM TESTING.
THE OHIO STATE UNIVERSITY, PH.D., 1976

© Copyright by
Edward Ira Cohen
1978
A FINITE DOMAIN-TESTING STRATEGY

FOR COMPUTER PROGRAM TESTING

DISSERTATION

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

By

Edward Ira Cohen, B.S., M.S.

* * * * *

The Ohio State University
1978

Reading Committee:

Lee J. White
H. William Butteimann
B. Chandrasekaran
Stuart H. Zweven

Approved By

Lee J. White
Adviser
Department of Computer
and Information Science
To my Mother and Father
I wish to express my sincere gratitude to Professor Lee J. White, Professor B. Chandrasekaran, Professor H. William Buttelmann, and Professor Stuart H. Zweben for their time and efforts as members of my dissertation reading committee. Special gratitude is due Professor White for his untiring assistance, guidance, and encouragement throughout the course of developing my dissertation research. I wish to thank the Department of Computer and Information Science for the financial support provided, and I wish to express my gratitude to the CIS faculty, staff, and graduate students for providing a congenial and stimulating atmosphere in which to work. Thanks is due to the Instruction and Research Computer Center for the support afforded by their facilities, and in particular I wish to thank the staff of the CIS DECsystem-10 for their assistance and cooperation. I also wish to gratefully acknowledge the Air Force Office of Scientific Research for the financial support provided through grant 77-3416 during the latter stages of this research. Finally my deepest appreciation is reserved for my parents for their devotion, sacrifices, and continual encouragement throughout the long years of my education.
VITA

August 19, 1950. . . . Born - Boston, Massachusetts

1972. . . . . . . . . .  B.S., Rensselaer Polytechnic Institute, Troy, New York

1972-1977 . . . . . .  Graduate Teaching Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1973. . . . . . . . .  M.S., The Ohio State University, Columbus, Ohio

1977-1978 . . . . . .  Graduate Research Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Programming Languages and Translators. Professor Lee J. White

Studies in Computer and Systems Programming. Professor Clinton R. Foulk

Studies in Computer Architecture. Professor Ming T. Liu
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ......................................................... i
VITA ..................................................................................... iv
LIST OF TABLES .................................................................. vii
LIST OF FIGURES. ............................................................... ix

Chapter

1. INTRODUCTION ............................................................. 1
   1.1 The Importance of the Research Area ....................... 1
   1.2 Statement of the Problem ......................................... 2
   1.3 The Approach to the Problem and Expected Results .... 4
   1.4 Overview of the Dissertation .................................... 6

2. SURVEY OF RELATED RESEARCH. ............................. 8
   2.1 Software Design and Development ............................ 8
   2.2 Computer Program Verification ................................ 13
   2.3 Computer Program Testing ....................................... 17

3. BASIC CONCEPTS AND DEFINITIONS. ........................ 24
   3.1 A Sample Programming Language ............................ 24
   3.2 The Standard Program Form .................................. 27
   3.3 Program Paths ....................................................... 29
   3.4 Symbolic Path Execution ....................................... 32
   3.5 Path Predicates ..................................................... 34
   3.6 A Model of Program Computation ............................ 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The Input Space</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Input Space Domains</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Predicate Types and Domain Characteristics</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>Predicates Which Do Not Produce Border Segments</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>Infeasible Paths</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>An Example Program and Input Space Partitioning Diagram</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of the Error Classification Scheme</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Domain Errors</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Transformation Errors and Coincidental Correctness</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Missing Path Errors</td>
<td>53</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>The Two-Dimensional Linear Case</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>The General N-Dimensional Linear Case</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>The General Nonlinear Case</td>
<td>77</td>
</tr>
<tr>
<td>6.5</td>
<td>Testing Equality and Nonequality Predicates</td>
<td>81</td>
</tr>
<tr>
<td>6.6</td>
<td>Effectiveness for Transformation and Missing Path Errors</td>
<td>87</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Paths and Predicate Interpretations</td>
<td>57</td>
</tr>
<tr>
<td>2. Predicate Statistics for COBOL Programs</td>
<td>182</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A Binary Search Program</td>
<td>26</td>
</tr>
<tr>
<td>2. Digraph Representation of Binary Search Program</td>
<td>31</td>
</tr>
<tr>
<td>3. Symbolic Execution of A Program Path</td>
<td>33</td>
</tr>
<tr>
<td>4. Models of Program Computation</td>
<td>39</td>
</tr>
<tr>
<td>5. Predicate Types and Regions</td>
<td>44</td>
</tr>
<tr>
<td>6. Exterior and Interior Borders of A Domain</td>
<td>46</td>
</tr>
<tr>
<td>7. Simple and Complex Predicate Dominance</td>
<td>51</td>
</tr>
<tr>
<td>8. Predicate C3 Dominated by (C1 AND C2)</td>
<td>54</td>
</tr>
<tr>
<td>9. Infeasibility Caused by [(NOT C(3))]</td>
<td>54</td>
</tr>
<tr>
<td>10. Sample Program With Five Predicates</td>
<td>56</td>
</tr>
<tr>
<td>11. Input Space Structure - Snapshot 1</td>
<td>58</td>
</tr>
<tr>
<td>12. Input Space Structure - Snapshot 2</td>
<td>59</td>
</tr>
<tr>
<td>13. Input Space Structure - Snapshot 3</td>
<td>61</td>
</tr>
<tr>
<td>15. Fully Developed Input Space Structure</td>
<td>63</td>
</tr>
<tr>
<td>16. The Three Types of Border Shifts</td>
<td>71</td>
</tr>
<tr>
<td>17. Domain Errors For An Equality Predicate</td>
<td>73</td>
</tr>
<tr>
<td>18. Domain Errors For An Inequality Predicate</td>
<td>75</td>
</tr>
<tr>
<td>19. Subdomains for Simple Missing Predicates</td>
<td>83</td>
</tr>
<tr>
<td>20. Subdomains for Compound Missing Predicates</td>
<td>85</td>
</tr>
<tr>
<td>21. Test Points for A Two-Dimensional Linear Border</td>
<td>92</td>
</tr>
<tr>
<td>22. Improperly Selected OFF Point</td>
<td>94</td>
</tr>
<tr>
<td>23. Domain Testing for Border Shifts</td>
<td>96</td>
</tr>
<tr>
<td>24. Domain Test Points for Closed and Open Borders</td>
<td>98</td>
</tr>
<tr>
<td>25. Domain Test Points for A Three-Dimensional Border</td>
<td>102</td>
</tr>
<tr>
<td>26. Necessity of Linearly Independent ON Test Points</td>
<td>103</td>
</tr>
<tr>
<td>27. Necessity of Properly Chosen OFF Test Point</td>
<td>103</td>
</tr>
<tr>
<td>28. Complexity of A General Nonlinear Border</td>
<td>107</td>
</tr>
<tr>
<td>29. Test Points for A General Nonlinear Border</td>
<td>113</td>
</tr>
<tr>
<td>30. Testing A Two-Dimensional Nonlinear Border</td>
<td>116</td>
</tr>
<tr>
<td>31. Borders Tangential At ON Test Points</td>
<td>116</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>32.</td>
<td>Test Points for A Compound Inequality.</td>
</tr>
<tr>
<td>33.</td>
<td>Equality as A Compound Inequality in the Limit.</td>
</tr>
<tr>
<td>34.</td>
<td>Domain Testing for An Equality Predicate</td>
</tr>
<tr>
<td>35.</td>
<td>Domains Defined With Compound OR Predicates</td>
</tr>
<tr>
<td>36.</td>
<td>Domain Test Points for the Discrete Case</td>
</tr>
<tr>
<td>37.</td>
<td>The Identification of Adjacent Domains</td>
</tr>
<tr>
<td>38.</td>
<td>Possible Adjacent Domain Structures</td>
</tr>
<tr>
<td>39.</td>
<td>Data Flow Model of A Short Example Program</td>
</tr>
<tr>
<td>40.</td>
<td>Composite Borders and Independent Borders</td>
</tr>
<tr>
<td>41.</td>
<td>Shift of A Single Independent Border</td>
</tr>
<tr>
<td>42.</td>
<td>An Optimal Set of Test Points</td>
</tr>
<tr>
<td>43.</td>
<td>Points Defined by the Domain Testing Strategy</td>
</tr>
<tr>
<td>44.</td>
<td>Difficulty of Generating Optimal Test Points</td>
</tr>
<tr>
<td>45.</td>
<td>Sample Input Space and Domain Test Points</td>
</tr>
<tr>
<td>46.</td>
<td>Correct Sample Input Space for Domain Errors</td>
</tr>
<tr>
<td>47.</td>
<td>Correct Sample Input Space for Missing Path Error</td>
</tr>
<tr>
<td>48.</td>
<td>Sample Errors for An Equality Predicate</td>
</tr>
<tr>
<td>49.</td>
<td>The Three Stages of Domain Test Data Generation</td>
</tr>
<tr>
<td>50.</td>
<td>Schematic Diagram of Stage I Processing</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1 The Importance of the Research Area

The importance of the software component of current computer systems has become obvious in recent years, and the many failures of these large and complex systems are well-documented. Software design, development, and validation are engineering tasks of major complexity, and the frequent delays encountered in the development of complete systems can usually be traced to software-related problems. Since the development of software is a labor-intensive task, its cost in a large system development project is enormous and is expected to continue to rise precipitously. Most importantly, software is the weak link in system reliability, and unless we find better ways to decrease the number of undiscovered errors, it will become a major impediment to the future utility of all computer systems. Software systems are becoming larger, more complex, and more costly, and significant progress must be made in developing a sound software engineering discipline, or these systems will soon outgrow our capabilities to control them.

Since software is complex and manually produced, it is very error prone, and some form of validation will always be necessary. Computer program testing is the traditional approach to this problem of "software quality assurance," but the scientific study of this technique is in its infancy, as is the entire field of "software engineering." Program testing has always been a very personal task, based on the intuition and motivation of the individual programmer involved. Undoubtedly some people do test their programs well while others do little or no testing at all, but in every case the testing is based on personal style rather than scientific principles. The overall goal of our research is to replace the intuitive and ad hoc principles by which current program testing procedures are designed with a methodology based on a formal treatment of the program testing problem. In addition, since a complete
solution to the problem is impossible, we must evaluate the performance of any proposed technique and identify its limitations.

Software engineering encompasses many diverse aspects of software research. Its general goal is the production of software which is understandable, reliable, maintainable, and efficient, without being exorbitantly expensive. Much work has been done on new design techniques and standards, both for individual programs and for complete systems. These new techniques also emphasize modifiability, which implies that the software will be easy to maintain. Research in software quality assurance has concentrated on two approaches, program verification and program testing, while program verification would provide a formal and fairly complete guarantee of program correctness, the practical difficulties inherent in this mathematical technique make it unlikely that it can be utilized on a large scale in the near future. We believe that a program testing methodology, which is based on a formal treatment of the problem, can be used to achieve the desired level of software reliability and can easily be applied in practice. So, in summary, computer program testing is a promising approach to the problem of software reliability, which itself is of the utmost importance, since, if software is unreliable, all of its other properties are meaningless.

1.2 Statement of the Problem

To test a program we execute it with a finite set of sample input values. A test case is defined as a single set of sample data and contains one value for each of the N input variables in the program. The set of test data is then defined as the complete set of test cases used to test a program, and if there are T test cases, the test data will contain a total of T*N individual values. The goal of the testing process is limited to the detection of an error. Any attempt to identify the error, its cause, or an appropriate correction is properly categorized as debugging and is beyond the specific scope of the testing process. So, testing is essentially error detection, while debugging is the more difficult process of error correction. Of course, in practice these two distinct activities overlap and are frequently combined into a single testing/debugging phase in the software development cycle.
The output produced for each test case must be evaluated to determine if the program is correct for that particular test case, and in this way the input-output relationship is verified for this particular case. If the output is incorrect, the program obviously contains an error, and the testing process has successfully indicated its existence. If the test case is processed correctly, we not only can conclude that the program is correct for this test case, but we can also infer that the program would be correct for other related input data. Therefore the basis for the success of the testing process is that the correctness of the program can be inferred from the observed correctness of a finite set of test cases selected according to some specific criterion. Of course, the amount of confidence we gain from testing a program depends on how intelligently we select the specific test data for the program. The crux of the program testing problem is to formulate a test data selection criterion which will always define a finite set of test cases, which is of reasonable size and provides a high level of confidence.

The natural unit of testing is the program path since the processing done by a program consists of the computations performed along its many paths. In this type of path-oriented testing strategy, program paths are tested individually in some sequence, and we can identify two distinct phases in the process, path selection and test data generation. A program typically contains a great number, in fact a potentially infinite number, of control flow paths. In the first phase a "path selection strategy" is used to select a finite sequence of paths, which are in some sense the most effective paths for testing. In the second phase specific test cases are generated for each of the selected paths in sequence. In this research we will investigate the second problem, that of an effective test data generation criterion for a single path. The goal of our research is a general characterization of a small finite set of test cases which, if processed correctly, would imply that no error exists for the path being tested. Our results for this second phase are independent of the first phase in that our test data selection strategy makes no assumption about how a specific test path might have been selected.

In this research we need to make some basic assumptions which can be easily satisfied in practice. Program testing is a very powerful process, and therefore we will concentrate on those types of errors which are the most difficult to test for. We therefore assume that any program being tested has been compiled successfully. Since input/output processing is so closely linked to a particular machine/compiler environment, we assume that all I/O errors
have previously been eliminated. Since in program testing we must evaluate the correctness of the input-output relationship for specific data points, we also assume that infinite loop errors and all errors causing premature termination have been discovered and corrected prior to testing. So we are really concentrating our efforts on errors in which the program seems to operate as it should, i.e., terminates normally and produces the type of output expected, but computes incorrect output values for some input values. These "logic errors" can be viewed as errors occurring in some phase of the process by which a program is transformed from a vague notion to actual code, e.g., writing program specifications, designing an algorithm to meet the specifications, or coding a program to implement the algorithm. Also, these errors have traditionally been recognized as the most difficult type of error to find and correct.

In order for a program testing methodology to be accepted by practitioners it must be easy to use and should require only a minimal amount of extra work by the user. Therefore we assume that no information other than the program itself is provided by the programmer to guide the test data selection strategy. However, a test case is useful only if it can be determined whether the program has executed it correctly. In our research we assume that an "oracle" is available which can decide unequivocally if the output is correct for any specific test case, but in practice this decision must be made by the user. Our methodology generates a set of test points for the user, and it is only reasonable to require that the user interpret the testing results.

1.3 The Approach to the Problem and Expected Results

Our approach is based on the predictable geometric characteristics of the set of input data which follows a particular path. This predictability allows certain general conclusions about the correctness of the path based only on the results of a small set of test cases, selected according to some specific criterion. A program which reads N input variables and produces M output variables can be viewed as a complex function mapping points in an N-dimensional input space to points in an M-dimensional output space. The control flow in the program partitions the input space into a set of mutually exclusive and exhaustive domains. Each of these domains corresponds to a particular path in the
program and consists of the set of input data points in the input space which will cause the corresponding path to be executed. So in testing a particular path we are actually testing the computations performed by the program over a specific input space domain.

The partitioning structure of the input space reflects the possible control flows in the program and is central to our testing strategy. Every input space domain has certain geometric characteristics, and in particular the form of the domain boundary is very predictable. Our testing strategy combines this predictability and the observed results of a small number of test cases to infer the correctness of the program for the particular domain being tested. The necessary test cases are selected based on their geometrical relationship to the entire domain and its boundary. Since the structure of a particular domain is completely determined by the control flow conditions for the corresponding path, all the information required by the domain testing strategy is contained in the sequence of program statements constituting the path. In summary, in this research we look beyond the program itself to analyze the input space domain strictly as a geometrical entity. In this way we are able to transform the problem from its usual form as an informal study of programs and programming to a more formal investigation into the geometry of input space domains.

Research on the problem of software reliability is just beginning, and to our knowledge the domain testing strategy is the first methodology for selecting test data which has been shown to be effective in detecting a major class of errors. It provides a test data selection criterion which is based on a mathematical analysis of program computation and the testing process. In addition, the criterion can lead to practical and efficient implementations. Those few subclasses of errors for which the strategy is not guaranteed to be completely effective are precisely defined and are shown to be basic theoretical limitations to the reliability of any conceivable testing strategy.

The results and analytical techniques developed in this research represent a significant step toward a practical and logically sound computer program testing methodology. This work provides both an effective strategy which can be implemented directly and a solid foundation for future investigations.
1.4 Overview of the Dissertation

Since program testing is but one problem in the general area of software engineering, in the next chapter we provide a brief review of recent work in software design and development. We also survey various results obtained in studies of program verification. Of course a much more detailed survey of current research in program testing is provided and constitutes the bulk of chapter two.

In chapters three through five we develop the foundation on which to base the domain testing strategy. Chapter three defines and discusses some preliminary concepts which are central to this research, and in particular two important notions, program paths and path predicates, are stressed. In addition, a model of program computation is presented which introduces the idea of a partitioned input space. The partitioning of the input space is of the utmost importance in our work, and the full details of this concept are carefully developed in chapter four. The properties of the input space as a whole and of individual domains and their borders are emphasized, since these characteristics most directly affect the domain testing strategy. An error classification scheme is necessary in order to analyze and evaluate a program testing strategy, and in chapter five the scheme adopted for use in our program testing research is presented. The classification is based on the observable effects of an error rather than on the specific source of the error, and each class is described in depth. In addition certain theoretical problems associated with testing for each type of error are identified and discussed.

The main body of research results is presented in chapters six and seven. Chapter six contains a complete description and validation of the basic domain testing strategy. Domain testing is first developed for the simplest case of linear inequalities in a two-dimensional input space. It is then generalized to higher dimensional spaces, the general nonlinear case, and both equality and nonequality predicates. Finally, the effectiveness of domain testing for transformation errors and missing path errors is evaluated. In chapter seven we address many of the assumptions made in presenting the results of chapter six. The strategy is extended to include domains defined with compound predicates and domains in a discrete input space. We also demonstrate how domain testing can be modified to satisfy the requirement that adjacent domains should not compute the same function. An analysis of coincidental correctness is presented which shows that its
practical implications may be insignificant in most cases. In chapter seven we also consider an optimal strategy, which defines the minimal set of points needed for domain testing. Finally, the error detection capability of our methodology is demonstrated using a sample program containing various types of programming errors.

In order to demonstrate the practicality of the domain testing strategy we outline a method of implementation in chapter eight. The implementation is based on standard linear programming techniques, and we focus on the changes necessary in these well-developed algorithms in order to apply them to our particular problem. Finally, in chapter nine our results are summarized, and the significant contribution of our research toward the goal of greater software reliability is discussed. In addition, open questions in this area suggested by our research are identified, and possible directions for future investigation are proposed.
CHAPTER 2

SURVEY OF RELATED RESEARCH

Computer program testing is one approach to the problem of "software quality assurance", which itself is a part of the larger field of "software engineering". To put our work into perspective a general overview of research in software design and development is presented in section 1. In section 2 research in program verification, which is a theoretical approach to the problem of software quality assurance, is surveyed. Section 3, which is most germane to our work, provides a detailed discussion of research in computer program testing.

2.1 Software Design and Development

Based on data from existing computer systems Boehm [BOEHB73] has identified software as the major source of cost overruns, schedule delays, and reliability problems. Unfortunately, if current trends continue, we can expect these software problems to worsen in the future. Traditionally the system hardware component was of prime importance, but currently and in any future large scale system development we can expect the cost of the software to be between three and seven times the cost of designing and developing the hardware. Boehm further estimates that fully one per cent of the Gross National Product is spent on all phases of software design, development, and maintenance. This huge expenditure is divided into approximately 35 per cent for analysis and design, 15 per cent for coding, and fully 50 per cent of every dollar spent on software is expended during the testing, debugging, and validation phase. From this Boehm concludes that research in software reliability holds the most promise for improving software technology.
Recent papers by Boehm [BOEHB76] and Morgan [MORGD77] provide excellent surveys of software engineering research, both work in progress and promising directions for future investigation. Boehm provides a general definition of software as both computer programs and all the documentation needed for their development, operation, and maintenance. Software Engineering is then defined as

"the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them." (*)

Unfortunately, scientific knowledge available on which to base such an important engineering discipline is very small.

Many software monitoring systems have been developed to provide both static and dynamic program profiles. These systems provide quantitative and meaningful data for compiler writers, programmers attempting to optimize program performance, and software engineers. Knuth [KNUTD71] studied a random sample of FORTRAN programs written both by students and by professional programmers. The basic conclusion of this study was that programs consist mainly of very simple expressions and constructs. For example, he found that 80 per cent of all assignment statements were of the forms \( V1 = V2 \), \( V1 = V2 + V3 \), or \( V1 = V2 - V3 \). Also, 70 per cent of all DO loops in the programs profiled contained three or less statements. In [ELSHJ75, ELSHJ76] Elsnoff studies 120 production PL/I programs and shows similar results, such as the fact that 80 per cent of all DO constructs are used only to group statements, 97 per cent of all arithmetic operators are + or -, and 98 per cent of all expressions contain fewer than two operators. Gordon and Salvadori [GORDJ75, SALVA75] have also developed a similar monitoring system for COBOL programs. All of these studies conclude that typical programs consist of very simple statements and constructs. However, the high complexity of the programs is caused by the complicated control flow and data flow. These results are surprising since they seem to be independent of both the programming language used and the level of programmer experience.

In [STUCL72, SUCL73] Stucki describes the Program Evaluator and Tester (PET) system. PET automatically inserts a set of software monitors into a FORTRAN program being tested, and therefore the program seems to monitor or measure its own execution. This self-metric capability

(*) BOEHB76, p. 1220.
provides statistics such as the percentages of statements executed, branches taken, and subroutine calls made during the test run, and this information can be used to evaluate how thoroughly the test data have exercised the program. It also produces range data for each variable assignment made and for DO loop indices.

Fosdick [FOSDL76] describes DAVE, a system for validating, debugging, and documenting FORTRAN programs. **Data flow analysis** is the study of the relationships between the statements which assign values to variables and the statements in which these variables are subsequently used. DAVE uses these "def-use relationships" to identify various data flow anomalies which signal likely programming errors, e.g., a path on which a variable is uninitialized or is assigned a value but never used. The data flow information produced is also useful in program documentation. These software tools provide needed information about how people really write programs. We expect systems of this type to become standard software development tools.

Computer programming is a human process, and therefore aspects of human psychology should be useful in the study of the programming process. Weinberg [WEING71] provides an excellent overview of the human factors involved in programming. In particular he discusses the problems of designing good experiments to study programmers and how a programmer’s personality traits correlate with the quality of the work he produces. In [HALSM72, HALSM73] Halstead describes a theory of "software physics" which views algorithms as distillations of human thought processes. He advances the hypothesis that algorithms possess a general structure obeying physical laws. A set of these laws, based on counts of operator and operand occurrences, is developed and validated experimentally.

Perhaps the most active area has been the development of new software design techniques, both for complete systems and for individual programs. Two major concepts which are common to most recently developed strategies are modular design and top-down structured design. In [STEVW74, STAYJ76] Stevens and Stay describe a modular, top-down design technique, which is used within IBM. In this technique the overall system function is iteratively decomposed into more distinct subfunctions. Using the Hierarchical-Input-Process-Output (HIPO) approach, each function at each level of the system structure is divided into three phases: input, processing, and output. In this way a modular, top-down design is achieved, and the set of HIPO charts, which were developed during the design, serve as system documentation.
Using a top-down design methodology there are many possible modularizations which will produce a correct system. However, certain modularizations will make the system easier to maintain and modify in later stages of its development. In [STEVn74] Stevens proposes the concepts of module strength and module coupling as two measures of the quality of a particular modularization. Module strength refers to the cohesiveness of each module. A high level of module strength is desirable because each module would implement a logically identifiable system subfunction. Module coupling refers to the complexity of the interface between modules and both the amount and type of data which one module must know about the other. A loose or low level of coupling is desirable because changes internal to one module are less likely to affect the other modules with which it communicates. Stevens concludes that high module strength and low module coupling characterize a system which is easily maintained, and he provides a set of guidelines to evaluate the degree of module strength and coupling for any particular system design.

Parnas [PAHN71, PARN72] developed an "Information Hiding Principle" and a design technique subsequently called "Parnas specifications". In this design methodology a list of necessary logical design decisions is formulated, and a separate module is defined to implement each of these decisions. Since a later design change will typically affect a single design decision, the change will be limited to a single module rather than the many modules which might be affected in other modularizations. The details of how a design decision is implemented within a module are hidden from other modules, so any change to these details will be invisible to the rest of the system. This design technique will produce clean module interfaces containing the minimum information necessary for communication. In addition each module will be more comprehensible, since it does not refer to the internal details of any other module. Parnas specifications, based on the information hiding principle, are widely used and have proved to be very successful in the development of large software systems.

A top-down design strategy has also been applied at the individual program level. Wirth [WIRTN71] has developed a program design technique called Stepwise Refinement. In this design strategy the function is decomposed into subtasks represented by processing specifications. The data needed by the program are refined into detailed data structures in parallel with the function decomposition. At each level of refinement certain design decisions must be made, and test data can be chosen to validate each of these decisions.
The major improvement in program design has been the development of a structured programming discipline. In a now famous letter, Dijkstra (DIJKE68A) observed that the density of GOTO statements correlates inversely with program quality. He therefore called for the elimination of the GOTO statement from all high level languages as a solution to the problem of poor programming practices. He argued that since human intelligence is better at grasping static relationships than dynamic processes, the static textual structure of a program should be as close to its dynamic execution as possible. Since the GOTO statement creates a wide gulf between the static and dynamic appearance of a program, it is a major obstacle in understanding programs. This observation led to much activity in trying to write GOTO-free programs, but unfortunately these programs were often less understandable than the programs they were meant to replace. The GOTO statement was incorrectly identified as the cause of unreadable programs, but a high density of GOTO statements is really the result of the more basic problem of poor program design. In (DENNP74) many papers develop this idea, and it is shown that high quality, structured programs can be written with GOTO statements used judiciously to implement the basic structure of the program. The evolution of the concept of structured programming beyond the simplistic solution of eliminating all GOTO statements has allowed us to focus on the underlying process of developing high quality, understandable, structured programs.

The management of a large software development project has traditionally been undisciplined and based on expediency. Recently many novel management techniques have been developed specifically for use in software development projects. Weinberg (WEING71) has analyzed the human communications problems of various types of programming team organizations, and in particular he discusses the dynamics of group interactions in software development. In (FAGAM70) Fagan discusses various techniques such as egoless programming, structured walk-throughs, and code inspections. Baker (BAKEF72) discusses a novel programming team organization called Chief Programmer Teams, which is in use within IBM. It is based on a specialization of function in the software development process. The team is headed by a chief programmer, who is responsible only for technical design decisions. All administrative details are handled by an administrative assistant, and the system documentation, in the form of a "program support library", is maintained by a programming secretary. In this way the technical people can spend their time programming, and the support tasks are handled by people trained specifically for those tasks.
A recent topic of much interest is the area of "Language Design for Reliable Software". The work in this area is basically an attempt to identify those aspects of current programming languages which, while providing powerful programming capabilities, also cause major problems of software reliability. The trend seems to be toward smaller and simpler languages consisting of less error-prone constructs. A recent conference on this topic (WORID77) provides many excellent papers on specification languages, new programming languages such as PASCAL and EUCLID, and the general problems of designing a language which will enhance the possibility of producing software which can be easily validated.

The field of software engineering encompasses many diverse research activities, and this discussion by no means constitutes a complete survey. Many conferences have been devoted to work in this field, and the proceedings of these meetings (IEEE73, IEEE75, HOROE75, FOXJ76) provide a fairly complete survey of current work. Software engineering is an active discipline at present, and the magnitude of the problems to be solved would indicate that it will remain so for many years.

2.2 Computer Program Verification

Every computer system suffers from a lack of reliable software. Too often operational programs are incorrect with respect to the function they are supposed to implement. Two approaches to this problem are being pursued by various research groups. In program testing, correctness is determined by observing the results produced for a well-chosen set of specific input values. On the other hand, the program verification technique attempts to mathematically prove program correctness by formally describing the semantics of the program.

Many early papers laid the conceptual framework for the later development of experimental program verification systems. In [NAURP66] Naur characterized the verification problem as proving that a dynamic description of a process leads from a static description of the state of the variables at the start of the process to a static description of the variable state at the end of the process. He proposed to represent the dynamic process as a sequence of static snapshots, each capturing the state of the variables at a particular point in the program. A program
would be verified if it could be proven that this sequence of snapshots leads logically from the input snapshot to the output snapshot.

Floyd [FLOY70] proposed a similar though more formal method which has been the basis for much of the subsequent applied work in the field. He hoped to be able to prove that a program is correct, terminates, and is equivalent to another program by formally defining the meaning or semantics of the program. In this inductive assertion technique a program is viewed as a directed graph, and an assertion is associated with each edge in the graph. Each statement, represented as a node in the graph, must map the assertions on each incoming edge to the assertions on each outgoing edge. In this way partial correctness, which means that the program is correct only if it terminates, can be proven. In addition termination can be proven if an entity can be formed which decreases monotonically during the execution of the program until it reaches a value which insures that the program will reach a termination point.

Total correctness consists of a proof of both partial correctness and termination. Floyd's inductive assertion approach uses a system of axioms and rules of inference based on the first-order predicate calculus. In addition he shows that the minimal set of assertions necessary to prove a program correct consists of an input assertion, an output assertion, and an assertion cutting each innermost loop in the program. This reduced set of assertions can be used to deduce an assertion for each edge in the program digraph. Therefore the amount of work necessary to produce a proof using an automatic verifier can be greatly reduced.

The formal work of Floyd was implemented by King [KING70, KING71] in his dissertation research. He produced a verifying compiler for a simple language based on a straightforward application of the inductive assertion method. This was the first attempt to apply the elegant theory of program verification, and it demonstrated some of the problems inherent in the method. In particular the set of necessary verification conditions quickly grew beyond the power of the theorem prover being used. King has since expanded his original work by integrating it into a highly interactive testing and verification system called EFFIGY [KING76, KING77]. He discusses the prospects for formal verification by saying

"...However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future."

(*)
The major practical problem encountered in these early program verification systems was that the number of program paths and verification conditions quickly grew beyond the capabilities of the theorem provers used. It was soon realized that this problem might be somewhat alleviated by using a higher level language with powerful array operators, thus reducing the number of program loops. APL is such a language, and Gerhart [GEiHS72] investigated using the inductive assertion method for the verification of APL programs. She found that a lot of the bookkeeping details necessary in other programming languages are incorporated within the high level operators in APL, and therefore programs tend to be simpler. As a result Gerhart was able to verify programs solving more difficult problems than those used in previous program verification research.

Laventhal [LAVEM74] extends the inductive assertion technique to programs which manipulate data structures. He precisely defines specific structural classes by structural invariants, which constitute an abstract description of the data class. Once the effects of each programming language statement on this abstract definition of the structural class are formulated, a proof that the program maintains these structural invariants can be easily incorporated into the inductive assertion technique. Example verifications of three programs manipulating singly-linked lists are provided, demonstrating that the technique can be extended to fairly complicated data structures.

At the same time that these verification systems were being developed, progress was also being made in extending the formal theory of program correctness. Manna [MANNZ69, MANNZ09A, MANNZ70] has been in the forefront of this work. He equates the concept of partial correctness with a proof of the satisfiability of a formula in the first-order predicate calculus; similarly, total correctness is equated with a proof of the unsatisfiability of a second formula. Manna extends this analysis to various classes of programs and utilizes these first-order formulas in studying various termination and program equivalence problems. The difference between partial and total correctness is proving that the program always terminates, and this termination problem can be analyzed separately. Sites [SITEi74] describes his techniques for proving that a program "terminates cleanly", by which he means that the program contains neither infinite loops nor semantic errors causing premature program termination. His results can be put to practical use in extending a proof of partial correctness for a program.

(*) KINJ76, p. 385.
correctness to a proof of total correctness. Of course the Turing machine halting problem prevents Sites' techniques from working in all cases, and indeed his method fails to prove termination when a loop exit condition is obscure or subtle. Related techniques and investigations are also reported by Von Henke [VONHF75], Waldinger [WALDR73], and Negureit [NEGEBB77].

One of the major obstacles to completely automatic program verification is the generation of the required set of loop invariant assertions. Katz and Manna [KATZ573, KATZ576] investigate various heuristic strategies for automatically generating candidate loop invariants. A different approach has been developed by Chacon [CHANC74], who proposes using variable values generated during test runs of the program to provide specific variable values which can be used in generalizing to loop invariant assertions.

A verification methodology combining inductive assertions and program testing has been investigated by Geller [CELL476]. To facilitate the use of test data in the verification process, he introduces the concept of distributed correctness, in which a part of the output assertion can be proven to be true at an intermediate point in the program, as long as it remains unchanged from that point until the end of the program. Geller uses actual test data to form a test data assertion, which then provides the basis for a generalization process. This generalization process produces a synthesized assertion, which can then be used as an inductive assertion in verifying the program. In addition to reducing some of the practical limitations to program verification, this work is important because it demonstrates that the techniques of program verification and program testing can be combined very effectively.

Detailed surveys of program verification have been compiled by Elspas [ELSPB72] and London [LONDON75]. In addition, a clear and concise description of the verification process is presented by Manna [MANNZ74]. It now seems unlikely that we will be able to prove nontrivial programs correct in the near future. However, advances in program verification continue, and they have been successfully utilized in hybrid verification/testing systems. In addition the insight gained from program verification research has helped us to better understand the programming process.
2.3 Computer Program Testing

Computer program testing has always been used to validate programs. Even though testing cannot provide absolute software reliability, our confidence in the correctness of a program will increase significantly after observing its correctness for a systematically chosen set of specific test data. For the near future program testing will continue to be a practical approach to the goal of software validation. Tanenbaum [TANEA76] discusses the failings of program verification while extolling the virtues and usefulness of program testing. In the past program testing has often been an undisciplined process based on the intuition of an individual programmer. Therefore a goal of research in this area is the systematic and scientific analysis of the testing process.

The first symposium devoted solely to the problem of software testing and validation took place in 1972, and the proceedings were later published as a book edited by Hetzel [HETZW73]. These proceedings represent the earliest survey of work in this particular area. Various papers covering such diverse topics as program proving, software design, programming language design, and the feasibility of software certification were presented in addition to many papers on the various aspects of program testing. Subsequent proceedings [IEEE73, IEEE75, HORO75] provide more recent surveys of the work currently being pursued. In addition more concise but unfortunately very narrow surveys by Huang [HUAJ75A] and Miller [MILLE77] have appeared in the literature. Research activity in this area is continuing at a high level and has produced a large volume of published material. The remainder of this section details the important developments which have taken place.

Unfortunately a sound theoretical base for program testing does not yet exist. A paper by Goodenough and Oenart [GOODJ75] is the most important contribution to date in the development of a theory of program testing. In this paper fundamental testing concepts are discussed, and a set of test data characteristics are rigorously defined. In particular, the importance of test data reliability is discussed, and possible causes of test data unreliability are demonstrated. These characteristics are used in formulating the Fundamental Theorem of Program Testing, which concisely states the conditions necessary for a totally effective test of a program. The set of definitions developed in this paper is gaining acceptance as a standard framework in which to discuss various testing strategies and represents the first formal description of the general
process of program testing. In this paper the authors also describe a practical strategy for program testing and propose possible directions for future research.

Hamlet [HAMIL75, HAML77] investigates the relationship between program testing, compiling with optimization, and computability theory. Within this framework he discusses many decidability problems which prevent program testing from being completely effective. He defines a "maximal test set" for a program fragment as one which will be affected by any nontrivial change in the fragment. In this case a nontrivial change is one which actually alters the function computed. Although maximal test sets do exist, it is impossible to find them in general. He describes a compiler implementation in which the set of test data, in the form of input-output pairs, is incorporated into the syntax of the program. This allows program testing to be done during compilation, but the test data must be selected manually. In [HAML76, HAML76A] Martín describes techniques for proving both the equivalence of two programs and the "conditional correctness" of a program. In proving conditional correctness he assumes that a set of programs is available which is known to contain a correct program for the function being implemented. In these techniques a finite set of test data is used to eliminate programs from the set until the single correct program remains.

As more experimental systems for automatically generating test data have been developed, certain fundamental problems have been identified. Gabow [GAB076] analyzes the complexity of algorithms for two problems that arise in automatic test path generation. He develops a highly efficient algorithm for building a path which passes through a specified set of flow graph nodes. Quite often we also want to test a path which passes through certain nodes but which completely avoids other flow graph nodes. Unfortunately, Gabow shows that the problem of building a path which satisfies these "impossible-pairs restrictions" is NP-complete. Other basic problems in test data generation are discussed by Kamamoororthy [RAMAC76]. In particular he proposes a heuristic solution to the problem of testing program paths whose predicates contain array references, with subscripts which depend on the value of input variables. These indeterminate array references are a major practical problem which must be solved in every test data generation system, and it is hoped that this heuristic can lead to efficient implementations.

Test data generation systems are currently limited to solving sets of linear constraints. Even though nonlinear constraints are generally rare, they are very common in
certain classes of programs, such as numerical analysis routines. The capability of existing nonlinear programming algorithms is severely limited, and more powerful techniques are needed to adequately test mathematical software. In [ELS87] Elspas describes techniques being investigated for the solution of nonlinear inequalities associated with program path constraints. In most cases a set of constraints will be a mixture of both linear and nonlinear equalities and inequalities. Linear-nonlinear decoupling is a technique in which linear and nonlinear constraints can be solved separately. In this way, the more powerful linear techniques can be used for the bulk of the constraints, and the less powerful nonlinear techniques are used only for the subset of nonlinear constraints. The power of existing linear techniques has also led Elspas to investigate their use for solving nonlinear inequalities. A nonlinear constraint can be replaced by a linear approximation if the nonlinear function is simple enough, e.g. quadratic, to be closely approximated by some linear function. Of course we must be careful that the solutions generated for the linear approximation also satisfies the original nonlinear constraint. By replacing as many of the nonlinear inequalities by linear approximations as possible and then using linear-nonlinear decoupling to isolate the remaining nonlinear constraints, Elspas hopes to be able to use existing nonlinear techniques to solve sets of constraints which previously were thought to be beyond the capability of these techniques. The goal of all current research in test data generation is limited in that any test case satisfying the conditions for a path is an acceptable solution. Therefore one goal of our research is to determine if certain test data points are more effective than others in testing a specific path.

Many hybrid debugging/testing/verification systems based on the concept of symbolic program execution have been investigated and implemented. Symbolic program execution is a technique in which the values assigned to input variables are symbols, such as the input variable names themselves, rather than specific numerical quantities as in normal execution. The values of all other variables are then stored in the form of symbolic arithmetic expressions in terms of the input variables. At the completion of the symbolic execution of a specific program path, the values of both the complete set of output variables and the path condition (*) are also generated as symbolic formulas over

(*) The path condition represents the total set of constraints which must be satisfied for the path to be executed. It consists of the conjunction of the conditions produced at each branch point on the path.
the input symbols. Symbolic output values offer the advantage of a general representation of the computation done on the path rather than a specific numerical result as in traditional program testing. To circumvent the limitations of his verifying compiler, King and his associates at IBM Research have implemented an experimental symbolic execution system called EFFIGY [KING75, KING76]. EFFIGY is basically a symbolic manipulator for a small integer sublanguage of PL/I. A user must specify the path to be tested interactively by selecting an alternative at each branch point encountered in the program. Once selected, this path can be tested either with real data supplied by the user or with symbolic data. King has made use of his earlier work by incorporating a limited verification capability into EFFIGY, based on a package of analysis routines from his verifying compiler. EFFIGY will attempt to verify a particular path or segment of a path with assertions supplied by the user. However, its verification capability is limited, and some very complex path segments will not be verified even though they may be correct. EFFIGY represents a major advance in the capabilities offered by integrated software reliability tools, but this type of system must be extended to larger and more useful programming languages before it can be incorporated into the software development cycle.

Symbolic execution systems have also been developed for FORTRAN and LISP programs. Clarke [CLAIR76] has developed a system to generate test data and symbolically execute FORTRAN programs. The program testing capabilities of this system are basically similar to those of the EFFIGY system, and in addition real test data for a user specified path can be automatically generated if the path condition consists entirely of linear constraints. In this system, the capability to detect certain common programming errors is enhanced by the use of "artificial constraints". In this technique extra constraints are added to the path condition forcing the system to try to find input data for which the programming error would occur. Boyer [BOYER75] describes the SELECT system, which is a hybrid verification/testing system for LISP programs. For a specific execution path the system can verify that the path is correct with respect to a user-supplied output assertion for the path. SELECT also generates simplified symbolic values for both the output variables and the input data constraints which cause the path to be executed. In addition if these constraints are linear, the system will select actual input values which can be used to test the path. Howden [HODD77] reports on still another similar system called DISSECT. One significant advance incorporated in this more recent system is the capability to store a sequence of DISSECT commands for a
particular program. In this way long sequences of commands do not have to be reentered to retest a program after each debugging phase. In addition some primitive control structures are provided in the command language to allow a user the flexibility to alter the course of the test run dynamically. These systems have proved to be a fruitful area of research, and it is hoped that their development will continue, further enhancing their usefulness and applicability.

Howden has been very active both in system development as described above and in more theoretical research. In [HowDh75, HowDh75A] he defines a Path Analysis Testing Strategy as a general class of strategies in which test data is generated for a subset of the paths in a program. This subset can be selected according to various criteria, and Howden describes one such set of rules. Basically he divides the complete set of program paths into a number of classes, and test data are generated for a single path in each class. The set of rules defining these classes focuses on the loops in the program. A path constitutes a boundary test of a particular loop if the loop body is executed only once (no iteration); an interior test of a loop is one in which the loop body is executed at least twice. Two paths belong to different classes if any of the following conditions apply.

(1) The paths differ other than in the way they execute loops.

(2) One path is a boundary test of a loop while the other is an interior test.

(3) The paths enter or exit a loop differently.

(4) Both are boundary tests of a loop but follow different paths through the loop body.

(5) Both are interior tests of a loop but follow different paths through the loop body on the first iteration of the loop.

Howden claims that this set of rules will produce an intuitively complete set of test data. Although this set of path selection rules is intuitively appealing, it fails to address many important issues. For example, the fifth rule does not differentiate between paths which execute a loop differently on other than the first iteration. Howden has formulated a reasonable starting point for this problem, but
a thorough analysis of the path selection process is needed.

In a later paper Howden [HOWD76] analyzes the reliability of a general path analysis testing strategy. He defines a reliable testing strategy as one for which the correctness of the program can be inferred with complete confidence from the correct execution of the test cases. This broad definition of reliability should not be confused with the more restricted definition used by Goodenough and Gerhart. In [GOODJ75] they define a reliable test data selection criterion as one which insures selection of tests which are consistent in their ability to reveal errors, as opposed to necessarily using able to detect all errors. Howden defines three classes of errors which can occur for a specific path, and he analyzes the reliability of the strategy for each class. A computation error is one in which the path is executed for the correct set of input values but computes the wrong function. A domain error is one in which the computation is correct, but an error in a control flow statement causes the path to be executed for the wrong set of input values. Finally, a superset error is one in which some input values execute this path which should actually follow some other path. To obtain these results Howden has to assume that every path is tested, and even then his reliability results are not general for all domain errors. Although the generality of these necessary assumptions show that the path analysis testing strategy cannot be completely reliable in practice, this work represents a significant advance in the development of a formal basis for program testing. Howden's results demonstrate that the effectiveness of a testing strategy greatly depends on the type of error involved. Therefore the particular error classification used is important in the analysis of any testing strategy. Other error classification schemes have been formulated by Amory [AMOR75] and Goodenough and Gerhart [GOODJ75, GERH76], but Howden's scheme seems the most natural for program testing and is very similar to the error classification used in our research.

Many other systems and techniques have been developed to help validate software. In [MILLE74, MILLE75, MILLE77] Miller describes RXVP, an automated verification system for FORTRAN programs. Its goal is to select a set of paths such that each possible outcome of each branching point in the program is executed by at least one of the selected paths. However, test data for these paths must be constructed manually by the user. Goodenough and Gerhart [GOODJ75] and Walsh [WALSH77] describe two test data methodologies which use similar types of decision tables. In this approach each condition or combination of conditions relevant to the
correct operation of the program is determined both from the program specifications and from the internal structure of the program itself. Each of these conditions is entered into the table, along with the computations desired for that particular case. Test data for each of these table entries must then be selected.

Current software reliability research which is based on sound theory, such as the work on program verification, has proven very difficult to apply in practice. On the other hand, practical approaches, such as the various program testing systems and techniques currently being used, are very informal, and the extent of their error detection capability cannot be precisely determined. With the insight gained from the theoretical work and the experience gained through the practical work we are now ready to pursue research which is both analytical and practical. This work should be based on a formal analysis of the problem, and it must also be capable of being put into practice. These goals seem to be the essence of what the discipline of software engineering aspires to achieve.
CHAPTER 3
BASIC CONCEPTS AND DEFINITIONS

In this chapter the basic concepts used in this research are defined and discussed. In the first two sections we describe a simple programming language and a standard program form which will be used in the development of programming examples. In section 3 a program "path" is defined, and aspects of this important concept are discussed. Section 4 is a brief description of the standard technique of symbolic path execution, and various types of path predicates are discussed in section 5. A model of program computation is presented in section 6, and the concept of a program input space is also introduced.

3.1 A Sample Programming Language

In designing this language we have attempted to keep it simple, concise, and easy to use while maintaining sufficient power to program nontrivial algorithms. The language is structured and consists of a few very powerful control flow constructs. There is no provision for any sort of block structure, and only elementary I/O capabilities are provided. Since the statements available in this language are typical of high level languages, a brief discussion of these statements should be sufficient for the reader to understand later program examples.

Only those features which are needed to write example programs and program segments have been included in this language. Therefore a single data type is used, and singly-subscripted arrays of these variables are allowed. This data type is real-valued, and we assume that a value is truncated when it is used for variables, such as subscripts and DO-loop indices, which are usually thought of as integers. We have found that this single data type has been adequate for the purposes of our research. However, we realize that our results have to be extended to other
numeric and nonnumeric data types before they can be applied to real problems.

Input/output is very machine-dependent, and since it is not central to this research, only the most elementary I/O capabilities are provided in this language. Input is performed by a simple READ statement; output is accomplished with a similar WRITE statement. These two statements allow lists and vectors of variables to be read or written. However, no specific information relating to data format or hardware environment is required. Examples of these statements are provided in the program in figure 1.

The only types of control flow allowed in this language are sequence, alternation, and iteration. Therefore every program must necessarily be structured. All computation is done with an arithmetic assignment statement in which a single variable is assigned a value. The right hand side of an assignment statement is an arithmetic expression using variables, constants, and a set of basic arithmetic operators (+, −, *, /). The standard operator hierarchy is observed, and parentheses can be used to override the standard hierarchy in the usual way. Assignment statements provide the basic sequential flow of control.

An alternation type of control flow is achieved by using the IF-THEN-ELSE-ENDIF construct. The conditional associated with the IF statement is a general predicate consisting of a Boolean combination of arithmetic relational expressions. Any well-formed program segment, including the null program segment, can be used in the THEN and ELSE options of the IF construct. The ENDIF statement is just a delimiter for the IF construct. It makes the nesting structure of a program clearer and eliminates any possible problem with an ambiguous ELSE clause.

A very general and powerful iteration construct is included in the language. It consists of a DO statement, loop body, and ENDDO delimiter. The DO statement can be in one of three forms:

1. DO I=INIT,FINAL,INCR;
2. DO WHILE (predicate);
3. DO I=INIT,FINAL,INCR WHILE (predicate);

The loop body can be any well-formed program segment, and the ENDDO is just a delimiter to clarify the iteration structure of the program.
(1) READ (XIN(I), I=1, SUBMAX), VALSRCH
(2) LOBOUND = 1
(3) HIBOUND = SUBMAX
(4) VALFIND = 0
(5) DO WHILE ((HIBOUND ≥ LOBOUND) AND (VALFIND = 0))
(6) MIDVAL = (HIBOUND + LOBOUND)/2
(7) IF XIN(MIDVAL) > VALSRCH *MIDVAL TRUNCATED*
(8) THEN HIBOUND = MIDVAL - 1
(9) ELSE IF XIN(MIDVAL) < VALSRCH
(10) THEN LOBOUND = MIDVAL + 1
(11) ELSE VALFIND = MIDVAL
(12) ENDIF
(13) ENDIF
(14) ENDDO
(15) WRITE VALFIND;

Figure 1: A Binary Search Program
No unconditional transfer statement has been included in the language, and therefore statement labels are unnecessary. The exclusion of a GOTO statement and the form of the language constructs lead naturally to well-structured programs. This restriction has not proven to be a hindrance in developing program segments illustrating the various concepts and results of this research.

The generality of the structures allowed within DO-loops and IF constructs provides for both nested loops and nested IF structures. Loops and IF structures can also be nested within each other. In addition, blocks of assignment statements can be used anywhere a single assignment statement is allowed. These rules permit a wide variety of expressive control flows and can be used to write nontrivial programs such as the example of a binary search program in figure 1. In this example XIN is the sorted vector to be examined; VALSRCH is the value to be searched for; and VALFIND is the index of the element in XIN equal to VALSRCH. If the search fails, VALFIND will be zero. In this example the input space has a dimensionality of SUBMAX + 1, and the output space has a dimensionality of one.

3.2 The Standard Program Form

In order to simplify later discussions and analyses we define a standard program form by introducing certain syntactic restrictions into the language. The variables used in a program are divided into three classes. If a variable appears in a READ or WRITE statement, it is classified as an input or output variable respectively, and all other variables are called program variables. However, any variable used only to index an array in a READ or WRITE statement is not considered to be an input or output variable. There are restrictions on where each of these variable types can be used. Input variables cannot appear on the left hand side of assignment statements. Similarly, output variables can only be used on the left hand side of assignment statements and in WRITE statements. Of course by definition program variables will never appear in READ or WRITE statements. We are imposing these additional restrictions in order to produce a clear delineation between the three types of variables and to discourage the use of a single variable for multiple purposes.
In order to keep the three phases of a program separate, restrictions on the placement of I/O statements are also imposed. A program can contain only a single READ statement which inputs the complete set of input variables. This READ statement must appear as the first statement in the program, and every other statement must be reachable from this READ statement along some executable path. Similarly, a single WRITE statement is used to output the complete set of output variables. This WRITE statement must be the last statement in the program, and it must be reachable from every other statement in the program along some executable path. With this set of restrictions each program will exhibit a standard form in which the input, processing, and output phases are discernible, and the sets of input and output variables are easily identified.

The assumption of a standard program form facilitates the presentation of the domain testing strategy but does not restrict the type of programs which can be tested. Many programs do not exhibit an input-process-output structure, and we must be able to apply our methodology regardless of where the input and output statements are located. In general the sequence of input variables used by a program depends on the particular path executed. For example, consider the following short program.

```
READ I;
IF I < 2
    THEN READ J;
ELSE J = 0;
ENDIF;
K = 2*I - J;
WRITE K;
```

On the path executing the THEN branch, the input variables are I and J, but on the path through the ELSE branch only variable I is used as input. In addition, the decision to input the second variable depends on the first input value. Therefore, the assumption that all input is performed at the beginning of execution is a major restriction when considering the entire program. However, our methodology is path-oriented, and test data are generated for one path at a time. When considering a single path, the sequence of input variables is completely determined, and therefore all input could be done at the start of the execution of the path. For example, one of the paths in the sample program is listed below on the left, and an equivalent sequence of
statements meeting the standard form of input-process-output is listed on the right.

```
READ I;
   I < 2 (condition)
   READ J;
   K = 2*I - J;
   WRITE K;
```

Since the sequence of input variables for a particular path is completely determined, we can transform the path into an equivalent path in the input-process-output format. So, in summary, the assumption of a standard program form can be made without loss of generality, since any path selected for testing can be transformed into an equivalent path in which all input is performed before any processing takes place.

### 3.3 Program Paths

The concept of a "path" is central to many types of program analysis, since the overall computation that a program performs consists of the computations performed on its many paths. Therefore in studying a program its set of paths provides a functional decomposition which is both natural and complete. A clear understanding of the program path concept is particularly important in program testing since most strategies are based on testing particular program paths.

Before formulating a definition for a program path we must develop a representation of the program as a directed graph. In general a directed graph $G=(V,A)$, where $V$ is a set of nodes or vertices and $A$ is a set of arcs or directed edges between elements of $V$. (*) In our sample language we define a set of basic program elements consisting of each READ, WRITE, assignment, IF, and DO statement and in addition each ENDIF and ENDDO delimiter. The directed graph representation of a program will contain a node for each occurrence of a basic program element and an arc for each possible flow of control between these elements. While THEN and ELSE statements do not appear in the digraph, the actions associated with them will be represented as nodes in

(*) HARAF69, p. 198.
the digraph. This digraph is a complete representation of the possible control flows in a program. Figure 2 presents an example digraph for the binary search program in figure 1.

We can sequentially number each basic element in the program, and the set of nodes in the digraph is \( V(1), V(2), \ldots, V(k) \). A control path is then defined to be a "walk" in the digraph starting with \( V(1) \), the initial node which is a READ statement for the \( N \) input variables, and ending with \( V(k) \), the final node which is a WRITE statement for the \( M \) output variables. A "walk" in a digraph is defined as an alternating sequence of nodes and arcs \( V(1)A(1)V(2)A(2) \ldots A(k-1)V(k) \) where each arc \( A(i) \) is directed from node \( V(i) \) to node \( V(i+1) \). (*) Therefore, it should be noted that two walks which differ only in the way that they execute a particular loop in the program will be defined to be two different control paths for our purposes. Using this definition the number of control paths in a program can be potentially infinite.

The path condition is the compound condition which must be satisfied by the input data in order for the control path to be executed. It is the intersection of the individual conditions which are generated at each branch point along the path, and therefore each control path has a distinct path condition. Input data which execute a path must simultaneously satisfy each individual condition in the compound path condition. If any input data exist which satisfy the path condition, the control path is also an execution path and can be used in testing the program. Unfortunately, a path condition might be unsatisfiable since the individual conditions from which it is formed may be mutually contradictory. In this case the control path can never be executed and is of no interest in testing the program. Therefore, an infeasible path is defined as a control path in the program digraph which is not executable because its path condition cannot be satisfied by any possible input values. Path infeasibility is a major problem since we might waste a lot of time trying to test a path before we discover that it is infeasible.

(*) HARAF69, p. 198.
Figure 2: Digraph Representation of Binary Search Program
3.4 **Symbolic Path Execution**

Symbolic path execution is basically a technique in which symbols are manipulated rather than actual numeric values. Its main advantage is that it produces a general representation of the computation performed on a path rather than numerical results for specific cases. The concept is important in program testing because it can be used as a mechanism to generate and analyze path conditions and individual data constraints.

When a path is executed symbolically, input variables are given symbolic values, in the form of character strings, rather than specific numeric values. These symbols should have some meaning for the user and are usually chosen to be the actual names of the input variables themselves. During the execution of a path the value of each variable is stored as a symbolic arithmetic expression in terms of input variables, constants, and the set of operators. When an assignment statement is executed the symbolic value of the left hand side variable is formed from the right hand side expression after each right hand side variable has been replaced by its current symbolic value, which in general will also be a symbolic expression. Of course, various neuristics can be applied to simplify the symbolic formula thus produced. At the end of the path the value of each output variable is generated as a long and complicated symbolic formula. In addition a symbolic representation of the path condition can also be generated.

The concept of symbolic variable values is used extensively in program testing. In general each predicate condition encountered along the path will be expressed in terms of program variables and input variables. However we are really interested in an equivalent constraint expressed in terms of only input variables. To find this equivalent constraint we have to replace each program variable by its symbolic equivalent in terms of input variables. Symbolic path execution provides this capability and can be used to generate test data automatically. In figure 3 the symbolic execution of a path through the THEN branch of the first and second IF constructs and the ELSE branch of the last IF construct is traced.
READ A, B;

(1) \[ C = 2A + 3B - 4 \]

(2) IF \( C > 3 \)

\[ \text{THEN } D = 2C - 2 \]

ELSE \( D = 1 \)

ENDIF;

(3) \[ E = D/2 + A + 1 \]

(4) \[ C = (E + D)*3 \]

(5) \[ F = C - o*D + E + A + o*B - 14 \]

(6) IF \( F = 3 \)

\[ \text{THEN } D = U - C \]

ELSE \( D = U + C \)

ENDIF;

(7) \[ G = E + 3 \]

(8) IF \( G > -1 \)

\[ \text{THEN } H = G \]

ELSE \( H = D + o*E - A + 3*B - 7 \)

ENDIF;

(9) WRITE F, G, H;

**SYMBOLIC EXECUTION**

<table>
<thead>
<tr>
<th>Pt</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2A + 3B - 4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>2A + 3B - 4</td>
<td>4A + 6B - 10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>2A + 3B - 4</td>
<td>4A + 6B - 10</td>
<td>3A + 3B - 4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>2A + 27B - 42</td>
<td>4A + 6B - 10</td>
<td>3A + 3B - 4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2A + 27B - 42</td>
<td>4A + 6B - 10</td>
<td>3A + 3B - 4</td>
<td>A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>2A + 27B - 42</td>
<td>-17A - 21B + 32</td>
<td>3A + 3B - 4</td>
<td>A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>2A + 27B - 42</td>
<td>-17A - 21B + 32</td>
<td>3A + 3B - 4</td>
<td>A</td>
<td>3A + 3B - 1</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>2A + 27B - 42</td>
<td>-17A - 21B + 32</td>
<td>3A + 3B - 4</td>
<td>A</td>
<td>3A + 3B - 1</td>
<td>-</td>
</tr>
</tbody>
</table>

Path Condition---

\((2A + 3B > 4) \text{ AND } (A = 3) \text{ AND } (3A + 3B \leq 0)\)

*Figure 3: Symbolic Execution of A Program Path*
3.5 Path Predicates

Each path in a program passes through a set of branch points, and each of these branch points has a conditional expression associated with it. The branch which will be followed at a particular branch point depends on whether the associated conditional expression is true or false. On a particular path the predicate produced at a branch point is either this conditional expression or its complement depending on which branch is taken. The path condition can then be viewed as the conjunction of the complete set of predicates or conditions which must be satisfied so that the conditional expression evaluated at each branch point causes the correct branch to be taken. Path predicates are important in this work since in testing a specific path we must generate input data which satisfies the set of predicates simultaneously.

In our sample programming language the conditional expression in an IF statement and each DO statement will produce predicates. For example consider the following IF statement, where S1 and S2 are program segments.

```
IF A ≤ B+3
  THEN S1;
ELSE S2;
ENDIF
```

The predicate \( A ≤ B+3 \) must be satisfied for any path taking the THEN S1 branch. Since \( A ≤ B+3 \) must be false for each path taking the ELSE S2 branch, the predicate \( A > B+3 \) must be satisfied in this case. For a DO statement such as:

```
DO I=INIT,FINAL,INCR;
```

the predicate produced would be either \( I ≤ FINAL \) or \( I > FINAL \) depending on whether we wanted to iterate or exit the loop respectively. For a WHILE type of DO-loop such as:

```
DO WHILE (D=A-B+2);
```

the predicate would be \( D=A-B+2 \) when executing the loop body again and \( D\neq A-B+2 \) when the execution of the DO-loop has been completed. Of course in the form

```
DO I=INIT,FINAL,INCR WHILE (D=A-B+2);
```

the predicate \( ((I ≤ FINAL) AND (D=A-B+2)) \) will be produced if we want to execute the loop body again. The predicate \( ((I > FINAL) OR (D ≠ A-B+2)) \) must be satisfied in order to
exit a DU-loop of this type.

The general predicate form is a Boolean combination of arithmetic relational expressions. The logical operators OR and AND are used to form these Boolean combinations. Each arithmetic relational expression contains a relational operator from the set \( (>, \leq, =, \geq, \neq) \), which form three complementary pairs \( (= \& \neq), (\leq \& \geq) \), and \( (> \& \leq) \). The negated operators \( \approx, \neq, \not\leq, \not\geq \) are superfluous because each of them is equivalent to one of the allowed operators \( (i.e., \approx \) is equivalent to \( \not\leq \)). By the same reasoning the logical operator NOT is also unnecessary. If a predicate consists of two or more relational expressions, then we call it a compound predicate. A simple predicate consists of just a single relational expression. In evaluating compound predicates the usual hierarchy of AND before OR is assumed, and of course parentheses can be used to override the standard hierarchy.

The linearity or nonlinearity of predicates is an important property since it is very difficult to solve a general set of nonlinear predicates. The following standard form describes a simple predicate and is also the form for each expression in a compound predicate.

\[
T(1) + T(2) + \ldots + T(m) \ ROP \ K
\]

In this form \( T(1), i=1, \ldots, m \) are general terms, \( ROP \) is a relational operator \( (=, \neq, <, \leq, >, \geq) \), and \( K \) is a numeric constant. Each term \( T(i) \) is of the form:

\[
P(i,1) \quad P(i,2) \quad P(i,n) \\
A(1)X(1) \quad X(2) \quad \ldots \quad X(n)
\]

where \( A(i) \) is a numeric coefficient, the \( X(j) \)'s are variables, and the \( P(i,j) \)'s are the exponents of the variables. If any \( P(i,j) \) is zero, its variable will in general be omitted. The general \( i \)-th term of a predicate is linear when \( P(i,j) = 0, j=1, \ldots, g-1, g+1, \ldots, n \) and \( P(i,g) = 1 \). In other words the term can contain just a single variable, and in addition this variable must be raised to the first power. A linear predicate is then defined as one in which each term is linear. Some examples of linear and nonlinear predicates follow.
The above discussion of predicate linearity must be extended to compound predicates. In general a compound predicate is linear when each of its component simple predicates is linear. A path condition formed by simple predicates defines the set of input data which will cause the path to be executed. However, a path condition containing a compound predicate of the form \( [C(i) \text{ OR } C(i+1)] \) defines the union of two sets of input data. For example \( C(1) \text{ AND } [C(2) \text{ OR } C(3)] \text{ AND } C(4) \) is equivalent to \( [C(1) \text{ AND } C(2) \text{ AND } C(4)] \text{ OR } [C(1) \text{ AND } C(3) \text{ AND } C(4)] \). In this case the set of input data satisfying the path condition is the union of sets of data, each described by one of the conjuncts in the path condition. So if the compound predicate \([C(i) \text{ OR } C(i+1)]\) is nonlinear, but either of the simple predicates constituting it is linear, at least one of the solution sets for the path will be described by linear predicates.

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 only input variables. If we replace each program variable appearing in the predicate by its symbolic value in terms of input variables, we get an equivalent constraint which we call the interpretation of the predicate. A particular interpretation is equivalent to the original predicate in the sense that input variable values satisfying the interpretation will lead to the computation of program variables satisfying the original uninterpreted predicate. A single predicate can appear on many different execution 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. However, a predicate can have the same interpretation on two different paths if the variables appearing in the predicate are computed by the same subsequence of assignment statements on both paths. The following program segment provides example predicates and interpretations.
READ A, B;

IF A > B
    THEN C = B + I;
    ELSE C = B - I;
ENDIF;

D = 2*A + B;

IF C <= 0
    THEN E = D;
    ELSE E = 0;
ENDIF;

IF D = 2
    THEN F = E + A;
    ELSE F = E - A;
ENDIF;

WRITE F;

In the first predicate, \( A > B \), both \( A \) and \( B \) are input variables, so there is only one interpretation. The second predicate, \( C \leq 0 \), will have two interpretations depending on which branch was taken in the first IF construct. For paths on which the THEN \( C = B + 1 \) clause is executed, the interpretation is \( B + 1 \leq 0 \) or equivalently \( B \leq -1 \). When the ELSE \( C = B - 1 \) branch is taken, the interpretation is \( B - 1 \leq 0 \), or equivalently \( B \leq 1 \). Even though the third predicate, \( D = 2 \), appears on four different paths, it only has one interpretation, \( 2*A + B = 2 \), since \( D \) is assigned the value \( 2*A + B \) in the same statement on each of the four paths.

3.6 A Model of Program Computation

A program which reads a set of \( N \) input variables and writes a set of \( M \) output variables is said to map from an \( N \)-dimensional input space to an \( M \)-dimensional output space. Each set of specific values for the \( N \) input variables is represented as a single point in the \( N \)-dimensional input space, and similarly each set of specific values for the \( M \) output variables represents a single point in the \( M \)-dimensional output space. In this model the \( N \)-dimensional input space is the domain of the function computed by the
program, and the $M$-dimensional output space is its range.

The computation performed by a program is usually modeled as a single function mapping each point in the $N$-dimensional input space to a point in the $M$-dimensional output space. This model is shown conceptually in the upper diagram of figure 4. Every point in the domain will be mapped to some point in the range, so the function is total. However, since two distinct inputs can produce the same output and not every output value need ever be produced, the function is not necessarily one-to-one nor onto. This simple model represents a program as a single, monolithic entity without regard for any of its internal structure and is not sophisticated enough for our purposes.

The model we will use views the program as a set of executable paths. The computation performed on any one of these paths is called a transformation, and therefore the overall computation which a program performs is represented as a set of transformations rather than just a single mapping. The transformation for a particular path will be applied to those input values which satisfy the path condition and cause the path to be executed. This set of input values forms a domain of the $N$-dimensional input space. Therefore the input space is partitioned into domains which are determined by the predicates in the program. Since the points in each domain can be mapped to any points in the output space, there is no corresponding structure imposed on the output space. This complicated model is shown in the lower diagram of figure 4.

In summary, a program is composed of a set of executable paths, and each of these paths will be executed for data points in the domain of the input space which is defined by the path condition. Finally, the computation performed for data points in this domain is represented as a transformation defined by the sequence of assignment statements on the path. The input space structure and its relationship to the program is central to the development of a program testing strategy and is analyzed in depth in chapter 4.
Figure 4: Models of Program Computation
CHAPTER 4

INPUT SPACE STRUCTURE

This chapter provides a detailed discussion of various important aspects of program input space structures. The general characteristics of the total input space and of a single domain are described in the first two sections respectively. In section 3 the relationship between various types of predicates and the resulting shape of the domain is explored. Predicates which do not produce orders of an input space domain are discussed in section 4, and this leads to an analysis of infeasible paths in the next section. Finally, a sample program and its input space partitioning structure are developed in section 6.

4.1 The Input Space

To test a program we must select input data which exercise certain aspects of that program. Therefore the relationship between input data values and the actual program code is of utmost importance in this research. Since the program testing strategy we have developed is based on the structure of input data spaces, a detailed understanding of the general characteristics of input spaces is necessary.

The complete set of possible input values form an \( n \)-dimensional Euclidean space, where \( N \) is the number of input variables. If an array of values is input, each element of the array is considered to be a separate input variable. For a program in the standard form the dimensionality of the input space can easily be determined from the number of variables in the READ statement at the beginning of the program. Each point in the input space represents a particular set of values for the \( N \) input variables, and the value of each variable can be viewed as one of the \( N \) coordinates of the point.
We assume that every input space is bounded in each direction by the minimum and maximum values the corresponding variable can have. This property is reflected in the path condition by a pair of min-max predicates for each of the N input variables such as \( X \geq \text{MIN} \) and \( X \leq \text{MAX} \). These min-max constraints do not appear in the program but are automatically appended to each path condition. Since a single data type is used for all variables in our language, each variable will have the same min-max constraints, and therefore the input space is an N-dimensional cube in this case. A bounded input space implies that the input space domain for each path is also bounded.

An important property of the input space is whether it is continuous or discrete. In an idealized theoretical machine a real-valued data type would imply a continuous input space, but in reality the finite size of data representations means that only a discrete subset of real values can actually be represented. In this research we assume an idealized continuous input space to develop results which can then be extended to the more complicated and realistic discrete case.

4.2 Input Space Domains

The input space is partitioned into a set of domains. Each domain corresponds to a particular executable path in the program and consists of the input data points which cause the path to be executed. More formally, an input space domain is defined as a set of input data points satisfying a path condition consisting of a single conjunction of simple predicates. (*) To insure that each input point belongs to some domain we have assumed that infinite loops and premature termination errors, such as overflow, have been eliminated before testing. Therefore each input causes an execution path to be completed, and the partitioning of the input space is also exhaustive.

The input space domain for a particular path is defined by the path condition in that every point in the domain must satisfy the path condition. Since the path condition is the

(*) A path containing only simple predicates corresponds to one input domain. However, a path containing compound predicates of the form \( C(1) \lor C(2) \) may correspond to more than one domain.
intersection of predicate interpretations, the domain can also be viewed as the intersection of the solution sets of these predicate interpretations. The boundary of the domain is defined by the individual predicate interpretations and consists of **border segments**, where each segment is the section of the boundary determined by a single simple predicate in the path condition.

These border segments are determined by the particular predicate interpretations on the path, not by the original predicates which appear in the program. Even the form of the border segment does not depend on the original predicate. For example, with input variables A and B, the linear predicate, \( A < C + 2 \), can lead to a nonlinear border segment, \( A < B^2 + 2 \), when \( C = B^2 \). Similarly, a nonlinear predicate, \( C > A^2 + B \), will produce a linear border segment, \( A > B \), when \( C = A^2 + A \). So, in conclusion, the partitioning structure of the input space is completely determined by predicate interpretations, and in discussing borders and predicates we are referring to the particular interpretations of the predicates.

The boundary of a domain is not necessarily part of the domain itself. In general the boundary consists of many border segments, and each of these segments can be open or closed independently of the others. A **closed border segment** is part of the domain and is formed by predicates with \( \leq \), \( \geq \), or \( = \) operators. An **open border segment** forms part of the domain boundary but is not actually part of the domain itself and is formed by \( < \), \( > \), and \( \neq \) predicates. Of course a border segment which is open for one domain is closed for some adjacent domain or domains since the points on the border must lie in some domain.

The general form of a simple, linear predicate interpretation is

\[
A(1)X(1) + A(2)X(2) + \ldots + A(n)X(n) \text{ ROP } b
\]

where the relational operator (ROP) can be \( < \), \( \leq \), \( > \), \( \geq \), \( = \), or \( \neq \). However, the border segment which any of these predicates defines is a section of the surface defined by the equality

\[
A(1)X(1) + A(2)X(2) + \ldots + A(n)X(n) = b
\]

since this is the limiting condition for the points satisfying the predicate. In a general \( n \)-dimensional space this linear equality defines a hyperplane, which is the general \( n \)-dimensional geometric equivalent of a plane. The boundary of a domain described by linear predicates consists
of hyperplane segments, each of which is determined by a single, simple predicate in the path condition.

The input space domain can be viewed as the intersection of the domains satisfying the individual predicate interpretations. Predicates expressed as $<$ or $>$ relationships define domains which are open half-spaces, those with $\leq$ or $\geq$ define closed half-spaces, an equality is satisfied by points on a hyperplane, and a nonequality ($\neq$) predicate defines two open half-spaces. A convex set is one in which for any two points in the set, the line segment joining the points is also completely contained in the set. Half-spaces and hyperplanes are convex sets, and since the intersection of convex sets is also a convex set, an input space domain is a convex set if the path condition is linear. (*) So in general an input space domain is a convex polytope bordered by hyperplane segments, each of which may be open or closed. Of course, since a nonequality predicate defines two open half-spaces, the input domain can also be the union of convex polytopes. However we can view this type of domain as a single convex polytope with slices, defined by hyperplanes, missing from it. Domains made up of multiple convex polytopes can also be the result of compound predicates of the form $(C(i) \text{ OR } C(i+1))$.

4.3 Predicate Types and Domain Characteristics

While domains which are defined by linear predicates are known to be convex sets bounded by hyperplane segments, no simple characterization of nonlinear domains can be formulated. Nonlinear surfaces can be arbitrarily complex, and domains which are constrained by nonlinear predicates are very difficult if not impossible to analyze. Therefore the following discussion is limited to domains and borders formed by linear predicate interpretations.

A very simple example will serve to demonstrate the form of the domains which satisfy predicates expressed with the various relational operators. In figure 5 a two dimensional input space with input variables $I$ and $J$ is bounded by the following min-max constraints.

$$I \geq 0 \quad I \leq 10 \quad J \geq 0 \quad J \leq 10$$

(*) HADLGol, p. 202-204.
Figure 5: Predicate Types and Regions
In addition the line, \( I - J = 2 \), divides the space into three parts, two open half-spaces and the line itself. Area 1 is the open half-space where \( I - J < 2 \), area 3 is the open half-space where \( I - J > 2 \), and area 2 is the hyperplane where \( I - J = 2 \). Also the predicate \( I - J \leq 2 \) corresponds to the closed half-space consisting of areas 1 & 2, the predicate \( I - J > 2 \) is satisfied by points in the closed half-space consisting of areas 2 & 3, and the nonequality predicate is associated with the two open half-spaces 1 & 3. This figure also shows how a domain defined with a nonequality predicate can be viewed as a single domain with the hyperplane (slice) \( I - J = 2 \) missing.

The exterior boundary of an input space domain is that part of the boundary which is determined solely by the inequality constraints. Each border segment in the exterior boundary is a segment of a hyperplane defined by the constraint with the inequality operator replaced by \( = \). For example a constraint such as \( I + 2*J \leq 3 \) will produce a border segment which is part of the hyperplane \( I + 2*J = 3 \), since this is the limit of the set of points satisfying the inequality. The limits of the hyperplane segment which actually forms part of the border depend on the way in which the hyperplane intersects the hyperplanes produced by other inequality predicates. In figure 6 the solid lines represent a set of two-dimensional hyperplanes forming the exterior boundary of a domain. Again each of these segments can be open or closed depending on whether the predicate is a strict inequality or not.

While inequalities define the exterior boundary of a domain, each nonequality \( (\neq) \) constraint produces an interior border. The constraint \( A \neq b \) is equivalent to the compound predicate \((A < B) \text{ OR } (A > B)\). In this form it is clear that the addition of a nonequality predicate to a set of inequalities can split the domain defined by those inequalities into two regions, as shown by the dashed lines in figure 6. In some sense a nonequality predicate has the same effect as a compound OR predicate. However in this case we know that the two regions have been carved from a single larger domain by the removal of a single hyperplane. So we can treat a domain defined with nonequality constraints either as the union of multiple subregions or as a single region with slices missing from it depending on which view is more useful.

The effect of equality constraints on an input space domain is very important and complex. If we start with an \( n \)-dimensional input space, an equality predicate constrains the domain being defined to lie in a particular hyperplane of that space. A hyperplane of an \( n \)-dimensional space is
Figure 6: Exterior and Interior Borders of A Domain
also an \((N-1)\)-dimensional space itself. For example a hyperplane in a three-dimensional space is a plane, and a plane itself is a complete two-dimensional space. So each equality constraint can reduce the dimensionality of the final input space domain by one. Since higher dimensional domains are more complex, equality constraints tend to simplify the structure of a particular input space domain.

So in summary an input space domain is defined by a set of predicates containing three very different types of predicates. The equality constraints limit the domain to some lower dimensional subspace; the inequality constraints define the exterior boundary of the domain in this lower dimensional space; and the nonequality constraints cut slices out of the domain defined by the inequalities.

An input space domain for a path which contains a compound predicate can be characterized essentially by extending the previous description. A compound predicate of the form \(C(i) \text{ AND } C(i+1)\) has no particular novel effect on the path condition or on the domain except that this single predicate can produce multiple border segments, one for each simple predicate in it. However a compound predicate of the form \(C(i) \text{ OR } C(i+1)\) defines the union of two regions as the input space domain for the path. These multiple regions can be disconnected, partially overlapping, or one can be completely contained within the other. The regions are not related other than the fact that they can have many predicates in common, and they must be treated as truly distinct domains. An additional complication arises when one of these domains does not materialize because of infeasibility. If one of the two domains which could be formed is infeasible, the compound OR predicate will produce just a single domain where we would expect two domains to be defined. We should also note that a compound predicate interpretation with an OR can be produced by a compound predicate expressed with an AND operator. On certain paths, for example those executing the ELSE branch of an IF construct, the predicate condition in the program must be false in order for the path to be executed. Since \(\text{NOT } C(i) \text{ AND } C(i+1)\) is equivalent to \(\text{NOT } C(i) \text{ OR } \text{NOT } C(i+1)\), the latter interpretation can be produced by a compound predicate with an AND operator. Compound predicates introduce no new concepts, but they certainly can complicate an input space structure since many separate domains can correspond to a single path.

A predicate which refers to an element of an input array can cause major complications. An \textit{indeterminate array reference} is the occurrence in a particular predicate interpretation of an indexed variable whose index depends on
an input variable. For example, if there are three input variables \( J \), \( X(1) \), and \( X(2) \), and \( J \) can only have the value 1 or 2, a predicate interpretation such as \( X(J) < 3 \) will actually produce the constraint \( X(1) < 3 \) over part of the input space and \( X(2) < 3 \) over another part of the space. This is a major complication since a single predicate can have two interpretations on the same path. Although this is a practical problem which must be solved in the implementation of any test data generation system, an analysis of it would not offer any valuable conceptual insights. A heuristic technique in which values are selected for the index variables first thus binding the array references is a promising approach and has been investigated by Ramamoorthy [RAMAC76]. Since this problem is not of great theoretical importance, we will always assume that a predicate does not contain any indeterminate array references.

4.4 **Predicates Which Do Not Produce Border Segments**

Not every predicate affects or constrains the input space domain, so the number of predicates in the path condition only provides an upper bound on the number of border segments forming the boundary. In general the number of border segments in the exterior boundary of the domain can be less than the number of inequality constraints on the path. Also, each nonequality predicate will not necessarily cause a hyperplane to be removed from the domain. Finally, only certain of the equality constraints will actually reduce the domain to a lower dimensional space. This situation can be caused in two different ways which are discussed below.

An **input-independent predicate interpretation** is completely independent of any input variables and reduces to a simple relation between constants. When it is appended to a path condition two things can happen. If the arithmetic relation is true, the predicate is a tautology, is satisfied by all input values, and does not further constrain the input space domain. If the relation is false, the predicate can never be satisfied, and the path is infeasible. Since a single predicate can be input independent on some paths but not on others, input independence is a property of a particular interpretation and not of the predicate itself.
The following program demonstrates various types of input independence.

```
READ A, B;
IF A > 0
  THEN C = A + B;
ELSE C = 1;
ENDIF;
IF C > 0
  THEN D = 2*C;
ELSE D = C + 3;
ENDIF;
DO I = 1, 5, 1; (P3)
  D = D + 2;
ENDDO;
E = D + C;
WRITE E;
```

The second predicate has two interpretations. The interpretation \( A + B > 0 \) will be produced for paths through the THEN branch of the first IF construct, but the input independent interpretation, \( I > 0 \), will be produced on the other paths where ELSE \( C = 1 \) has been executed. In this case the path through the ELSE branch of the second IF is infeasible since the predicate to be satisfied is \( C \leq 0 \) which is interpreted as \( I \leq 0 \), and is obviously unsatisfiable. A more common type of input independent predicate is produced by an iterative DO-loop. A DO statement produces a predicate each time it is executed, and in this case the input independent predicates

\[
1 \leq 5 \quad 2 \leq 5 \quad 3 \leq 5 \quad 4 \leq 5 \quad 5 \leq 5 \quad 6 > 5
\]

will be produced in executing the DO-loop. We expect this to be the case since a DO-loop of this form must be executed five times regardless of the specific input values being used. Every path on which the loop is executed other than five times is infeasible since a predicate such as \( 4 > 5 \) or \( 6 \leq 5 \) would be generated in the path condition. The input independence of these loop-generated predicates is very important since there can be a large number of these predicates, but none of them affect the boundary of the input space domain.
It is also possible for predicates which do constrain input variables not to affect the boundary of the input space domain. If a particular predicate is superceded by a more restrictive predicate, it is said to be a **redundant predicate**. We call this phenomenon **predicate dominance** since the more restrictive constraint dominates the redundant predicate. In the case of **simple dominance** the redundant predicate is completely dominated by another predicate, e.g. the predicate \( I \leq J + 5 \) is dominated by the predicate \( I \leq J + 2 \) since it is implied by this more restrictive condition. In the case of **complex dominance** a combination of predicates is required to completely dominate the redundant predicate, and none of these predicates is completely dominant by itself. In either case the redundant predicate will not form any part of the border and is superfluous since it does not further constrain the domain.

Examples of both simple and complex dominance are provided in figure 7. The upper diagram shows how the addition of a single more restrictive predicate causes a predicate to become redundant, and in the lower diagram the dashed extensions of the dominant borders show that neither predicate by itself dominates the redundant predicate.

Since a nonequality predicate is the least restrictive type, it can very easily be dominated. If the domain defined by the other predicates lies completely in one of the half-spaces defined by the nonequality predicate, it will be unaffected by the predicate. An equality predicate is the most restrictive type and can frequently dominate many other predicates. For example, the constraint \( I = 2*J + 1 \) dominates an inequality such as \( I < 2*J + 3 \) since the hyperplane defined by the equality lies completely in the half-space satisfying the inequality. This equality also dominates a nonequality constraint such as \( I \neq 2*J \) by similar reasoning. An equality constraint can also be redundant since it can be implied by the combination of other equalities. Consider the following three equalities:

\[
\begin{align*}
(1) \quad I + J &= 2 \\
(2) \quad 2*J - 3*J &= -1 \\
(3) \quad 4*J - J &= 3
\end{align*}
\]

Since the third equality is equivalent to twice the first plus the second, only two of the three are independent constraints, and the third is redundant. A redundant equality does not further constrain the input data and does not act to reduce the dimensionality of the input space domain. Unfortunately, even though redundant predicates are
quite common and simplify the structure of the input space domain, it is difficult to recognize them a priori, particularly in the case of complex dominance.

4.5 Infeasible Paths

An infeasible path is a control path whose path condition is always unsatisfiable. This means that there are no points in the input space which simultaneously meet each constraint in the path condition. This is a fairly common occurrence and in no way should be equated with a programming error or even poor programming style. Certain control paths exist in the program which were never meant to be executed. Unfortunately, recognizing an infeasible path a priori is a very difficult problem, since we have to decide whether the path condition is satisfiable or not.

The problem of determining whether a path is feasible is known to be NP-complete in the general case. This means that no deterministic algorithm is known which executes in a polynomial amount of time. A path is feasible only if input values exist which cause the path condition to be true. In general, the path condition is a Boolean expression in conjunctive normal form (product of sums), e.g., \( C(1) \text{ AND } C(2) \text{ AND } [C(3) \text{ OR } C(4)] \text{ AND } C(5) \). In addition, there is no limit on the number of simple predicates which may be used in any compound predicate, e.g., \( [C(3) \text{ OR } C(4)] \). If we replace each simple predicate by a Boolean variable, such that the variable has the value 1 only if the predicate is true, the path condition is equivalent to a general Boolean expression in conjunctive normal form (CNF). Therefore, the infeasible path problem is equivalent to the satisfiability problem for Boolean expressions in CNF. Aho, Hopcroft, and Ullman prove that this satisfiability problem is NP-complete if as many as three Boolean variables are allowed in each disjunctive term. (*) So, the path feasibility problem is theoretically NP-complete because of the possibility of compound predicates using the Boolean operator OR. Even though the feasibility of a path containing only simple predicates can be determined using linear programming algorithms, a significant amount of computation may be wasted if many of the paths to be tested are infeasible.

(*) AHOA74, Theorem 10.3 and Theorem 10.4, p. 377–384.
If we picture the formation of the input space domain for an infeasible path by intersecting the area defined by each predicate in turn with the domain generated by the previous predicates, at some point the infeasibility will be observed as a null intersection. An equality predicate will cause the infeasibility if the hyperplane it defines does not intersect the domain defined by the predicates previously considered. Infeasibility occurs for an inequality if the entire domain currently formed lies in the half-space not satisfying the new inequality. A nonequality predicate creates an infeasibility if the domain currently formed has been reduced (by equalities) enough so that it lies completely within the hyperplane not satisfying the nonequality constraint. Of course the order in which we examine the predicates affects the particular way in which the infeasibility is first observed.

One last point is the relationship between redundant predicates and infeasible paths. If a predicate is redundant for some path, it causes a related path to be infeasible. This related path is the one whose path condition is the same except for the redundant predicate which appears in its negated form. For example, if \( C(3) \) is implied by \( (C(1) \text{ AND } C(2)) \), then \( \text{NOT } C(3) \) must contradict \( (C(1) \text{ AND } C(2)) \). This relationship is diagrammed in figures 8 and 9.

4.6 An Example Program and Input Space Partitioning Diagram

In this section the various aspects of input space structure which have been discussed previously will be presented in the context of a complete program and a diagram of its input space structure. Some explanation of the conventions and notation used in this example is needed. In the sample program the code structure is a sequence of IF constructs. Since we have five IF constructs, each path is represented by a string of five letters, each either T or E, where the first letter is associated with the first IF construct and so on. For example ETEET denotes a path following the ELSE branch in the first, third, and fourth IF constructs and the THEN branch in the second and fifth IF constructs. The shorthand notation \( T-T-T-T-T \) represents the set of all paths which execute the THEN branch of the first IF construct. Each path is also assigned a number to relate it to the appropriate input space domain in the diagrams. In order to clarify the relationship between predicates and the order segments they produce, each predicate has been
Figure 8: Predicate C3 Dominated by (C1 AND C2)

Figure 9: Infeasibility Caused by \[\text{NOT } C(j)\]
assigned a boundary form such as solid line, dashed line, etc., and these border forms are listed with the program in figure 10. In addition, the new border segments added in each snapshot of the input space structure appear darker than the other border segments. The small arrows attached to the border segments in figures 11-15 point to the domain for which the border is a closed border. Of course, this example uses two input variables to allow a two-dimensional view of the partitioning of the input space, but even in this simple example we see that the partitioning of the input space can become quite complicated. The concepts and phenomena displayed in this two-dimensional example will extend to general spaces of higher dimensionality. The example program being studied is listed in figure 10, the development of the input space structure is traced in figures 11 through 15, and all control paths and predicate interpretations are summarized in table I.

The first predicate in every program must have a single interpretation since there is only one sequence of assignment statements possible from the beginning of the program up until this first decision or branch point. In this example the first predicate, \( C > 6 \), will be interpreted as \( I + 2*J > 7 \) since \( C = 1 + 2*J - I \). This single interpretation is seen in figure 11 as a single continuous border segment spanning the input space. The predicate \( I + 2*J > 7 \) will be part of the path condition for every path in the set \( T \), and the predicate \( I + 2*J \leq 7 \) will be in the path condition for paths of the form \( E \). Even though these predicates have different operators, they are considered to be a single interpretation.

The second predicate has two interpretations depending on the value assigned to \( D \). On paths of the form \( T \), the variable \( D \) is assigned the value \( C - I \), and \( D \geq C + 2 \) is interpreted as \( I \leq -2 \). On \( E \) paths \( D = C + I \), and the interpretation is \( I \geq 2 \). Again, for paths taking the ELSE branch in the second IF construct, the relational operator is the negation of the operator which actually appears in the program (see table I). These two interpretations produce two discontinuous and non-overlapping border segments as diagrammed in figure 12, and each border segment spans one of the subspaces bounded by the border generated from the first predicate.

The third predicate, \( F = 0 \), has only the one interpretation \( 2*I + J = 2 \), since the assignment statements which define the variables used in the predicate both directly, \( F = 2*C - 3*J \), and indirectly, \( C = I + 2*J - 1 \), are common to all paths. Therefore the predicate produces the single border spanning the input space as shown by the heavy
Figure 10. Sample Program with Five Predicates

```
IF \( z + 2 \times z + 2 \times f = 0 \) + 2
THEN H
ELSE H
IF u = \( f + 2 \times f + 1 \)
THEN u = f - 2 + f
ELSE u = f
IF f = \( z \times c - f \times f \)
ELSE f = f
IF u = c + 2
THEN u = c + 1
ELSE u = c - 1
IF c < 0
THEN I = I
```

\[ \text{PRINT H} \]
<table>
<thead>
<tr>
<th>#</th>
<th>PATH</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>DOM.</th>
<th>PREDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ITITIT</td>
<td>1+2J&gt;7</td>
<td>J&lt;2</td>
<td>21+J=2</td>
<td>J&lt;3</td>
<td>1+2J&gt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>2</td>
<td>ITITIE</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>1+2J&lt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>3</td>
<td>ITITET</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>1+3J&gt;0</td>
<td>1,4,5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ITITEE</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>1+3J&lt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>5</td>
<td>ITITET</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+J&lt;2</td>
<td>J&lt;3</td>
<td>1+2J&gt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>6</td>
<td>ITIITE</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>1+2J&lt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>7</td>
<td>ITITET</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>1+3J&gt;0</td>
<td>4,5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ITITET</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>1+3J&lt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>9</td>
<td>IITTTT</td>
<td>1&gt;2</td>
<td>21+J=2</td>
<td>J&lt;3</td>
<td>-21+2J&gt;9</td>
<td></td>
<td>infeas.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>IITTTT</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>-21+2J&lt;9</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>11</td>
<td>IITTTT</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>-21+3J&gt;9</td>
<td>4,5</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>IITTTT</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>-21+3J&lt;9</td>
<td></td>
<td>2,3</td>
</tr>
<tr>
<td>13</td>
<td>ITIITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+J&lt;2</td>
<td>J&lt;3</td>
<td>-21+2J&gt;9</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>14</td>
<td>ITIITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>-21+2J&lt;9</td>
<td></td>
<td>2,3,5</td>
</tr>
<tr>
<td>15</td>
<td>ITIITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>-21+3J&gt;9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>ITIITT</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>-21+3J&lt;9</td>
<td></td>
<td>2,3</td>
</tr>
<tr>
<td>17</td>
<td>IITITI</td>
<td>1+2J&lt;7</td>
<td>J&lt;2</td>
<td>21+J=2</td>
<td>J&lt;3</td>
<td>51+2J&gt;0</td>
<td>1,2,5</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>51+2J&lt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>19</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>51+3J&gt;0</td>
<td>1,5</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>51+3J&lt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>21</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+J&lt;2</td>
<td>J&lt;3</td>
<td>51+2J&gt;0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>51+2J&lt;0</td>
<td></td>
<td>1,3,4</td>
</tr>
<tr>
<td>23</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>51+3J&gt;0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>51+3J&lt;0</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>25</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>1&lt;2</td>
<td>21+J=2</td>
<td>J&lt;3</td>
<td>21+2J&gt;9</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+2J&lt;9</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>27</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>21+3J&gt;9</td>
<td>4,5</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+3J&lt;9</td>
<td></td>
<td>infeas.</td>
</tr>
<tr>
<td>29</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+J&lt;2</td>
<td>J&lt;3</td>
<td>21+2J&gt;9</td>
<td>1,3</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+2J&lt;9</td>
<td></td>
<td>1,3</td>
</tr>
<tr>
<td>31</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>J&lt;3</td>
<td>21+3J&gt;9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>IITITI</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>21+3J&lt;9</td>
<td></td>
<td>2,3</td>
</tr>
</tbody>
</table>
Figure 11: Input Space Structure - Snapshot 1
Figure 12: Input Space Structure - Snapshot 2
solid line in figure 13. This predicate shows the effects of both equality and nonequality predicates. Input data following each feasible path through the THEN branch in this IF construct must satisfy the predicate $2*I + J = 2$, and the corresponding input space domains are reduced from two-dimensional areas to segments of the line $2*I + J = 2$. This reduction in dimensionality is expected as a result of an independent equality constraint. These domains can still be characterized as one-dimensional convex polytopes, and the borders of each of these domains, which are endpoints of the various line segments, are one-dimensional hyperplanes. So even though these domains look very different, they still fit the general characterization developed previously. Data following a feasible path through the ELSE branch must satisfy the predicate $2*I + J \neq 2$. Therefore each corresponding input space domain consists of two regions separated by the hyperplane $2*I + J = 2$, and this can best be seen in figure 15, the fully developed input space structure.

The fourth predicate, $G < -3$, has two interpretations since $G$ can be assigned two different values. However these two interpretations are degenerate since $F - 2*I + 2$ is equivalent to $J$ when $F$ is replaced by its symbolic value, $2*I + J - 2$. This degeneracy causes the two interpretations to appear as a single border in figure 14, and the example was designed in this way to demonstrate the possibility of degenerate interpretations. However, this single border represents two different interpretations and must be treated as such by the testing methodology.

The fifth predicate, $2*D + 3*E \geq F - G + J$, has 16 interpretations, but in figure 15 we see only four border segments corresponding to this predicate. In table 1 we see that the 12 interpretations which do not lead to border segments occur on infeasible paths or are dominated by other predicates on feasible paths. We also see the effect of the previously discussed degeneracy in that the 16 interpretations produce only 8 different predicate forms. For example, the interpretation for paths 1 & 2 is the same as the interpretation for paths 5 & 0.

The problem of path infeasibility can be seen in this example since only one half of the control paths are actually executable paths associated with input space domains. Also, predicate dominance is apparent in that many of the input space domains in figure 15 have fewer than five border segments, other than min-max borders, in their boundaries.
Figure 13: Input Space Structure - Snapshot 3
Figure 14: Input Space Structure - Snapshot 4
Figure 15: Fully Developed Input Space Structure
So, in summary, every input space is finite, bounded by a set of min-max constraints, and it can be partitioned into a set of domains. One or more of these domains corresponds to each executable path in the program and contains all input data points which will cause the path to be executed. Assuming linear predicate interpretations, each domain geometrically forms a convex polytope and is bounded by hyperplane segments defined by various predicate interpretations on the corresponding path. The input space structure represents the conditions determining the control flow in the program, and its geometry is used extensively in developing the domain testing strategy.
CHAPTER 5
ERROR CLASSES AND RELATED THEORETICAL LIMITATIONS

In this chapter we formulate an error classification scheme and analyze various types of errors. Section 1 provides an overview and briefly describes the three error classes. The important class of domain errors is defined in section 2, and the characteristics of these errors are described for various types of predicates. Section 3 analyzes transformation errors, and the theoretical ramifications of coincidental correctness are discussed. The third and final class, missing path errors, is described in section 4, and a detailed discussion of these errors is presented, analyzing a missing path error as a missing predicate.

5.1 Overview of the Error Classification Scheme

In order to develop a program testing strategy and to analyze its effectiveness, we need a thorough understanding of the errors which we are attempting to detect. The error classification scheme must capture essential differences in the testing problems presented by various errors. Each class must be precisely defined, and the classification scheme must be exhaustive over all possible errors. Of course, as previously noted, we assume both that each program has been compiled successfully and that all I/O errors, infinite loops, and errors causing premature program termination have been eliminated prior to testing. (*) The appropriateness of our error classification scheme depends on how helpful it is in developing and describing an effective program testing strategy. Each error class should be defined because of the similarity of the problems it presents for testing, and the differences between error

(*) A strategy for eliminating the majority of these errors, based on program verification, has been developed by Sites [SITER74].
classes should be important with respect to testing. In this work the success of any error classification scheme can only be judged by how useful it is in analyzing the testing process.

First we must decide what to use as the basis of the error classification scheme. An error exists when there is a discrepancy between what the program computes for some set of input values and what we expect it to compute. In general each error will affect many points in the input space, and we will find the error if we choose one or more of the affected points as test points. Most error classification schemes which have been developed are based on the cause of the error, e.g., incorrect program specifications, misplaced parentheses, misspelled variable name, etc., and in studies of programming style and error sources this type of scheme works quite well. However, in program testing the important characteristic of an error is its effect on the input-output relationship, and a natural error classification scheme is one which is based on the observable effect of the error regardless of the specific source. The input space points which are affected by an error in the path being tested form a pattern within the associated input space domain. Each error class will produce a characteristic pattern of incorrect data points, and the predictability of these patterns will be used in designing an effective testing strategy.

The natural unit of testing is the program path or equivalently the input space domain for a particular path. As developed in section 3.6, the computation performed along any path consists of two parts, the transformation which defines the functions computed for the output variables and the path condition which defines the domain of the input space for which the path is executed. An error on the path can affect either or both of these components, and our error classification scheme is based on which component is incorrect and the characteristics of the set of input points affected by the error. Many error classifications have appeared in the literature, and the scheme used in this research is similar to one described by Howden (HOWDEN76). However, there is a discrepancy in the terminology, since Howden defines three classes, viz., domain errors, computation errors, and subcase errors, which correspond to domain errors, transformation errors, and missing path errors respectively, as used in this research. The following paragraphs provide an informal description of the classification scheme. A precise definition of each error class is presented in the section of this chapter analyzing each specific class.
A path contains a **domain error** if an error in some predicate interpretation causes a border segment to be "shifted" from its correct position or to have the wrong form, i.e., relational operator. A domain error can be caused by an incorrectly specified predicate or by an incorrect assignment statement which affects a variable used in the predicate. If a predicate is located within a DO-loop, it may cause more than one border to be in error since it can produce many predicate interpretations for a particular path. In addition a single assignment statement error can affect many predicate interpretations because the variable whose value is assigned incorrectly can be used in many different predicates, both directly and indirectly. Finally, if the domain error is caused by an incorrect assignment statement which also affects the function computed for some output variable, the transformation might also be incorrect. So, in summary, an error is a domain error if it causes some border of the domain to be incorrect, regardless of whether the functions computed for the output variables are correct or not.

A path contains a **transformation error** when the path condition defining the domain is correct, but the function calculated for some output variable does not compute the correct output value for all points in the input domain. Since the domain is defined correctly, this type of error must be caused by one or more incorrect assignment statements, each of which affects only output variables and is not used either directly or indirectly in forming any predicate interpretation for the path.

A path contains a **missing path error** when a border is missing which would divide the domain into two domains, one associated with the test path and a second domain corresponding to the missing path. This type of error occurs when some special condition requiring different processing is omitted. This special condition takes the form of an extra border which would divide the domain into two domains, thus creating the missing execution path. Therefore, a missing path error can be viewed as a problem caused by a missing predicate which would subdivide the domain and create a new execution path. This type of error differs from a domain error since the existing borders defining the domain are correct. It differs from a transformation error since no function exists which would be correct for the entire domain, and therefore the domain must be divided into two domains.

Since a single path may contain more than one error, we must be careful that our analysis of one error is valid even when that error is accompanied by other errors. We find an
error by selecting test data for which the program computes one or more incorrect output values. Therefore, an effective testing strategy for a particular class of errors must guarantee that at least one of the selected test points will lie in the area of the domain which would be affected by every possible error in the class, and this must be true regardless of the number of errors affecting the domain. In general the existence of a second error on a path will cause a second area of the domain to be incorrect. Therefore, except in the unlikely event that two errors exactly cancel each other, multiple errors will not be a problem. In fact, it will be easier to identify an incorrect path if that path contains more than one error, since the program will compute incorrect results for a larger portion of the input domain.

In summary, we classify an error by the characteristic effect it produces rather than the specific programming mistake which causes it. The error classes, viz., domain errors, transformation errors, and missing path errors, have been defined according to which component of the model of computation is incorrect. The remainder of this chapter analyzes each of these errors in detail and identifies important theoretical limitations in testing for errors of each class.

5.2 Domain Errors

Traditionally, programming mistakes which cause the wrong branch to be followed at some decision point in the program have been loosely referred to as "control flow errors." However, in this work we need a precise definition of this type of error rather than an intuitive description. As defined earlier, the domain boundary consists of border segments, each of which corresponds to a simple predicate in the path condition. A domain error is defined as a discrepancy in any of these borders which causes the program to compute one or more incorrect output values for at least one input data point. The incorrect border causes these points to be in a domain corresponding to a different path, and therefore the wrong transformation is computed. An incorrect assignment statement affecting a predicate interpretation may also affect the transformation computed on the path, but an incorrect border segment is defined as a domain error regardless of whether the transformation is correct or not. Finally, we should note that not all predicate errors are classified as domain errors. In section 4.4 we noted that redundant and input-independent
predicate interpretations do not produce border segments. Since a domain error can only occur in a predicate which produces a border segment, incorrect redundant and input-independent predicates are not defined as domain errors.

A linear border segment is part of the hyperplane defined by the corresponding linear predicate interpretation, after replacing the relational operator by an equality. There are two ways in which a predicate interpretation may be incorrect. A border shift is defined as a domain error in which some coefficient of the predicate interpretation is incorrect, where the set of coefficients includes the constant term. When a border is shifted, the predicate essentially defines the wrong hyperplane. An error in the "form" of the border occurs when the wrong relational operator is used in the predicate. A common example of the latter case is when we use the operator ≤, when we really should use the strict inequality <. In this case the border segment is closed when it should be open, but the hyperplane defining the border is correct. Of course, a single domain error might consist of both a border shift and an incorrect relational operator. Each of these possibilities must be taken into account in developing an effective testing strategy.

A border shift of large magnitude is easy to detect since many input points are affected by the error, and a border shift of small magnitude is difficult to detect since relatively few points are affected. In referring to the magnitude of a border shift we are implicitly comparing the given border with the border that would exist in a correct program. However, there are an infinite number of solutions, i.e., correct programs, for any particular problem, and the apparent effect of a domain error depends on the specific correct program to which we compare the given program. A border which is incorrect may appear to be a border shift of relatively small magnitude when compared to one possible correct program but a fairly major shift in comparison to some other correct program. Since a single correct program for a problem cannot be identified, the effect of a domain error cannot be uniquely determined. Therefore, in developing a testing strategy no assumptions can be made about the magnitude of a border shift, and all possible border shifts must be considered.

Our testing methodology does not depend on the existence of a unique correct program. The domain testing strategy defines a set of test points for each border. If the program produces correct output values for each of these test cases, we are able to conclude that the given border is
the true border for some correct program. On the other hand, if the program produces incorrect output values for any of these test cases, the program must contain an error, and the test data has successfully detected it. The magnitude of the error is immaterial, since the test data does not provide any information other than the fact that an error exists. Furthermore, when a test case is processed incorrectly, there is no way even to determine the class of the detected error, using only the test results. In order to describe our results in a clear and concise form, at times we will discuss the difference between a given border and a correct border. However, we do not assume that there is a unique correct border or even that a border corresponding to the given border can be identified in any correct program. In summary, any reference to a correct border or a correct program is made only for purposes of exposition, and these concepts are not central to the development of the domain testing results.

Points on or near a border are most useful in detecting border shifts. A particular point in the input space is affected by a domain error when the error causes the point to lie in the wrong domain, or equivalently to follow the wrong path. This means that only points in the area lying between the true, i.e., correct, position of the border and the given position of the border will be in error, and we must select a test point from this area to catch the error.

There are three ways in which a border shift may affect a particular domain such as D1 in the diagrams in figure 16. In these diagrams the solid line is the given border, the dashed line is the correct border, and the shaded area is the region affected. In the upper left hand diagram the border has shifted to reduce the size of domain D1 while enlarging domain D2. The points in the shaded area will be in error, since for these points the output values will be computed by the transformation for domain D2, while in fact the transformation for domain D1 should be used. The upper right hand diagram is similar except in this case the shift has enlarged domain D1 and reduced D2. Again the shaded region will be in error since these points are in domain D1 but should be in domain D2. In the lower diagram we see that the given border has shifted so that it now intersects the correct border. In this case, while some points in D1 should be in D2, other points in D2 should also be in D1, and these regions constitute the two shaded areas in the diagram. Our testing strategy must be equally effective for each type of border shift, and we should be able to make use of the fact that regardless of the type or magnitude of the shift, the points on and near the given border are most likely to be affected. In certain cases, e.g., the top
Figure 10: The Three Types of Border Shifts
diagram, only points outside of the domain being tested are affected, and therefore we expect it to be necessary to select some test points which lie outside of the domain being tested.

There are three types of predicates: equalities (=), inequalities (\(<, \leq, >, \geq\)), and nonequalities (\(\neq\)), and we must consider domain errors for each of these types. As described in chapter 4, equalities constrain the entire domain to lie within some lower dimensional space. The set of inequality constraints defines the exterior boundary of the domain within the subspace defined by the equality constraints. Finally, the nonequality constraints define distinct hyperplanes which are not part of the domain defined by the inequalities. We must analyze all possible shifts and relational operator errors for each of these three types of predicates.

Since an equality predicate constrains the domain to a lower dimensional space, a border shift of an equality predicate means that the domain has been constrained to the wrong subspace. If the equality predicate defines a \(K\)-dimensional hyperplane, a linearly independent set of \(K\) points completely determines the hyperplane, and no other \(K\)-dimensional hyperplane can contain this set of points. A set of points is linearly independent if no point in the set can be expressed as a linear combination of the other \(K\) points. Since each point can also be viewed as a vector, the linear independence of a set of points is equivalent to the standard concept of a linearly independent set of vectors. (*) Two distinct \(K\)-dimensional hyperplanes can have any number of points in common as long as not more than \(K-1\) of them are linearly independent. Therefore, the given domain and the correct domain can overlap in an indeterminable number of points as long as not more than \(K-1\) of them are linearly independent.

If a domain error in an equality predicate is caused by an incorrect relational operator rather than a border shift, the domain being tested should actually be of a higher dimensionality. For example, in the upper diagram in figure 17, the correct domain would be \(D1\) or \(D2\) if the predicate producing border \(b\) is supposed to be a strict inequality. Similarly, the domain would be \(D1 \cup B\) (**) for one of the weak inequalities and \(D2 \cup B\) for the other. The correct

(*) HADLGo2, p. 40.

(**) In these discussions the notation \(D1 \cup D2\) should be read as the union of \(D1\) and \(D2\).
Figure 17: Domain Errors For An Equality Predicate
domain would be $D_1 \cup D_2$ if the equality in fact should be a nonequality constraint. In addition to having the wrong form, i.e., an incorrect operator, the border can simultaneously be shifted. For example a predicate interpretation such as $I + J = 3$ might be used instead of $I + 2*J < -1$. The analysis for this type of compound domain error is similar to the various cases described above for simple relational operator errors except that the domains involved in the previous analysis, $D_1$ and $D_2$, must now be defined by the shifted border $B^*$, as in the lower diagram in figure 17, instead of the given border $B$. Obviously, the addition of a border shift should make these errors even easier to detect.

One last case must be considered, since it is possible that the correct equality constraint is actually either redundant or input-independent. In the upper diagram in figure 17 this would correspond to a correct domain which is $D_1 \cup B \cup D_2$, and essentially this means that the domain has been constrained to lie in a lower dimensional space than it should. This type of equality error is the most extensive, since all points in both domains, $D_1$ and $D_2$, execute the wrong path and will detect the error.

Domain errors for inequality predicates can also be caused by a simple border shift, an incorrect relational operator, or a combination of the two. We can picture a border shift as the border actually moving from its correct position to its given position as calculated in the program. The effect of a border shift is clearly that each point over which the border passes will be in the wrong domain. Therefore, some points in the domain being tested should actually be in adjacent domains, and other points in adjacent domains belong in the tested domain.

A severe type of relational operator error occurs when the sense of the inequality is reversed, e.g., $<$ instead of $>$, since in this case the entire domain is incorrect. In figure 18 domains $D$ and $D'$ would be switched if the sense of border $B$ were reversed. (*) A less severe and more common operator error occurs when a closed border is specified instead of an open border or vice-versa, e.g., $<$ instead of $\leq$. The only points affected by this type of error are the points on the border itself, and obviously we must test a point on the border to detect this type of error. If the inequality should in fact be an equality, the domain for the

(*) The domain being tested is $D$, which is equivalent to $D_1 \cup D_2$, and the adjacent domain is $D'$, which is equivalent to $D_3 \cup D_4$. 
Figure 18: Domain Errors For An Inequality Predicate
Each nonequality predicate can also contain any one of the previously described types of domain errors. A nonequality predicate defines a hyperplane, interior to the domain, which is not part of the domain. Therefore, if a nonequality border is shifted, the associated hyperplane should actually be part of the domain. In addition, some other hyperplane in the domain being tested does not belong in this domain. Since the nonequality constraint is the complement of an equality constraint, every possible type of relational operator error, with or without a border shift, is analogous to one of the previously described cases for an equality predicate. For example, if the nonequality constraint should actually be the strict inequality $<$, then the complementary equality constraint on a related path should actually use the operator $\leq$. At this point a detailed discussion of each possible type of domain error for nonequality orders would be repetitious, since the characteristics of each type are so closely related to the complementary equality predicate error.

In this detailed and comprehensive analysis of domain errors we have considered every conceivable type of predicate error and also possible combinations of these errors. Such a logically complete treatment is necessary so
that we can be sure that the domain testing strategy developed will be effective for the total class of domain errors. An important observation is that points on and near the borders seem most likely to be affected by any possible domain error. In addition domain errors in which a border is shifted only slightly seem to be the most difficult to detect since a relatively small area of the domain will reflect the error.

The existence of more than one error on the path facilitates error detection in the associated domain. More than one assignment statement on a path may be incorrect, and a particular border may be affected by several of these errors since many variables are used in forming a predicate interpretation. However, regardless of how many incorrect assignment statements affect a predicate, the error is still reflected in the input space as a single border shift. If these assignment statement errors exactly cancel each other in this predicate interpretation, the border would be in the correct position, and for this domain no border shift would exist. Many of the border segments defining a domain may be in error simultaneously. Each border shift will cause more points to be in the wrong domain, and one border shift cannot totally cancel the effects of another border shift. Therefore each additional border shift makes it easier to detect the fact that a domain error exists for the associated path. Essentially, it is easier to detect that a path contains errors when many points in the domain are affected, and conversely the most difficult errors to test for are those which affect a small and unpredictable area of the domain.

5.3 Transformation Errors and Coincidental Correctness

If a program computes $M$ output variables, the transformation for a path consists of $M$ functions, each in the form of a simplified canonical expression over the input variables. The correct transformation for a domain is defined as one which produces correct output values for all points in the domain. An incorrect transformation is defined as one which produces at least one incorrect output value for one or more points in the domain. Therefore, at least one of the $M$ functions of an incorrect transformation differs from the corresponding function of the correct transformation. Using these basic concepts, a transformation error occurs when the domain is defined correctly but an incorrect transformation is calculated on
the path associated with the domain. In this case the domain must be correct, since if it were incorrect, the error would be defined as a domain error rather than a transformation error. Therefore, some transformation must exist which is correct for all points in the domain, but it must be determined whether the transformation which is computed in the program is actually this correct transformation. A transformation error is caused by an incorrect value assignment made to a variable which is used to compute one or more output variables, but since the domain is defined correctly, we know that the incorrect variable is not used, either directly or indirectly, in any predicate on the path.

The transformation error affects the entire domain, and therefore it would seem to be very easy to test for this type of error, since it would be detected by testing any single point in the domain. However, if the transformation is incorrect, it may still produce correct output values at any specific input point since the value computed by two different functions can coincide for particular input values. For example, \( C = A + B \) and \( C = A \times B \) will both assign the value 4 to \( C \) when \( A = 2 \) and \( B = 2 \). Therefore, when the correct output value is calculated for a specific input, we know that either the transformation is correct or the transformation is incorrect and coincides with the correct transformation at the particular input point tested. We define coincidental correctness for a specific input point as the calculation of the correct output values by an incorrect transformation. Incorrect results will be calculated for other points in the domain, and the pattern of coincidentally correct points seems to be unpredictable. Within the class of all computable functions there may exist other functions similar to the desired function which coincide with it over some portion of the domain. If we are so unfortunate as to select only points from this part of the domain for testing, then we will not detect the transformation error. This means that theoretically we have to test every point in the domain to completely preclude the presence of a transformation error. However, in practice, we expect two arbitrary functions to coincide at a set of input points of measure zero, and the possibility of coincidental correctness will not significantly reduce the confidence we gain from testing. In chapter 7 our analysis of coincidental correctness supports this claim, and various techniques for detecting this condition are proposed. Coincidental correctness can also affect the detection of domain errors. If a particular input point is in the wrong domain, it is possible that at this particular point the transformation for that domain coincides with the transformation which would have been computed in the correct
domain. Therefore a domain error might not affect the output values computed for the specific test point selected.

We now formalize these coincidental correctness results as a theorem using the following notation.

IS — the entire input space of a program
TD — a set of test data
x — a single input data point
X — any set of input data points

We also need to define the following three types of correctness.

Output Correctness is defined as the calculation of correct output values without regard to how those values are computed. It is defined for a point or set of points and is abbreviated as OC(x) or OC(X).

Total Correctness is defined as the calculation of correct output values using the correct transformation and is abbreviated as TC(x) or TC(X).

Coincidental Correctness is defined as the calculation of correct output values using an incorrect transformation and is abbreviated as CC(x) or CC(X).

A program can produce correct output values for a particular input data point either by using the correct transformation or by using an incorrect transformation which coincides with the correct one. Therefore the following is true for any input point x.

OC(x) \implies TC(x) \lor CC(x)

The results of each test case are evaluated by determining only whether the output values are correct without regard to how they are computed. Therefore, even if correct results are produced for a set of test data, we can only conclude that OC(TD) is true, rather than the stronger condition TC(TD), since coincidental correctness can occur for any or all of the test cases. However, since a transformation which produces correct output values for all points in the input space is by definition a correct transformation, we can conclude the following.
\( \text{OC(IS)} = \text{TC(IS)} \)

In order to prove a program correct using a testing methodology we must be able to infer its correctness for the entire input space from its true correctness for the selected set of test data. A reliable program testing strategy is therefore defined as one for which the following must be true for any set of test data selected according to the strategy.

\[ \text{TC(TD)} \rightarrow \text{TC(IS)} \]

With this set of definitions we can formally state and prove the following important result.

\[ \text{THEOREM 5.1} \quad \text{Because of coincidental correctness the only reliable program testing strategy is an exhaustive test of every data point in the entire input space.} \]

\[ \text{PROOF:} \quad \text{For a reliable testing strategy } \text{TC(TD)} \rightarrow \text{TC(IS)}, \text{but based on testing results we can only conclude that } \text{OC(TD)} \text{ is true. Therefore, a reliable testing strategy requires that } \text{OC(TD)} = \text{TC(TD)}. \text{ Since CC(x) may be true for any test case, this condition can only be satisfied when TD = IS.} \]

This theorem also means that coincidental correctness is a theoretical limitation to the detection of transformation errors for a single domain. Since an incorrect transformation may be coincidentally correct for every test point in the domain, we cannot preclude the possibility of a transformation error using any reasonable testing strategy. However, a transformation cannot be coincidentally correct for all points in the domain, since by definition it would then be the correct transformation. Therefore, for any single domain the only reliable testing strategy for transformation errors is an exhaustive test of every input data point in the domain.

Regardless of how unlikely it is, coincidental correctness can occur for any input point, and we cannot be completely sure that a program is correct based on testing, unless we exhaustively test the program for every possible combination of input values. Even though we have identified coincidental correctness as a theoretical limitation to the effectiveness of testing, this does not invalidate testing as a useful software reliability technique. We expect
coincidental correctness to be a very rare event, and the resultant loss of confidence will be insignificant in most practical situations.

5.4 Missing Path Errors

A missing path error is defined as an error in which borders are missing which would divide the domain into two or more domains, at least one associated with the existing test path and another with a new path which does not presently exist in the set of execution paths. This is similar to a domain error in that some points in the domain belong in another domain, but the difference in this case is that the other domain does not exist in the input space structure defined by the given program. The domain must be subdivided since no transformation exists which is correct for all points in the domain. On the other hand, if such a transformation were to exist, the domain would not have to be subdivided, and we have defined this as a transformation error rather than a missing path error.

The missing border in the input space domain corresponds to a missing predicate on the path. Adding this extra predicate creates the new path which is executed by points in the extra domain defined by the new border. For example, the assignment statement in the code segment on the left would be replaced by the IF construct on the right.

\[
X = Y/(Z + 1); \\
\text{IF } Z = -1 \\
\text{THEN } X = 0; \\
\text{ELSE } X = Y/(Z + 1); \\
\text{ENDIF};
\]

The border defined by the new predicate must intersect the domain, or the new path would be infeasible and would not correct the error. So, a missing path error can be viewed as a missing border in the input space and as a missing predicate in the program.
The missing border may correspond to a predicate which is actually missing from the program, and in this case the missing path does not appear in the program at all. However, our definition is general enough to include a second possible type of missing predicate. A predicate which exists on the path might contain an error making it redundant or input-independent, when in fact it should actually affect the domain. In this case the missing path is a control path, but it is currently infeasible because of the error in the redundant or input-independent predicate. In either case a border is missing which should divide the domain in two, and the two cases are indistinguishable with respect to testing.

The predicate which must be added to the path determines the part of the domain which corresponds to the missing path, and in figure 19 the missing border is labeled B. This predicate can be simple or compound, and if it is simple, it can be an inequality, an equality, or a nonequality. In order to detect a missing path error we would have to select a test point in that part of the domain which should be executed by the missing path, and therefore the characteristics of this subdomain are very important. If the missing predicate is a nonequality, the subdomain for the missing path is the entire domain minus the new border itself, and in figure 19 a domain consisting of points in both D1 and D2 would be executed by the missing path. This subclass of missing path errors is easy to test for since any point in the domain not on missing border B will detect the error. The subdomain in the inequality case would be the region on one side of the border or the other, and the actual size of this region depends on the location of the new border within the domain. In figure 19 the new domain formed by the missing border is either D1 or D2 if the missing predicate is a strict inequality. In addition, if the missing inequality predicate is not strict, i.e., ≤ or ≥, border B is also part of the subdomain for the missing path. One important property is that there must be at least one extreme point of the domain on either side of the new border, except in the rare case in which the new border is exactly the same as one of the exterior border segments of the domain. This predictable characteristic will be useful in detecting this subclass of missing path errors.

A missing path error of reduced dimensionality is defined as one in which the subdomain for the missing path is a region of lower dimensionality than the entire domain. This occurs when the missing predicate is an equality, since in this case the region for the missing path is just the border itself. In figure 19 only points on the missing border, B, will detect this type of error. Since the border
Figure IV: Subdomains for Simple Missing Predicates
is a region of measure zero with respect to the domain and since there is no indication of where in the domain the error might lie, there is no information on which to base a test data selection strategy for this subclass of missing path errors.

A missing predicate can also be compound using the boolean operators AND and OR. The case in which the missing predicate is of the form \( C_1 \lor C_2 \) can be viewed as the simultaneous occurrence of two missing predicates, each of which is simple. This type of predicate is unique in that the missing path error may produce more than one new border and may subdivide the domain into more than two domains. A typical example is shown in the upper diagram of figure 20, where the shaded regions constitute the subdomain for the missing path. We see that each of the shaded areas is characteristic of the subdomain defined by a simple missing inequality predicate, and this type of error will be no more difficult than the inequality case. The lower diagram of figure 20 shows a subdomain defined by a compound AND predicate. In this case the subdomain can be located anywhere in the domain, and in addition it can be arbitrarily small, even just a single point. More complicated compound predicates will produce subdomains which may consist of many of these unpredictable regions.

Missing path errors are a theoretical limitation to the reliability of any finite program testing methodology, and these results are summarized formally in the following theorem and corollary. We now restate a basic assumption which is important in presenting these results.

**Assumption:** No information other than the program itself is available to help guide the selection of test data.

**THEOREM 5.2** No finite testing strategy can detect all occurrences of missing path errors of reduced dimensionality.

**Proof:** By assumption no information other than the program itself is available. Since no indication of the possible existence of a missing path error is contained within the program, nothing is known which can be used to guide the selection of test points to detect missing path errors. If a missing path error of reduced dimensionality exists for a specific path, it can be detected only in a measure zero subdomain of the input domain for the path. Therefore, the
Figure 20: Subdomains for Compound Missing Predicates
probability of selecting a test point in this subdomain is of measure zero, and the detection of missing path errors of reduced dimensionality is beyond the capability of any finite testing methodology.

Corollary 5.3 The only reliable testing strategy for the general class of missing path errors is an exhaustive test of every data point in the entire input space.

**Proof:** Any compound predicate may be the missing predicate for some missing path error. Since there exist compound predicates which are satisfied by only a single set of input values, each point in the input space constitutes a potential domain for some missing path. Therefore, every point in the input space must be tested to preclude the existence of missing path errors.

This result is stated for the general class of missing path errors since it is not true for the more specific class of missing path errors of reduced dimensionality. For this subclass the domain for the missing path is defined by an equality predicate, and therefore all points on some hyperplane intersecting the domain will be affected. So, a reliable strategy in this case is one which tests at least one point on each hyperplane intersecting the domain. For example, if the domain is constrained only by closed borders, a reliable testing strategy is an exhaustive test of all boundary points, since every hyperplane intersecting the domain must intersect the boundary of the domain. However, this strategy is not reliable for domains constrained by open borders, since points on an open border are not part of the domain and are not affected by a missing path error.

So, in summary, missing path errors of reduced dimensionality and those which correspond to certain compound missing predicates are another theoretical limitation to the reliability of finite program testing. In addition, these results provide another reason why the only completely reliable program testing strategy is an exhaustive test of the entire input space. However, because of the predictability and dimensionality of regions associated with missing inequality and nonequality predicates, an effective strategy can be formulated for these important subclasses of missing path errors.
CHAPTER 6
THE DOMAIN TESTING STRATEGY

In this chapter the main results of this research are presented in detail. After a brief overview in section 1, the domain testing strategy is carefully developed and validated for the simple case of two-dimensional linear inequalities in section 2. The strategy is then generalized to higher dimensional spaces and nonlinear inequalities in sections 3 and 4 respectively. Section 5 analyzes the testing of equality and nonequality predicates, and our methodology is shown to be equally effective for all types of predicates. The domain testing strategy is designed to detect domain errors, and in the final section of this chapter its effectiveness for transformation and missing path errors is evaluated.

6.1 Overview

The domain testing strategy is designed to detect domain errors and will be effective in detecting errors in any type of domain border under certain conditions. It characterizes a set of test points for each border segment which, if processed correctly, determine that both the form and the position of the border are correct. An error in the form of the border occurs when the wrong relational operator is used in the corresponding predicate, and an error in the position of the border occurs when one or more incorrect coefficients are computed for the particular predicate interpretation. The strategy is based on a geometrical analysis of the domain boundary and takes advantage of the fact that points on and near the border are most sensitive to domain errors. The primary goal of our methodology is to provide an effective technique for testing linear inequality predicates. However, the generality of the geometric arguments used in developing the strategy allow it to be extended to both equalities and nonequalities and to classes of nonlinear borders.
A domain error is detected by testing an input data point for which the program produces one or more incorrect output values. The error causes the point to be in the wrong domain or equivalently to follow the wrong path. We therefore assume that adjacent domains are associated with paths on which different functions are computed. This insures that if a test point is in the wrong domain, a different transformation will be calculated, and the domain error will be detected. Of course, this assumption will not be valid for some programs since different paths can compute the same function. However, we make this assumption in order to clarify the presentation of the methodology, and in chapter 7 we show how the domain testing strategy can be modified to overcome this restriction.

Some basic definitions are needed for the presentation of the domain testing results. The given border is defined as the border which is actually calculated by the program being tested, and the true border is defined as the border which would be calculated in some correct program. As discussed in section 5.2, there are an infinite number of correct programs for any problem. Therefore, a unique true border corresponding to a given border cannot be identified. Furthermore, it may be impossible even to identify a true border corresponding to the given border, if the correct program is substantially different from the given program. However, this problem does not affect the domain testing results. Our methodology tests the given border, trying to determine if it is the true border for some correct program. If the test points selected for a border are processed correctly, we can conclude that the given border is the true border for some correct program. On the other hand, if any of the test points produce incorrect results, we know that an error exists, and we can conclude only that the given border is incorrect. In this event the actual difference between the given border and the true border is immaterial. In developing the domain testing strategy we make no assumptions about the true border, and our results will be valid even if there exists no true border corresponding to the given border.

Each border segment is part of a hyperplane defined by the corresponding predicate, and the extent of the segment is determined by the way the hyperplane intersects the other borders. If the border is defined in a K-dimensional space, it can contain at most K linearly independent points, as described in section 5.2. Therefore, if we can identify, via testing, K linearly independent points which must lie on a true border, we can uniquely determine that border. So we want to select a set of test cases which can be used to identify points on a true border, and if these points also
lie on the given border, we conclude that the given border is correct.

The test cases selected will be of two types, defined by their position with respect to the given border. An **ON test point** is defined as a test point which lies on the given border. An **OFF test point** is defined as a test point which is a small distance $\varepsilon$ from the given border and which lies on the open side of the border. Therefore, we observe that when testing a closed border, the ON test points are in the domain being tested, and each OFF test point is in some adjacent domain. Conversely, when testing an open border, each ON test point is in some adjacent domain, while the OFF test points are in the domain being tested.

A domain defined by simple linear predicates is a convex polytope, and each point can be classified according to its position within the domain. An **interior point** is defined as one which is surrounded by an $\varepsilon$-neighborhood containing only points in the domain. Similarly, a **boundary point** is one for which every $\varepsilon$-neighborhood contains both points in the domain and points lying outside of the domain. Finally, an **extreme point** is a boundary point which does not lie between any other two points in the domain, or more formally, the extreme point cannot be expressed as a convex combination of any other points in the domain. (*) These definitions are extremely important since we will concentrate on boundary and extreme points of a domain in selecting test cases.

The domain testing strategy is first developed, explained, and validated in detail for the simplest case possible. This initial analysis is formulated by making many assumptions, which will later be relaxed in generalizing the results. We first characterize a set of test points for borders defined by linear inequality predicates in a continuous two-dimensional space. The analysis is then progressively generalized to include both higher dimensional linear predicates and classes of nonlinear borders. We then validate the strategy for domain errors in equality and nonequality predicates, and finally we evaluate the effectiveness of the methodology for transformation and missing path errors.

6.2 The Two-Dimensional Linear Case

An error in the form of a predicate, i.e., relational operator, affects large areas of a domain, but a border shift may affect a very small area. Therefore, we first define a set of test points for detecting border shifts, and then we will show that this set of points also detects all possible relational operator errors. The following two assumptions are basic to this research and are used throughout our work.

(1) An "oracle" exists which can determine the correctness of every test case. It decides only if the output values are correct and not whether they are computed correctly. If they are incorrect, it does not provide any information about the error and does not tell us what the correct values are.

(2) Coincidental correctness does not occur for any test case. If correct results are produced, we assume that the test point is in the correct domain rather than being coincidentally correct in the wrong domain.

We make many other assumptions to facilitate the initial presentation of our results. Unlike the first two assumptions, the following are not essential in obtaining these results, and we will generalize our analysis to overcome each of these restrictions.

(3) The path corresponding to each adjacent domain computes a different function than the path being tested.

(4) The input space is continuous rather than discrete.

(5) Each border is produced by a simple predicate.

(6) Each border is produced by an inequality predicate.

(7) The given border is linear, and if it is incorrect, the correct border is also linear.

(8) The input space is two-dimensional, corresponding to a program which reads only two input variables.
Assumption 3 is important, since if paths for adjacent domains compute the same function, a test point for their common border will not be affected by a domain error which causes it to be in the wrong domain. By assuming a continuous input space we are able to present the domain testing results without the complications caused by finite data representations. These two assumptions are very important, but we will relax these restrictions in chapter 7. Assumptions 5 through 8 allow us to define a simple case for the initial presentation. They are not necessary for our results, and in the more general cases analyzed in later sections each of these assumptions will be eliminated.

The domain testing strategy defines a set of test points for each border. If any of these test cases produces incorrect results, the program contains an error, and the test data has successfully detected it. More importantly, the test points must be selected so that if correct results are produced for all the test points selected for a border, we can conclude that the border is correct. Since the present analysis is limited to linear borders in a two-dimensional input space, each border is a line segment. Therefore, the true border can be determined if we know two points which lie on it.

The results of the selected test points must allow the identification of two points which lie both on the given border and on the true border. We test each border with both ON and OFF points. For both closed and open borders, by definition, the ON points lie in a different domain than the OFF points. If correct results are produced for each test case, a true border must lie between each ON point and each OFF point. Therefore, we test each border segment with three points, two ON points and a single OFF point. These three points must be selected so that they form an ON-OFF-ON sequence along the border segment as shown in figure 21, where A and C are ON test points and B is the OFF test point. (*)

These three test points detect every border shift of a magnitude greater than 6. Since a true border must lie between each ON point and each OFF point, it must intersect the two line segments connecting B with A and C, where both BA and BC have an open end at point B and are closed at the other end. Therefore, we know that two points, one on segment BA and one on segment BC, lie on the true border, and this serves as a partial identification of the true

(*) In all figures small arrows indicate the domain which contains the border segment itself.
Figure 21: Test Points for A Two-Dimensional Linear Border
order, since it must be one of the lines intersecting these two line segments. In a continuous space we can make \( \varepsilon \) arbitrarily small, and as \( \varepsilon \) approaches zero, the OFF-ON line segments, \( BA \) and \( BC \), become arbitrarily close to the given order. In the limit, these line segments actually become indistinguishable from the given border itself, and the two intersection points between the line segments and the true border also lie on the given border. Two points determine a unique line, and since the given border and the true border have two points in common, we conclude that the given border is identical to the correct border. Since \( \varepsilon \) can be made arbitrarily small, an OFF point can always be found which can detect border shifts of any specified magnitude. However, the continuity of the input space also means that arbitrarily small border shifts are possible. Regardless of how small we make \( \varepsilon \), border shifts smaller than \( \varepsilon \) may not be detected, and therefore we must qualify our results. So, in conclusion, the three test cases will detect all border shifts greater than \( \varepsilon \), and this strategy can be applied to each border segment in the domain boundary.

By definition, a path containing a domain error may also compute an incorrect transformation. Any test point selected within the domain, viz., ON points for a closed border and OFF points for an open border, may produce incorrect results because of the border shift or the incorrect transformation. Therefore, an incorrect transformation only causes additional test points to produce incorrect results. Assuming that coincidental correctness does not occur, a test point which would detect the border shift cannot produce correct results because of an incorrect transformation. Therefore, the three test points defined above will detect domain errors whether the transformation is correct or not.

The single OFF point for each border must be selected between the two ON points in order to satisfy the ON-OFF-ON sequence requirement. In addition there is a second condition which must be satisfied by the selected OFF point. In certain cases it is possible that a border shift will not be detected if this additional requirement is not satisfied. For example, in figure 22, none of the three test points will detect the border shift, but this type of error should be detected by the OFF point. The problem is that the OFF point lies outside of the region bounded by the extensions of the adjacent borders, since this is the region affected by the border shift. This seems to be a pathological case since the interior angle between the borders is nearly 180 degrees and the extreme point is almost nonexistent. In addition, the OFF point is very close to the border, and as
Figure 22: Improperly Selected OFF Point
described in chapter 8 we select the OFF point midway between the UN points. However, this condition can occur regardless of where the OFF point is located, and we must check each OFF point to insure that it is properly located.

Since we can find OFF points arbitrarily close to the border, we can always find a point satisfying the above condition. If the OFF point satisfies all the constraints defining the domain, other than the constraint for the border being tested, it must lie within the region bounded by the extension of the adjacent borders. Therefore, the basic requirement that a border shift of the type depicted in figure 22 causes the OFF point to be in the wrong domain is satisfied. So, we are not completely free in choosing the OFF point, but it is very easy to check that the selected OFF point meets this requirement.

A more intuitive explanation of domain testing might be helpful at this point. Essentially the three test cases bracket the given border so that if it has shifted, one or more of the test cases must be in the wrong domain. In chapter five we identified three types of border shifts, and we can now see how the ON-OFF-ON sequence of test points works in each case. Figure 23 shows the three types of border shifts and the selected test points, and again the solid lines are the given borders, and the dashed lines are the true borders. In the upper left hand diagram the border shift has reduced domain UL. Test points A and C will be processed correctly since they are in domain D1 and would still be in domain D1 if the border had not shifted. However the border has shifted past test point B, causing it to be in domain D2 instead of domain D1. Since the program will now follow the wrong path when executing point B, incorrect results will be produced. In the upper right hand diagram test point B will be processed correctly since it is still in domain D2, but both A and C will detect the shift since they should also be in domain D2. Finally, in the lower diagram only point C will be incorrect since the border shift causes it to be in D1 instead of D2. We see that because of symmetry one or the other of the ON points will always be affected by this type of border shift. In addition the OFF point might also be shifted from domain D1 to D2. Therefore, the ON-OFF-ON sequence is effective since at least one of the three test points must be in the wrong domain as long as the border shift is greater than ε.

We must also demonstrate the reliability of domain testing for domain errors in which the form of the predicate is incorrect, i.e., the wrong relational operator is used. If the direction of the inequality is wrong, e.g., ≤ is used instead of ≥, the domains on either side of the border are
Given Border

True Border

Figure 23: Domain Testing For Border Shifts
interchanged, and any point in either domain will detect the error. A more subtle error occurs when just the border itself is in the wrong domain, e.g., $<$ is used instead of $\leq$. In this case the only points affected lie on the border, and since we always test UN points, this type of error will always be detected. It is also possible that the correct predicate is an equality rather than an inequality. In this case the correct predicate would define one domain consisting of just the border and another domain consisting of the regions on either side of the border. One or the other of these regions is therefore associated with the wrong path in the given program, and any point in this region will detect the error. These relational operator errors are more severe than border shifts in the sense that all points in a large region are affected, and therefore they are easier to detect.

The domain testing strategy requires $3B$ test points for a domain, where $B$, the number of border segments in the boundary, is bounded by the number of inequality predicates encountered on the path. However we can reduce this cost by sharing test points between adjacent borders of a domain. The only requirement for the UN points is that they must be boundary points. Since an extreme point is a boundary point for two adjacent borders, it can be used as an UN test point for both borders. In this case the extreme point serves a double purpose, and the number of UN points needed to test the entire domain boundary can be reduced by as much as one half. This is demonstrated in figure 24 by points A, B, and C. So the number of test points, $TP$, required to test the complete domain boundary lies in the following range.

$$2B \leq TP \leq 3B$$

This type of sharing is possible regardless of whether the borders are open or closed, and figure 24 contains examples of all three possibilities: adjacent borders which are both closed, both open, and mixed.

Even more significant savings are possible by sharing the test points for a common border between two adjacent domains. If both domains are tested independently, the common border between them is tested twice, using a total of six test points. If this border has shifted, both domains must be affected, and the error will be detected by testing either domain. Therefore, the second set of test points is unnecessary and can safely be omitted. In detailing these possible efficiencies we have implicitly assumed that the cost of testing can be measured by the number of points which must be generated. However, we must be careful since the amount of extra work necessary to realize these
Figure 24: Domain Test Points for Closed and Open Borders
efficiencies may outweigh the savings they entail. In section 7.5 we consider the global problem of test point sharing over all domains of the input space.

The basic geometric argument used in validating the effectiveness of an ON-OFF-ON sequence of test points can also be applied to an OFF-ON-OFF sequence. Since we can still construct two OFF-ON line segments which identify two points common to both the given and correct border, an OFF-ON-OFF sequence also detects border shifts greater than $\Theta$. However, the ON-OFF-ON sequence is preferable for many reasons. The use of a single extreme point as an ON test point for two adjacent borders is impossible using an OFF-ON-OFF sequence, and therefore more test points are needed. Furthermore, it may be easier to generate extreme points than OFF points near extreme points. In addition, the previously described problem of selecting an effective OFF point would be encountered in selecting both OFF points. Therefore, even though they both work, ON-OFF-ON testing is more efficient than OFF-ON-OFF testing, and in later sections we will extend the strategy based on the use of ON-OFF-ON testing sequences.

We now summarize the results of this section as a theorem. We first briefly restate the assumptions used in this section.

1. An "oracle" is available to determine the correctness of each test case.
2. Coincidental correctness does not occur for any test point.
3. A different function is computed for each adjacent domain.
4. The input space is continuous.
5. Each border is produced by a simple predicate.
6. Each border is produced by an inequality predicate.
7. The given border is linear, and if incorrect the true border is also linear.
8. The input space is two-dimensional.

We have defined $\Theta$ as the shortest distance from the OFF test point to the given border. In addition, the magnitude
of a border shift is the shortest perpendicular distance from the given border to the correct border. Finally, we define $B$ as the number of border segments in the domain boundary.

\[ \text{THEOREM 6.1} \quad \text{Given assumptions (1) - (8), the domain testing strategy is guaranteed to detect all domain errors of magnitude greater than } \varepsilon \text{ at a cost of no more than } 3B \text{ test points per domain.} \]

\[ \text{PROOF:} \quad \text{The ON-UFF-ON sequence of test points allows us to construct two } \text{ON-UFF line segments which lie within } \varepsilon \text{ of the given border. Assuming that correct results are produced for all three test points, a true border must intersect each ON-UFF segment. Therefore, a true border must lie within } \varepsilon \text{ of the given border, and all border shifts greater than } \varepsilon \text{ will be detected.} \]

\[ \text{The } \varepsilon \text{-limitation represents the worst case, since domain testing may detect border shifts smaller than } \varepsilon \text{, but their detection is not guaranteed. In any case, if } \varepsilon \text{ is small, the practical significance of this limitation will be negligible in most cases.} \]

\[ \text{0.3 The General } n\text{-Dimensional Linear Case} \]

\[ \text{The domain testing strategy developed for the two-dimensional case can be extended to the general } n\text{-dimensional case in a straightforward manner. The central property used in the previous analysis was the fact that a line is uniquely determined by two points. We can easily generalize this property since an } n\text{-dimensional hyperplane is determined by } n \text{ linearly independent points. So, whereas in the two-dimensional case we had to identify only two points on the true border, in general we have to identify } n \text{ points on the true border, and in addition we must guarantee that these points are linearly independent.} \]

\[ \text{The key to our methodology is that the true border must intersect every UFF-ON line segment, assuming that the program processes the set of test points correctly. Since} \]
we must identify a total of \( N \) points on the true border, we need \( N \) \text{OFF-ON} line segments, and we can achieve this by testing \( N \) points on the given border and a single \text{OFF} point. So we select \( N \) linearly independent \text{ON} test points and a single \text{OFF} test point whose projection on the given border is a convex combination of these \( N \) points. In addition, as in the two-dimensional case, the \text{OFF} point must also satisfy the constraints corresponding to all adjacent borders. Figure 25 demonstrates the set of test points for the three-dimensional case. The border depicted is one face of the domain boundary and is part of a plane in this three-dimensional case. \( X \) denotes a test point as usual, and the \text{OFF} test point is circled.

The validation of domain testing for the general linear case is based on the same geometric arguments used in the two-dimensional case. Assuming that all test points produce correct results, the \( N \) points must lie on one side of the true border, and the \text{OFF} point must lie on the other side. Therefore, the true border must intersect each \text{OFF-ON} line segment. Even though we do not know the specific intersection points, we do know that they must be linearly independent since the \( N \) points are linearly independent and the \text{OFF} point is a convex combination of the \text{ON} points. The \text{OFF} point is a distance \( \theta \) from the given border, and in the limit as \( \theta \) approaches zero, each \text{OFF-ON} line segment becomes arbitrarily close to the given border. So, in the limit, the \( N \) points which we know to lie on the true border also lie on the given border, and therefore we conclude that the given border is correct. However, as in the two-dimensional case, the \( \theta \)-limitation means that only border shifts greater than \( \theta \) will be detected. This development closely parallels that used in the two-dimensional case, and we should also note that the strategy is the same since the selection of the \( N+1 \) test points, along with the general independence criteria, is equivalent to the required \text{ON-OFF-ON} sequence when \( N=2 \).

The linear independence of the \( N \) points known to lie on the true border is a basic requirement of the methodology. It is guaranteed by the linear independence of the \text{ON} points and the selection of an \text{OFF} point whose projection on the given border is a convex combination of the \text{ON} points. Figure 26 shows test points for the three-dimensional case in which the \text{ON} points are collinear rather than linearly independent. In this case we cannot even be sure that we have identified three distinct points since a single point on the true border can lie on two of the \text{OFF-ON} line segments. Figure 27 shows test points in which the \text{OFF} point has been chosen improperly. In this case we have identified three points, but we cannot be sure that they are
Figure 25: Domain Test Points for a Three-Dimensional Border
Figure 26: Necessity of Linearly Independent ON Test Points

Figure 27: Necessity of Properly Chosen OFF Test Point
linearly independent, i.e., noncollinear in the three-dimensional case. In either event the geometric arguments used to validate the domain testing strategy are not valid. Therefore, the ON test points must be linearly independent, and the OFF point must be chosen so that its projection on the given cycle is a convex combination of the ON points. However, these requirements will be easily satisfied using the implementation technique described in chapter 3.

The domain testing strategy requires \((N+1)B\) test points per domain, where \(N\) is the dimensionality of the input space in which the domain is defined and \(B\) is the number of border segments in the boundary of the specific domain. However, we again can reduce this testing cost by using extreme points as ON test points. Each extreme point is formed by the intersection of at least \(N\) border segments, and therefore one point can be used to test up to \(N\) borders. In addition, extreme points are linearly independent by definition, and therefore we need not worry about checking for this property. Each border must be tested by \(N\) ON points, and any points beyond this are redundant and are unnecessary. So each extreme point will test \(N\) borders only in the optimal case, and we can usually expect a particular extreme point to be a redundant test for some of the borders it lies on. This allows us to calculate bounds on the number of required test points as expressed by the following inequality, where \(TP\) is the number of test points.

\[
2B \leq TP \leq B(N+1)
\]

So again we see that the number of test points can be significantly reduced by using extreme points rather than just boundary points. In addition, the sharing of test points for a common border between adjacent domains can be used to further reduce the total number of points required to test all the domains. Finally, since some of the \(B\) border segments may be produced by the min-max constraints which define the bounds of the input space, the number of test points can be reduced still further, if we can assume that these constraints are predetermined and need not be tested for domain errors.

We summarized the results of the last section as a theorem utilizing many assumptions. This section has addressed the serious and restrictive assumption of a two-dimensional input space. We have shown that the domain testing strategy can be extended to general \(N\)-dimensional input spaces, and therefore assumption (8) is no longer required. However, we still need assumptions (1) through (7), and therefore we restate them here.
(1) An "oracle" is available to determine the correctness of each test case.

(2) Coincidental correctness does not occur for any test point.

(3) A different function is computed for each adjacent domain.

(4) The input space is continuous.

(5) Each border is produced by a simple predicate.

(6) Each border is produced by an inequality predicate.

(7) The given border is linear, and if incorrect the true border is also linear.

Since this section extends the scope of theorem 6.1 rather than presents a completely new result, we summarize this section with the following statement of results rather than by restating theorem 6.1.

result 6.2 Given assumptions (1) - (7), the domain testing strategy is guaranteed to detect all domain errors of a magnitude greater than $\varepsilon$ regardless of the dimensionality of the input space. In addition, the cost is not more than $(n+1)B$ test points per domain.

This generalization is very significant since very few nontrivial programs have only two input variables. In addition the cost, as measured by the number of required test points, is reasonable and at worst grows only linearly as a factor of both the dimensionality of the domain and the number of border segments in the domain boundary.
0.4 The General Nonlinear Case

In this section we extend the methodology developed for linear borders to the more complex class of nonlinear borders. The domain testing strategy works well for linear borders because of their predictable geometric properties, and this allowed us to identify the correct border based on the results of a small number of test cases. In particular the number of points needed to identify each border was completely determined by the dimensionality of the domain. The linearity of the given border also enabled us to conclude that each of the intersection points between the true border and an OFF-ON segment would also lie on the given border in the limit as \( \theta \) approaches zero. We must try to apply these techniques to the class of nonlinear borders by formulating nonlinear analogies to the properties which were essential in the linear case.

The general class of all nonlinear functions can produce borders which are arbitrarily complex and which lack the predictable behavior so useful in the linear case. Therefore, the domain testing strategy cannot be applied to the total class of nonlinear borders. In addition, a domain constrained by nonlinear borders lacks the geometrical properties exhibited by linearly constrained domains, which are guaranteed to be convex polytopes. A finite number of points cannot be used to uniquely determine a nonlinear function, since there are always many functions of higher degrees of nonlinearity which pass through any finite set of points. This can be explained more intuitively by picturing a finite sequence of UN-OFF-ON-UN.... test points selected along a two-dimensional nonlinear border. Regardless of how many points are tested, there can always be some other border with a higher degree of nonlinearity which can weave in and out, producing the observed behavior at each test point, as diagrammed in figure 2d.

A finite domain testing strategy cannot be effective for the universal class of nonlinear borders, but we must determine whether this is caused by some fundamental difference between linear and nonlinear functions. If the problem is that we are considering too general a class of borders, then we should be able to extend the methodology to some well-defined subclasses of nonlinear functions. However, if the problem is caused by some basic characteristic of nonlinear borders, we will not be able to extend domain testing to any class of nonlinear functions.
Figure 28: Complexity of A General Nonlinear Border
The basis of our methodology is the identification of a finite set of points which are common to both the given border and the true border. We need to identify enough points to uniquely determine a particular function of the type and dimensionality being considered. In order to extend our results to particular subclasses of nonlinear functions, such as quadratic or cubic polynomials, we must assume that if the given order is incorrect, the correct order is in the same subclass, i.e., is of the same degree of nonlinearity.

In limiting the scope of possible correct borders we have removed the major obstacle to extending the domain testing strategy. For any specific subclass of nonlinear functions we can now easily determine the number of points needed to completely specify a particular function in the subclass. So, for example, we can successfully test a quadratic order and conclude that no other border having the same or a lower degree of nonlinearity, i.e., quadratic or linear, is possible. However, our assumptions do not allow us to make any conclusions about other types of higher degree nonlinear orders such as cubics, fourth degree polynomials, etc..

The number of points needed to determine a unique polynomial of a specific subclass can best be explained by analyzing the problem algebraically rather than in its geometric form. Given the dimensionality of the polynomial, i.e., the number of variables, and the degree of nonlinearity, we can formulate the most general form of the polynomial, which will include all possible terms. For example, the general form of a two-dimensional quadratic is given below, where \( X \) and \( Y \) are the variables, and \( A, B, C, \ldots \) are coefficients.

\[
AX^2 + BY^2 + CXY + DX + EY + F = 0
\]

The particular class of polynomials under consideration is determined by the form of the given border, and since we have assumed that the correct border is of the same type, it must also be described by this general polynomial form. Of course some coefficients might be zero for either the correct or the given order. So basically the problem is to determine unique values for the coefficients, given a set of points, i.e., \((X,Y)\) pairs, known to be on the correct border. Since these points also lie on the given border, we can conclude that it is correct.
The equation for a two-dimensional quadratic can be rewritten as follows.

\[ X^2 + 2XY + 2Y \cdot b + X^2C + X^2D + Y^2E + 1 \cdot F = 0 \]

In this form the original coefficients, A through F, are the variables, and the original variable terms, e.g., XY, X, and Y, act as the coefficients. Therefore, since we know the coordinates of each specific point, \([X(1),Y(1)]\), a homogenous equation which is linear in variables A through F can be generated for each point. This system of linear equations can be represented as the following matrix equation, which can be solved using existing mathematical techniques.

\[
\begin{bmatrix}
X(1) & Y(1) & X(1)Y(1) & X(1) & Y(1) & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
X(5) & Y(5) & X(5)Y(5) & X(5) & Y(5) & 1 \\
\end{bmatrix}
\begin{bmatrix}
A \\
B \\
C \\
D \\
E \\
F \\
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]

We define an independent set of points as a set which could be used to determine a unique polynomial in the subclass, and this concept should not be confused with linear independence as used in previous sections. The important issues are the number of points needed and how to determine if the points are independent. Since each point is known to lie on the given border, which itself is described by the general polynomial for the class, the system must be consistent. Therefore we need \(n-1\) independent points to be able to solve the system uniquely, where \(n\) is the number of variables in the system, e.g., we need five equations to solve for the six coefficients \((A,B,C,D,E,F)\) in the above example. Since the variables in this system are coefficients from the original polynomial form, we need one less point than the number of terms in the general polynomial form for the class.
In solving the above linear system we have five equations in six unknowns, and therefore one variable is a free parameter. This yields an infinite number of solutions, but since they differ only by a constant factor, they all represent the same function. So, in summary, we first determine the total number of terms in the most general polynomial for the specific subclass of nonlinear functions, and we need one less point than this number to uniquely determine a specific function in the subclass.

In the linear case the points found to be common to both the given border and the true border must be linearly independent in order for our geometric arguments to be valid, and a similar independence criterion can be formulated for the nonlinear case. Regardless of how many equations are actually generated, the rank of the matrix must be exactly \( K \), where \( K \) is one less than the number of terms in the general polynomial form. So the \( K \)th row of the matrix must not be expressible as a linear combination of the other \( K-1 \) rows. Since the matrix contains columns for the first degree terms, \( X \) and \( Y \), linear independence is a sufficient condition to insure that the matrix is of rank \( K \). However, it is not a necessary condition since even when the first degree terms are dependent, the higher degree terms can still be independent. Independence is easily maintained, and therefore we propose that any checking for it be done after the points are selected rather than as a precondition to selecting the points. In the rare event that a selected point produces a dependent linear equation, an additional point would have to be identified. (*)

A short example calculation of a unique function in a specific class should be helpful in clarifying these arguments. In order to keep the problem to a reasonable size, we consider the class of parabolas described by the following equation.

\[
2AX + BX + CY + D = 0
\]

Since there are four terms we should be able to determine a unique parabola from three independent points. The equation can be rewritten as

(*) A more detailed treatment of these elementary linear algebra concepts is available in HAILGØL, chapter 5, p. 162-187.
\[ x^2 A + xY + yC + 1D = 0 \]

and the following matrix equation can be formulated.

\[
\begin{bmatrix}
  x(1) & x(1) & y(1) & 1 \\
  x(2) & x(2) & y(2) & 1 \\
  x(3) & x(3) & y(3) & 1
\end{bmatrix}
\begin{bmatrix}
  A \\
  B \\
  C \\
  D
\end{bmatrix}
= \begin{bmatrix}
  0 \\
  0 \\
  0
\end{bmatrix}
\]

Assuming that we know that the points \((-1, -2), (3, -2), \) and \((0, -0.5)\) lie on the parabola, the specific matrix to be solved would be:

\[
\begin{bmatrix}
  1 & -1 & -2 & 1 \\
  1 & 3 & -2 & 1 \\
  0 & 0 & -0.5 & 1
\end{bmatrix}
\]

using Gaussian elimination we can transform this matrix into its equivalent row-echelon form.

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

This matrix can be used to solve for \(A, B, \) and \(C\) in terms of \(D, \) and \(A = D, B = -2*D, \) and \(C = 2*D.\) The desired parabola can then be determined as:

\[ x^2 - 2x + 2y + 1 = 0 \]
Note that since $w$ is essentially a free parameter, we have an infinite number of possible equations for the parabola, but since they differ from one another only by a constant factor, they all represent the same function.

A unique polynomial can be chosen from a specific class of polynomials using a finite set of points, because only a finite number of independent parameters are needed to describe the polynomial. For example, a circle is determined by three points, and the corresponding parameters are the radius and the two coordinates of the center. Therefore, if we can define a set of test points which can be used to identify points lying on a correct nonlinear order, the domain testing strategy can be extended to any class of nonlinear functions described by a finite number of parameters.

We can now relate these geometric and algebraic arguments to the actual testing strategy. In our methodology we do not actually have to identify a specific point which lies on both the given and true border, but rather we do characterize each of these points as lying on some ON-OFF line segment. So basically we must define a set of ON and OFF test points which would imply the existence of the necessary intersection points. In addition, these ON and OFF test points must be selected so that the independence of the intersection points is assured. We never actually have to solve a linear system of equations as described above, but we must insure that the set of intersection points characterized by the results of the selected test cases could be used in solving such a system. In summary, we will extend the domain testing strategy to subclasses of nonlinear orders by characterizing the set of test points necessary to infer the existence of $K$ independent points, common to both the given and true order. These intersection points are sufficient to determine a unique member of the class. Therefore, since they cannot lie on two distinct orders, we can conclude that the given order is correct and that no domain error greater than $\delta$ exists for the order.

The domain testing strategy can be extended to classes of nonlinear orders at a cost of $2K$ test points per order. We test the given order with $K$ ON-OFF pairs of points, such that the $K$ ON points are independent and each OFF point is chosen a distance $\delta$ from the corresponding ON point. Figure 27 demonstrates this set of test points for a nonlinear order. In order to clearly demonstrate a set of test points, the order in this diagram is only a two-dimensional quadratic, but the strategy itself can be used for all classes of nonlinear borders regardless of the
Figure 29: Test Points for A General Nonlinear Border
dimensionality of the input space. As in the linear case we must insure that the OFF points do not lie outside of the region bounded by the extensions of the adjacent borders. We know that the correct order must pass between the two points in each ON-OFF pair, and in the limit as \( \varepsilon \) approaches zero, we conclude that the ON point lies on the true border. We have identified \( K \) independent points common to both the given border and the correct border, and therefore we can conclude that the given border is correct within \( \varepsilon \).

Of course, since we have assumed that the correct order is no more complex than the given border, our conclusion is valid only for the specified class of nonlinear orders. However we could extend these results to larger classes by testing more than the specified \( 2K \) points. For example, if the given border is quadratic but we want to insure that not even a cubic correct order is possible, we would test the order as if it were a degenerate cubic, i.e., one in which all the third degree terms have zero coefficients. Even though it is theoretically possible to extend our results in this way, the rapid growth in the number of required test points makes its application very costly in most cases.

While a single OFF point was sufficient in the linear case, the independence criterion requires \( K \) OFF points for each nonlinear order. In the former case linearity allows the OFF point to be shared by all the ON points since the linear independence of the points identified as lying on the true border is guaranteed by the linear independence of the ON points themselves. If we were to test a nonlinear border with \( K \) ON points and a single OFF point, we would be able to conclude that the true and given borders intersect at \( K \) points, each of which lies on the given border somewhere along the curve connecting the OFF point and one of the ON points. However, since we do not know the specific points of intersection, we would not be able to conclude that these \( K \) points are independent. So, the required \( K \) OFF points allow us to conclude that the intersection points are the actual ON test points themselves, and since ON points are selected to be independent, the independence criterion is automatically satisfied. Furthermore, we know of no selection criterion for the ON points which would guarantee the independence of the intersection points using only one OFF point. Such a selection criterion would have to perform an analogous function to the linear independence of the ON points for the linear case. So an effective strategy requires the full set of \( 2K \) test points, and unfortunately \( K \) grows very rapidly as the dimensionality and degree of nonlinearity of the border increases. (*)
A two-dimensional nonlinear border is a very special case, and even though the general strategy is effective, a slightly different testing strategy can be formulated to reduce the number of required test points. The basic difference is that while the intersection of two higher dimensional nonlinear functions is in general some other nonlinear function containing an infinite number of points, the intersection between two-dimensional nonlinear borders is a finite set of points, the maximum number of which can be determined from the form of the function. For example a pair of two-dimensional quadratic curves can intersect in at most four points. This means that any set of more than four points cannot possibly lie on two distinct quadratics, and any five points uniquely determines a specific quadratic. Therefore, we do not have to worry about independence in the two-dimensional case, since any set of K-1 distinct points will produce a system of independent linear equations. For example, any three distinct points can lie on at most one circle, since two distinct circles cannot have more than two points in common.

We test a two-dimensional nonlinear border with K+1 points, e.g., six for a quadratic, selected in an ON-OFF-ON-OFF... sequence along the border as diagrammed for the closed border in figure 30. Since the correct border must pass through or to the outside of the given border at each ON point and inside the given border at each OFF point, the two borders must intersect an odd number of times, let us assume once, in each ON-OFF and OFF-ON interval along the border. The K+1 test points define K intervals on the border, each of which must contain at least one intersection point. We have shown that these K points must be independent, and since they cannot lie on two distinct borders, the given border must be correct within $\theta$.

We can also describe this technique in terms of the degrees of freedom provided by the equation describing the class of borders. A specific polynomial has limited degrees of freedom since it only has a finite number of coefficients. The correct border would have to weave in and out between the test points, as the dashed curve in figure

\[ (*) \text{ If } n \text{ and } u \text{ are the dimensionality and degree of nonlinearity of a border respectively, } K \text{ can be expressed as follows.} \]

\[ K = \sum_{i=1}^{u} C(n+i-1, i) \]
Figure 30: Testing A Two-Dimensional Nonlinear Border

Figure 31: Borders Tangential At ON Test Points
It shows, and this requires more degrees of freedom than the polynomial provides. In other words, no polynomial of the specific type has enough coefficients to allow it to trace the type of curve required to produce the observed test results.

There is one possible problem with this strategy which must be addressed. The correct border may be tangential to the given border at an ON point thus eliminating one of the postulated intersection points, and this situation is diagrammed in figure J. Even though there are only three intersection points in this case, we should notice that the dashed curve, representing a possible correct border, is similar to the one in figure 30 and would require the same degrees of freedom. Therefore, even though there are fewer than K intersection points, a polynomial of this class still cannot produce the required curve because it does not have enough coefficients. We should recall that two polynomials of a class can intersect in at most K-1 points but do not even have to intersect at all. A tangential intersection point requires as many degrees of freedom as the two intersection points it replaces, and therefore each tangential intersection reduces the possible number of total intersection points by one. For example, two distinct circles can intersect at two points, but if they intersect tangentially at one point, no other intersection point can exist. So this two-dimensional strategy is effective, and the cost is only K+1 test points per border instead of the 2*K test points required by the strategy described previously for general n-dimensional nonlinear borders.

The domain testing strategy has been extended to subclasses of nonlinear borders, but now we must address the practical problems of actually implementing the strategy. Obviously, the number of required test points grows very rapidly when we consider highly nonlinear functions in higher dimensional spaces. However, we can reduce the cost by imposing a more restrictive assumption on the class of possible correct borders. Instead of considering the complete class of polynomials having the same degree of nonlinearity as the given border, we can more narrowly define the class as polynomials which are expressed in the same terms as the given border. For example, if the given border is a parabola, the general class would be all quadratics, and the more restrictive class would be just the parabolas. In this simple case the savings are minor, but they are significant for more complex borders. For example a fourth degree border in four variables, described by

\[ 4 \quad 3 \quad 2 \]

\[ AX + BY + C2 + DW + E = 0 \]
would require eight test points using the restrictive subclass definition, but 138 test points would be required if we assume that any fourth degree polynomial is a possible correct border. Domain testing is very expensive for nonlinear borders, but it is somewhat flexible in that the cost can be reduced by restricting the class of correct borders considered. However, we must realize that this cost reduction is balanced by a corresponding reduction in the generality of the conclusions we can draw from the test results.

Regardless of the number of test points used, we still must face the problem of actually generating these points. In order to generate points in a domain defined by nonlinear borders we must be able to solve systems of nonlinear inequalities. Unfortunately, existing nonlinear programming techniques are not very powerful, and generating solutions or a nontrivial system is a formidable task. Our specific problem is even more difficult because the OH test points required by the domain testing strategy are very specific points rather than just any points satisfying the set of nonlinear inequalities. Therefore, at present it is doubtful that our methodology can be successfully applied to nonlinear borders.

Domain testing can be used for any class of functions which can be described by a finite number of parameters, if we assume that the correct border is in the same class of functions as the given border. Therefore, there is really nothing unique about linear orders, and they just constitute one of these classes. This is a significant generalization of our results, since we can now replace the linearity assumption with a much less restrictive assumption, number seven in the list below. We summarize the results presented to this point with the following list of assumptions and statement of results.

(1) An "oracle" is available to determine the correctness of each test case.
(2) Coincidental correctness does not occur for any test point.
(3) A different function is computed for each adjacent domain.
(4) The input space is continuous.
(5) Each border is produced by a simple predicate.
Each border is produced by an inequality predicate.

If the given border is incorrect, the true border is in the same class.

While these nonlinear results are promising, their general implementation must await the development of more sophisticated and powerful nonlinear programming algorithms. This is not true for the linear case, and an efficient implementation of domain testing for linear borders is described in chapter 8.

6.5 Testing Equality and Inequality Predicates

Each equality predicate must be tested for domain errors, and in addition the successful testing of these predicates reduces the number of test points required for the other types of predicates. The geometric similarity between inequalities and equalities allows us to apply the domain testing strategy to equality predicates with minor modifications. In this section we analyze linear equalities, and since the strategy is based on domain testing for linear inequalities, these results can be extended to nonlinear equalities in an analogous fashion to that developed in the previous section.

We first review the role of equality predicates in defining a domain, as first developed in section 4.3. Basically, each independent equality constrains the domain to lie in a lower dimensional space. So, if we have an n-dimensional input space and the domain is constrained by L' linear equalities, L of which are independent, the domain would be a convex polytope of dimensionality n-L, and L'-L of the equality predicates would be redundant.
inequality and nonequality predicates then define the domain within the lower dimensional subspace defined by the set of equality predicates. Even though each inequality and nonequality defines an $n$-dimensional hyperplane, the domain border produced is actually a cross-section of this hyperplane in the subspace defined by the equalities. This cross-section appears as a lower dimensional hyperplane and can be tested as such by our methodology. Therefore, if we effectively test the equality predicates, we can conclude that the domain is being constrained to the appropriate subspace. This means that each inequality and nonequality predicate can be tested with fewer ON points since the effective dimensionality is reduced.

An equality predicate is equivalent to a compound inequality predicate, e.g., $Y = X + 2$ produces the same constraint as $(Y \geq X + 2$ AND $Y \leq X + 2)$. In addition, this compound predicate can be analyzed as a limiting case of the compound inequality below, where $\varepsilon$ is some positive quantity.

$$Y \geq X + 2 \text{ AND } Y \leq X + 2 + \varepsilon$$

A domain defined by a predicate of this type is shown in figure 32, along with the test points which would be generated for this compound predicate. The domain is pictured as a two-dimensional slab with parallel borders, separated by a distance $\varepsilon$. As $\varepsilon$ decreases, these borders move closer together, thus shrinking the domain. In the limit as $\varepsilon$ approaches zero, they become a single border corresponding to the original equality predicate. This analysis clearly shows how an equality predicate reduces dimensionality since the domain has been reduced from a two-dimensional region to a line segment.

We can use this analogy to formulate an effective set of test points for an equality predicate, and in figure 33 we see the equality border and the proposed set of test points. In a general $n$-dimensional domain each of the parallel inequality borders would be tested with $n$ ON points and a single OFF point, but when they become a single equality border, we use a total of $n$ points on the border and two OFF points, one on either side of the border. Of course the ON points must be independent, and the projection on the border or each OFF point must be a convex combination of the ON points.

As described in section 5.2, domain errors for equality predicates can affect the form or the position of the border. Again we must evaluate the effectiveness of the
Figure 32: Test Points for A Compound Inequality

Figure 33: Equality as A Compound Inequality in the Limit
proposed set of test points for each possibility. Any shift in the border must affect one or more of the ON points since we test N independent points and no other equality border can contain all of these N points. If the form of the border is incorrect and the correct predicate is a nonequality, every test point is affected, since the two domains defined by the predicate would be interchanged. Similarly, if the correct predicate is a strict inequality (\(<\), \(>\)), every UN test point will be in the wrong domain, since the border itself does not satisfy a strict inequality. Errors on which the operator should be \(<\) or \(>\) are also detected since one or the other of the OFF points is in the wrong domain.

Domain errors in which both the position and the form of the border are incorrect are more difficult to analyze. First we consider the case in which the border should be an inequality and has also shifted. Essentially the given border defines two domains, one containing all the ON points and the other containing the two OFF points. Therefore, the only way all N+2 test points can be correct is for the true border to be positioned so that the OFF points are on one side of the border and the ON points are on the other side. This situation is impossible since each OFF point was chosen so that its projection on the border would be a convex combination of the ON points. For example in figure 34 there is no way to draw a line which would separate the two ON points from the two OFF points. Therefore no inequality predicate can produce the observed test results.

The last possibility is that the border has shifted and the correct predicate is a nonequality. Unfortunately, in this case there are certain conditions under which the proposed set of N+2 test points would not detect the domain error. If we are so unlucky that both OFF points happen to lie on the correct border while none of the ON points belong to this border, the error would not be detected, and this remote possibility is diagrammed as the dashed border in figure 34. For example when the equality predicate is used in an IF construct, the ON points would execute the THEN branch and the OFF points would execute the ELSE branch. If we were to change the predicate to a nonequality and shift it as described above, the ON test points would then satisfy the nonequality condition and would still follow the THEN branch. Similarly the OFF points, which are now on the true border would not satisfy the condition and would still follow the ELSE branch. Obviously this is a specific condition which may not be very likely, but it is possible, and a solution must be found. We can solve this problem by testing one additional point selected so that it lies both on the given border and on the correct border. Since this
Given Border

Correct Border

Figure 34: Domain Testing for An Equality Predicate
point lies on the given order, it follows the THEN branch but it should follow the ELSE branch because it does not satisfy the correct nonequality condition. It may seem impossible to find this particular point since we do not know the correct order. However, the original set of test points fail only when both OFF points happen to lie on the correct border. So, if we choose this additional point on the line segment connecting the two OFF points, we guarantee that it will lie on the true order in the single case for which it is needed.

Each equality predicate can be completely tested using a total of \( N+3 \) test points, but we must determine the effective dimensionality of an equality predicate. If we have an \( N \)-dimensional input space and the domain is constrained by only a single equality predicate, then the dimensionality of the equality is clearly \( N \), and the domain is defined in a subspace of dimensionality \( N-1 \). We test this single equality with \( N \) independent ON points since this is the minimal set of points needed to uniquely determine an \( N \)-dimensional hyperplane. In the general case a domain will be constrained by \( L \) independent equality predicates. If we consider these \( L \) independent equalities as a single constraint, they define a particular subspace of dimensionality \( N-L \) in which the domain must lie. In testing the set of equalities, we are trying to determine if they define the correct subspace. So, rather than generate \( N \) ON points for each equality, we use a single set of \( N \) ON points to test the entire set of equalities. Since the equalities define an \( N-L \) dimensional subspace, we need \( N-L+1 \) independent ON points to ensure that it is the correct subspace. So, instead of the \( N \times L \) ON points which would be generated in testing each equality separately, we only need \( N-L+1 \) ON points when we take the global perspective of testing the complete set of equalities taken together. However we still need two OFF points for each equality to preclude the possibility of an equality having the wrong relational operator, and the additional ON point between each pair of OFF points will also be required. So, in summary, we can test the set of \( L \) equalities with a total of \( (N-L+1) + (2L+L) \) points.

The domain will then be defined in a subspace of dimensionality \( N-L \). In addition, since testing the equality predicates reduces the effective dimensionality of each of the inequality and nonequality borders, we need only test \( N-L \) ON points for each of these borders rather than \( N \). Of course, the importance of this reduction for a particular domain depends on the proportion of its predicates which are equalities and the proportion of these equalities which are redundant.
One final observation we can make concerns the sharing of test points. In testing each inequality order we will use \( n-L \) test points, and since these test points have already been selected to be independent, we should also try to make maximum use of them as test points for the equality predicates.

Each nonequality order can be tested in a similar way as for an inequality order. A nonequality order divides the domain into two subdomains by defining a hyperplane which is not part of the domain. Therefore, the nonequality order is equivalent to an inequality order which is open for each of these subdomains. This enables us to treat it as a border which is open on both sides, and the set of test points can be formulated using this analogy.

An open inequality order is tested with \( n \) independent ON points and a single OFF point, where \( n \) is the dimensionality of the border. Since a nonequality is open on both sides, we test it with \( n \) independent ON points and two OFF points, one on either side of the border. Since the nonequality border is formed in the subspace defined by the equality predicates, its effective dimensionality is \( n-L \). We should notice that this is the same strategy we formulated for testing an equality border, and this is not surprising since a nonequality predicate on one path corresponds to an equality predicate on some other path.

We propose to test a nonequality border with \( n-L+3 \) test points, and we must again evaluate the effectiveness of this strategy just as we did for equality borders. Because of the similarities between equalities and nonequalities, the arguments are almost the same, but a brief treatment is included for completeness. Any simple border shift will be detected since no other border can contain the \( n \) independent ON points. If the predicate should actually be an inequality, one or the other of the OFF points will be affected. In addition all the ON points will be affected if the inequality is not strict (\( \leq \)). Finally, if the correct predicate is an equality instead of a nonequality, all the test points are in the wrong domain.

If the predicate should be an inequality and the border has shifted, at least one test point is affected, since no border exists which separates the ON points from the OFF points. The last case to consider is when the border has shifted and should actually be an equality. In testing equality predicates we found that an additional test point was required to preclude all instances of this type of domain error. In this case the additional test point would
oe unnecessary if we could guarantee that some other test point, possibly an UFF test point for some inequality border, would not lie on the correct border. However, we have no way of knowing this and since a particular domain might not be defined with any inequality predicates at all, we cannot in general guarantee the existence of another appropriate test point. Therefore we again propose an additional test point lying at the intersection of the given border and the line segment connecting the two UFF test points. We test each nonequality using a total of $N-L+4$ test points. Whereas test points can be shared between equalities and inequalities, essentially utilizing some test points for multiple purposes, in general the test points required for nonequality borders are not useful in testing any other borders.

The most important result of this section is that the basic domain testing strategy, first developed for inequalities, is also effective with minor modifications for all other types of predicates and therefore will detect all domain errors greater than $E$. At this point the set of restrictions has been substantially reduced, and we summarize the results of this section below, using the following assumptions.

(1) An "oracle" is available to determine the correctness of each test case.
(2) Coincidental correctness does not occur for any test point.
(3) A different function is computed for each adjacent domain.
(4) The input space is continuous.
(5) Each border is produced by a simple predicate.
(6) If the given border is incorrect, the true border is in the same class.
RESULT 6.4 Given assumptions (1) - (6), the domain testing strategy is guaranteed to detect all domain errors of a magnitude greater than \( e \) for all types of predicates, as long as the given order can be described by a finite number of parameters.

0.6 Effectiveness for Transformation and Missing Path Errors

Even though we cannot guarantee that the domain testing strategy will be completely effective for transformation errors and missing path errors, it is possible that a point selected for domain testing will detect the existence of an error of these other types. Therefore, in this section we evaluate the partial effectiveness of domain test points for transformation and missing path errors.

In practice, domain testing will detect transformation errors in many cases. In testing for transformation errors we know that the domain is correct by definition, and therefore some transformation exists which will compute the desired results for every point in the domain. We have defined coincidental correctness for a data point as the calculation of correct output values by an incorrect transformation. This occurs when two different transformations produce the same result for some specific input data, e.g., \( A = B + C \) and \( A = B \times C \) are coincident when \( B = C = 2 \). In theorem 5.1 we proved that coincidental correctness is a theoretical limitation for any finite testing strategy, and therefore domain testing cannot be completely effective in detecting transformation errors, with the assumption that coincidental correctness does not occur, any single point in the domain is completely effective in detecting transformation errors, since only one transformation exists which could produce the correct output for the test point. So, coincidental correctness is the only limitation to the reliability of domain testing for transformation errors.

If we consider the entire class of computable functions, we know that there exist functions which coincide with one another at an arbitrarily high number of points. However, we expect two arbitrarily chosen functions to coincide at a set of points of measure zero, and an analysis
of coincidental correctness presented in chapter 7 confirms this. So, we cannot guarantee the detection of transformation errors, but since domain testing already requires that many widespread points in the domain be tested, we do not believe that it would be worthwhile to generate any extra test points.

A missing path error of reduced dimensionality corresponds to a missing predicate which is a simple equality constraint, and in theorem 5.2 we proved that no finite testing strategy can detect this type of error. Furthermore, in corollary 5.3 we proved that the only reliable strategy for the total class of missing path errors is an exhaustive test of all possible input data. So, domain testing cannot be reliable for the general class of missing path errors, and missing path errors of reduced dimensionality constitute one subclass which we know to be beyond the capability of our methodology. However, domain testing may be reliable for other types of missing path errors, and we can evaluate its effectiveness for these subclasses.

If a domain contains a missing path error, it must be divided into two or more domains, at least one associated with the test path and one or more associated with a new path which does not exist in the program. Therefore, only points in that part of the domain corresponding to the new path will be processed incorrectly, and the error will be detected only if a test point lies in this subdomain. Of course, an existing incorrect predicate, which is redundant or input-independent, may also cause a missing path error, and in this case the missing path is a control path which is infeasible.

A missing path error corresponds to a predicate which is missing from the program, and missing path errors are categorized by the type of predicate needed to correct the error. A missing predicate can be simple or compound. If it is simple, it can be an equality, inequality, or nonequality, and compound predicates can use the Boolean operators OR and AND. (*) In the following discussion we assume that all OR test points are chosen to be extreme points of the domain and also that the missing predicate is linear.

(*) The characteristic effects of each of these error types are described in depth in section 5.4.
If the missing predicate is a simple nonequality, the subdomain for the missing path is the entire domain except for the nonequality border itself. Regardless of the dimensionality of the domain, it is impossible for all the extreme points tested to lie on this border. Therefore, one or more of them must be in the subdomain for the missing path, and domain testing can detect this type of missing path error.

If the missing predicate is a simple inequality, the missing path corresponds to that part of the domain lying on one side or the other of the missing border. Since at least one extreme point must lie on either side of every hyperplane intersecting the domain, our strategy always detects this type of missing path error in the two-dimensional case, since we must test every extreme point of a two-dimensional domain. However, in higher dimensions we do not test every extreme point, and we cannot guarantee that one of the test points lies on either side of the missing border. Therefore, domain testing will detect some of these errors, but it is not completely reliable. Theoretically, we could detect all of these errors by testing all the extreme points of the domain. However, because the potential number of such points could become excessive, we believe that the benefits would be far outweighed by the extra cost incurred.

Compound predicates can be expressed using OR, AND, or as a complicated form containing both of these Boolean operators. If the missing predicate is expressed only with OR operators, the subdomain for the missing path is simply a union of subdomains described by simple predicates. Therefore, the detection of this type of error is no more difficult than missing path errors corresponding to simple predicates. In fact detection should be easier in this case since it is more likely that a test point lies in one of the many subdomains defined by the simple predicates in the compound missing predicate.

When the missing predicate contains one or more AND operators, the subdomain for the missing path is unpredictable and can be as small as a single point. This is the type of missing path error which was used in proving corollary 5.3, and therefore domain testing cannot be reliable for this subclass. More complicated missing predicates, containing both AND and OR operators, may also correspond to these unpredictable subdomains, and domain testing cannot be guaranteed to detect these errors.
In summary, coincidental correctness is a theoretical limitation to the reliability of any reasonable testing strategy, as was proven in theorem 5.1. However, we believe that the points defined by the domain testing strategy are as effective as any other points in the domain, and we do not propose enlarging the set of test points in an attempt to overcome the coincidental correctness limitation. Missing path errors of reduced dimensionality have been proven to be a second theoretical limitation to program testing, and our methodology cannot be guaranteed to detect these errors. Even though we have shown that certain subclasses of missing path errors can be detected by enlarging the set of domain test points, the number of points required may be exorbitant, and it is doubtful whether it would be cost effective for the majority of programs.
CHAPTER 7

EXTENSIONS OF THE DOMAIN TESTING STRATEGY

Many assumptions were required in presenting the results of the previous chapter, and the first four sections of this chapter address some of these restrictions. By assuming that all predicates are simple, we know that the domain is convex in the linear case. This is an important property, and section 1 analyzes the application of domain testing to domains defined with compound predicates. In reality the input space is discrete rather than continuous because of the finite precision of any realizable data representation, and in section 2 we make some observations concerning the differences caused by discreteness. To facilitate the clear presentation of our results we have assumed that adjacent domains compute different functions, and in section 3 we describe how domain testing can be used to overcome this restriction. We have proven that coincidental correctness is a theoretical limitation inherent to the testing process itself, but an analysis of this phenomenon is presented in section 4 showing that it may not be significant in many practical programs. We have shown that certain points can be used to test more than one order, and this raises the question of how much test point sharing is possible in the best case. In section 5 we develop an "optimal domain testing strategy" by characterizing the minimal set of points needed to test all domains of a two-dimensional input space. We have validated the domain testing strategy using various geometric and algebraic arguments. Therefore, in section 6 the effectiveness of the domain test points is demonstrated using specific programming errors in a short sample program.

7.1 Domain Testing for Compound Predicates

In the previous chapter we assumed that a path contains only simple predicates, and this means that the corresponding set of input points consists of a single
domain, which is an \(n\)-dimensional convex polytope if the predicates are linear. In this section we consider the differences associated with the use of compound predicates, and we demonstrate how our methodology can be generalized to test paths containing these predicates.

The set of inputs corresponding to a path is defined by the path condition, consisting of the conjunction of the predicates encountered along the path. For example, the path condition is \(C(1) \text{ AND} \ldots \text{AND } C(Z)\) for a path containing the simple predicates \(C(1)\) through \(C(Z)\). If a compound predicate of the form \(C(1) \text{ AND} C(i+1)\) is encountered on the path, the path condition is still a single conjunction of simple predicates, and the only difference is that two of the simple predicates are produced at a single branch point on the path. So, a compound predicate using only the boolean operator \(\text{AND}\) does not affect the form of the domain, other than the fact that two order segments may correspond to this compound predicate. Therefore, no modifications of the domain testing strategy are required in this case.

However compound predicates using the boolean operator \(\text{OR}\) are more complicated. Consider a path containing the following predicates:

\[
C(1), C(2), \ldots, [C(1) \text{ OR } C(i+1)], \ldots, C(Z)
\]

The path condition in this case is the conjunction of these predicates, and in the standard disjunctive normal form (sum of products), it is the following.

\[
[C(1) \text{ AND } \ldots \text{AND } C(1) \text{ AND } \ldots \text{AND } C(Z)] \text{ OR } [C(1) \text{ AND } \ldots \text{AND } C(1+1) \text{ AND } \ldots \text{AND } C(Z)]
\]

So, clearly, the set of input data points following this path consists of the union of two domains, each defined by the conjunction of simple predicates, and in general any number of these domains are possible.

Assuming linear predicates, each of these domains is a convex polytope, but the domains may overlap in an arbitrary way. The major problem caused by these compound predicates is that the domains correspond to the same path, and the assumption that adjacent domains do not compute the same function is violated. We identify three cases of importance: domains which are distinct, domains which partially overlap, and domains which totally overlap.
In the top diagram in figure 35, domains D1 and D2 are defined by the compound predicate C1 OR C2, and domain D3 corresponds to some other path. The two domains do not overlap, or more formally the intersection of D1 and D2 is null. In this case our methodology can be applied to each domain separately, since the two domains for this path are not adjacent.

In the middle diagram in figure 35, the domains partially overlap, where D1 U D2 is the domain defined by C1, and D1 U D3 is the domain defined by C2. This situation can be more formally defined as the case in which the intersection of the two domains is a proper subset of each domain, e.g., D1 is a proper subset of both D1 U D2 and D1 U D3. In addition, regardless of how small the overlap might be, e.g., just at the boundary or even a single point, we still must address the problem of adjacent domains which compute the same function. In the example we cannot test the domains separately, since they are adjacent and the same function is computed in each. For example, any test point for C1, selected along that part of the border between D1 and D3 is ineffective since the same results are computed for it in both of these regions. So, in this case we must insure that the adjacent domain assumption is satisfied by selecting test points for C1 and C2 which lie in that part of the border adjacent to a domain for some other path, as shown in the middle diagram. In section 3 of this chapter we describe how domain testing can be extended to insure that the adjacent domain assumption is satisfied, and the modified techniques described there will successfully select appropriate test points in this case.

The final possibility is shown in the bottom diagram in figure 35, and this situation occurs when the intersection of the two domains equals one of the domains. In the example one domain, D1, is a subset of the other, D1 U D2. This presents a serious problem since there are no test points for C1 which can satisfy the adjacent domain assumption, and therefore C1 cannot be tested effectively. However, we do know that a shift in border C1 must affect points in a region of the adjacent domain near border C2, as shown in the bottom diagram of figure 35. So, the test points for border C2 may detect a shift of border C1, but we cannot guarantee detection, as shown by the dashed border in the diagram. Since we do not need to generate any test points for border C1, we propose generating some extra OFF points for border C2. Although this modification should improve our ability to detect this type of error, it cannot be completely effective.
Figure 35: Domains Defined with Compound OR Predicates
So, in summary, a compound predicate of the form \( C_1 \land C_2 \) is the same as two simple predicates, and domain testing can be applied to a domain defined with this type of compound predicate. In addition, if the compound predicate is of the form \( C_1 \lor C_2 \) and the domains are distinct, domain testing can be applied to each domain separately. However, if the domains overlap, we must address the problem of adjacent domains which compute the same function. Section 7.3 discusses how domain testing can be modified to ensure that the adjacent domain assumption is satisfied. These results mean that we can select effective domain test points if the domains partially overlap. In addition, although we cannot find effective test points for domains which overlap completely, we can recognize this situation and identify it as a border which cannot be tested effectively.

7.2 Domain Testing for Discrete Input Spaces

In this section we make some observations about applying our methodology to a domain defined in a discrete input space. The results presented in the previous chapter were developed assuming a continuous input space, and this assumption was important in validating the effectiveness of the methodology. However, in reality the finite size of any realizable data representation means that the input space consists of a discrete set of data points rather than a continuum. Therefore, we must examine domain testing without the assumption of continuous input data and identify any problems caused by this discreteness.

A discrete input space consists of an \( N \)-dimensional lattice of points. In this discussion we assume that the lattice is homogeneous in all dimensions, and this means that the same data representation is used for all input variables. We also assume that the spacing of the representable data points is constant throughout the entire range of each variable. This assumption would exclude any type of floating-point data, i.e., data in scientific notation form consisting of a normalized fraction and an exponent, since the spacing between representable values increases as the absolute magnitude of the value increases. The actual spacing between data points is the smallest positive number representable using the data format. For example, a variable with precision \((8,5)\) consists of eight significant digits, five of which are fractional digits, and the representable values are spaced 0.0031 apart. So, in summary, a discrete \( N \)-dimensional input space is an
N-dimensional lattice, which is homogeneous both throughout the entire range of each variable and in all dimensions.

The domain testing strategy requires that a very specific set of data points be tested. The ON points must lie on the given border and be independent, and the OFF point must be arbitrarily close to the given border on the open side. In addition, the OFF point must lie within the region bounded by the extensions of the adjacent borders. Unfortunately, it may be impossible to find points satisfying these requirements in the discrete case. Since a border segment contains only a finite number of data points, we cannot guarantee that $n$ linearly independent points can be found which lie on the border. Therefore, it may be necessary to select some of the ON points near the given border on the closed side. Similarly, since an OFF test point cannot be found which is arbitrarily close to the given border, it will now be some small finite distance from the border. These are the basic problems caused by discreteness, and we must reevaluate the domain testing strategy in light of these limitations.

In the continuous case we were able to find the required set of ON test points actually on the given border and an OFF test point arbitrarily close to the given border. This allowed us to conclude that in the limit as $\theta$ approaches zero, each OFF-ON line segment becomes indistinguishable from the given border. However, with the limitations described above, we cannot conclude that the OFF-ON line segments are indistinguishable from the given border, but only that they are close approximations of the given border.

Figure 36 shows a two-dimensional domain and the test points which might be used. This diagram shows that only one of the ON points actually lies on the given border, and we now define $\theta$ as the perpendicular distance from the OFF test point to the line connecting the ON test points. The correct border must still intersect each OFF-ON line segment, but since these segments now differ from the given border, we cannot conclude that the correct border intersects the given border. We define an $\theta$-set of borders as the set consisting of all borders which intersect each OFF-ON line segment. Since the correct border intersects each OFF-ON line segment, we conclude that the correct border must be a member of this $\theta$-set. In addition, we should note that the given border is also a member of the $\theta$-set because of the way in which the test points are chosen, and therefore it may in fact be the correct border. So, we can conclude that the correct border lies within $\theta$ of the given border, where $\theta$ can be the order of magnitude
Figure 30: Domain Test Points for the Discrete Case
of the finiteness or grid size of the particular data representation used. The dashed lines in figure 36 are two of the possible correct borders in the ε-set.

We have identified a major difference between discrete and continuous input spaces as the magnitude of ε. In the continuous case we are able to make ε arbitrarily small, but in the discrete case it appears that at best ε is typically of the same order of magnitude as the spacing between representable data points. So, in the discrete case border shifts greater than ε are still detected, but ε is a finite, though relatively small, distance. The higher dimensional linear case and the nonlinear case are much more complicated, and a more thorough analysis is required for these cases. We do know that the correct order must be in the ε-set of borders even in these more general cases, but at this time we do not have any specific results to report about the size of this set or the possible degree of difference between borders in this set.

We now make some observations about the practical significance of being unable to detect border shifts less than ε in a discrete space. The problem caused by discreteness is that the specific points defined by the domain testing strategy may not be representable, and we may have to test some other less effective points. However, the discreteness of the input space could also be an advantage, since the effect of the undetected domain error may be less extensive. A border shift affects only those data points lying between the correct border and the given border, and in a discrete space this set of points is finite and may be relatively small. So, discreteness limits our ability to select effective test points, but it may also limit the effect of an undetected domain error.

In summary, discreteness causes problems for the domain testing strategy, but at the same time it may also prove to be an advantage. The observations presented in this section are based on a preliminary investigation of the discrete case, but a much more thorough mathematical analysis is required before we can formulate any complete results.

7.3 Adjacent Domains which Compute the Same Function

An important assumption made in developing the results in chapter 6 is that adjacent domains do not compute the same function. If two adjacent domains compute the same
function, any test point selected for their common border is ineffective, since the same output values are computed for the test point regardless of the domain in which it lies. In this section we demonstrate how domain testing can be modified to insure that this condition is satisfied.

We must insure that different output values would be computed for each test point if it were in the adjacent domain because of a border shift. For example, in the upper diagram in figure 37 we must compare the functions calculated in domains D1 and D2 for test point A, D3 and D1 for B, and D1 and D4 for C. The major problem to be solved is the identification of these adjacent domains. We assume that when testing domain D1 the partitioning structure of the adjacent domains and the program paths associated with these domains are not known. On the other hand, if the adjacent domains and associated paths were known, it would be possible to select test points for which the adjacent domains compute different functions.

Certain test points, viz., OFF points for closed borders and ON points for open borders, lie in the adjacent domains. Therefore, all we need to do for these points is to compare the output values calculated by the program with the output values obtained by applying the transformation for the test path. For example, in the upper diagram in figure 37, test point B lies outside domain D1, and in executing the program for B, we get the values computed by the function for domain D3. In addition, we can monitor the execution of the program for point B and identify the path corresponding to domain D3. Since we are testing domain D1, we must have executed the path for D1 symbolically in order to generate the predicate interpretations defining the domain. At that time we also would have calculated the transformation for D1, and we can assume that it is still available. So, we can easily calculate the output values which would be computed for point B if it were in domain D1. This technique can be used for each test point lying outside of the domain being tested.

The output values which would be computed by the transformation associated with the adjacent domain must also be determined for each test point in domain D1. This is more difficult, since we do not know which path corresponds to the adjacent domain for each of these test points. In this case we perturb the input space by replacing the border with one which is slightly shifted. In the lower diagram of figure 37 we have replaced the border by a parallel border shifted a small distance ε. We see that by perturbing the border in this way, we have caused the ON points, A and C, to be in the adjacent domains, D2 and D4, respectively.
Figure 37: The Identification of Adjacent Domains
Therefore, we can now calculate the output values computed in the adjacent domains by executing the modified program with test points A and C, and we can also identify the paths associated with these adjacent domains. The actual modification of the border is accomplished by adding or subtracting 6 from the right hand side of the corresponding predicate. So, again we can compare the results calculated by adjacent domains to determine whether the assumption is satisfied for each test point.

Knowing the values which would be computed in the adjacent domain for each test point, we can determine whether the adjacent domain assumption is satisfied. We have two sets of output values for each test point, that computed by the program and that which would be computed in the adjacent domain. In addition, we have identified the path associated with the domain adjacent to each test point. If the two sets of output values are different for each test point, then the adjacent domain assumption is satisfied, and all test points are effective in detecting domain errors. Furthermore, if this is true for even one test point, we can select effective test data, since at least one adjacent domain computes a different function than the test domain. For example, in figure 37, consider the case in which only point B is an effective test point. We therefore know that domain D3 computes a different function than D1. We would discard test points A and C and select new ON test points in that section of the border segment adjacent to D3. This can be done since we have identified the path for domain D3. So, as long as one test point is effective, we can select test points satisfying the adjacent domain assumption. However, the amount of extra work required in this situation may be excessive. First we must determine the bounds of the section of the border segment which is adjacent to D3. Unfortunately, this entails processing the predicates encountered on the path associated with domain D3. Of course, we must then generate new test points lying in this section of the border. So, the cost of satisfying the adjacent domain assumption may be unacceptable if many of the adjacent domains compute the same function.

A problem is encountered when none of the selected test points satisfy the adjacent domain assumption. In this case we have not identified an adjacent domain which computes a different function, but we have obtained a lot of information about the adjacent domains. For example, if the same path is associated with the adjacent domains for all three test points, it may be the case that only one adjacent domain exists, as shown in the upper diagram of figure 38. In this case the border is superfluous, and we do not even have to test it. If the paths associated with the adjacent
Figure 3d: Possible Adjacent Domain Structures
domains are different, we may be faced with the type of structure shown in the lower diagram in figure 38. In this case, effective test data exists if D5 or D6 computes a different function than D1, but it may be very difficult to determine if an adjacent domain such as D5 or D6 even exists. The amount of extra work in this situation may be excessive, and the search for effective test data does not seem to be cost-effective in this case. So, even though we may not find effective test data, at least we can identify those border segments which cannot be tested effectively.

So, in summary, domain testing can be modified to determine whether the adjacent domain assumption is satisfied. These extended techniques will insure that the adjacent domain assumption is satisfied for all generated test data. However, in certain cases it will only be able to identify a border segment which cannot be tested effectively because the adjacent domains compute the same function as the domain being tested. We have described techniques which might solve this problem, but the extra work required may be unacceptable in many cases. In addition, the extra information obtained through the use of these techniques may provide useful feedback to the user, but further research is needed to determine how best to utilize this additional information.

7.4 An Analysis of Coincidental Correctness

In this section we analyze the phenomenon of coincidental correctness which has been shown to be a theoretical limitation inherent to the testing process itself. Coincidental correctness prevents any reasonable testing methodology from providing absolute reliability. It offers one explanation of why the only completely reliable testing strategy is an exhaustive test of all possible input data points, since it is theoretically possible for all points in a domain but one to be coincidentally correct. In analyzing this phenomenon from a practical viewpoint, we hope to be able to make some observations concerning the practical significance of this problem. We will also formulate procedures designed both to guard against the possibility of coincidental correctness and to improve our chances of detecting it if it should occur.

Coincidental correctness affects our ability to detect both transformation and domain errors. For a particular test point, coincidental correctness occurs for a
transformation error when the given transformation, which is incorrect, computes the same output values as the correct transformation. If a domain error causes a test point to be in domain D' instead of domain D, coincidental correctness occurs when the given transformation for D' computes the same output values as the correct transformation for domain D. In either case the problem is caused by the fact that two different transformations can coincide, i.e., compute the same output values, for any particular input data point.

We first develop a canonical representation for a transformation, and then will analyze the effect of various changes to that transformation. We have defined the transformation associated with a path as the set of functions, expressed in terms of the input variables, computed for the M output variables. However, a transformation is usually represented by the actual sequence of assignment statements which are executed along a path. In the following discussion I(1) through I(N) are the input variables, V(1) through V(Z) are the program variables calculated along the path, and O(1) through O(M) are the output variables produced at the end of the path. In addition, f(i) is the function computed for V(i), and the variables within brackets following f(i) represent those variables which can appear on the right hand side of the particular assignment statement. Using this notation, a transformation can be represented as the following sequence of assignment statements.

\[
\begin{align*}
V(1) &= f(1)[I(1), \ldots, I(N)] \\
V(2) &= f(2)[I(1), \ldots, I(N), V(1)] \\
& \quad \vdots \\
V(i) &= f(i)[I(1), \ldots, I(N), V(1), \ldots, V(i-1)] \\
& \quad \vdots \\
V(Z) &= f(Z)[I(1), \ldots, I(N), V(1), \ldots, V(Z-1)] \\
O(1) &= f(Z+1)[I(1), \ldots, I(N), V(1), \ldots, V(Z)] \\
& \quad \vdots \\
O(M) &= f(Z+M)[I(1), \ldots, I(N), V(1), \ldots, V(Z)]
\end{align*}
\]

This general form seems very complicated, but essentially it models the fact that each program variable is computed in terms of the input variables and the program variables which have already been computed. In addition each output variable can be computed as a function of the input
variables and all the program variables, but not any other output variables. Finally, in order to make the discussion clearer, each assignment uses a unique variable name, and this is reflected by the sequential numbering of the V's. So, a transformation is a sequence of assignment statements, and each assignment statement has restrictions on which variables can appear in the arithmetic expression on its right hand side.

The value assigned to a program or output variable can be computed using any of the input variables and those program variables already assigned values. The assigned value depends on the value of the variables used on the right hand side of the assignment statement, and the variables used to compute those right hand side variables. This observation will allow us to formulate a data flow model for a transformation which will be very useful in explaining and understanding the mechanics of coincidental correctness.

An important concept must be defined in order to develop a data flow model. A data flow dependence exists between two variables, V(i) and V(j), when a sequence of variables starting with V(i) and ending with V(j) exists such that each variable in the sequence appears on the right hand side of the assignment statement for the variable immediately following it in the sequence. If the sequence consists of just two variables, i.e., V(i) is used in the assignment for V(j), we refer to it as a direct dependence. If the sequence consists of three or more variables, we refer to it as an indirect dependence. Using this definition we see that i must always be less than j, since a variable cannot be used until it is assigned a value. We should also note that an output variable can be the second variable in this relationship but never the first, since an output variable cannot be used to calculate some other variable. A short example program and the set of data flow dependencies are shown in figure 39. In addition, a data flow diagram is included to show these dependencies graphically. In this diagram each variable is represented as a node, each direct dependence is shown as a directed edge from V(i) to V(j), and each directed path of length greater than one represents an indirect dependence. This data flow model demonstrates how a single error propagates to subsequent assignment statements along the paths of data dependence. Since input variables cannot be computed incorrectly, their use in assignment statements does not affect error propagation. Therefore, in order to clarify this discussion, we have chosen not to include input variables in the data flow model.
READ A, B
C = A + B
D = 2*C - 11
E = 3*D - B
F = D - 21
G = 3*F - A
H = G - E + 31
WRITE H

DATA DEPENDENCIES

Direct: [(C, D), (D, E), (G, F), (E, H), (F, G), (G, H)]

Indirect: [(C, F), (C, G), (C, H), (D, G), (D, H), (F, H)]

Figure 39: Data Flow Model of A Short Example Program
In this discussion we restrict the type of assignment statements to those which are expressed using a simple set of arithmetic operators (+, -, *, /). Therefore, each assignment statement in the transformation can be further broken down into a sequence of triples, each consisting of an operator and two operands. This form is frequently used internally by a compiler in translating an assignment statement into object code, and the particular sequence of triples generated depends on such things as operator hierarchy, placement of parentheses, etc. A short example transformation and the equivalent sequence of triples should clarify this discussion, where each T represents a temporary value computed by a triple.

\[
\begin{align*}
C &= 2*(A+B) - 1 \\
D &= C/2 + 2 \\
E &= D - 3*(C+1) + B/3
\end{align*}
\]

\[
\begin{align*}
T(1) &<---- < + A B > \\
T(2) &<---- < * 2 T(1) > \\
C &<---- < - T(2) 1 > \\
T(3) &<---- < / C 2 > \\
D &<---- < + T(3) 2 > \\
T(4) &<---- < + C 1 > \\
T(5) &<---- < * 3 T(4) > \\
T(6) &<---- < - D T(5) > \\
T(7) &<---- < / B 3 > \\
E &<---- < + T(6) T(7) >
\end{align*}
\]

We analyze coincidental correctness by evaluating how various changes affect the results computed by a sequence of triples. Therefore, we start with a sequence of triples representing the correct transformation. Our goal is then to categorize the possible types of changes which can be applied to this sequence of triples in order to formulate a set of specific conditions necessary for coincidental correctness.

Assuming that a change in the sequence produces a new sequence which is valid, i.e., represents some valid sequence of assignments statements, there are three possibilities. The new sequence may represent the same transformation, and in this case an error does not exist. The new sequence may represent a transformation which does not coincide with the original transformation at any point at all. The third possibility is the only one of interest, since by modifying the sequence, we produce a new sequence representing an incorrect transformation which is
coincidentally correct for certain input values. Therefore, for each type of modification we will try to determine the conditions under which coincidental correctness occurs.

First we consider changes in which a single component of some triple is altered. Of course, in this analysis we are implicitly assuming that two errors or two effects of a single error do not cancel one another, but this assumption will be relaxed later in this section. In this case either a different operator is used or there is an error in one of the operands. Consider a sequence of triples calculating temporary values \( T(1), T(2), \ldots, T(i), \ldots, T(Z) \). If we alter a component of some triple in this sequence, for \( T(i) \), there are two ways for the new transformation to coincide with the original transformation. Immediate coincidence occurs if the value calculated for \( T(i) \) is unchanged for some specific input data. In this case there is no error propagation since each subsequent triple using \( T(i) \) as an operand will still be using the correct value. On the other hand, if the value now calculated for \( T(i) \) is incorrect, each triple using \( T(i) \) is affected, since one of its operands is now incorrect. However, coincidental correctness can still occur in this case if the value calculated for each \( T(j) \), where the triple for \( T(j) \) uses \( T(i) \) as an operand, does not change even though the value of the operand \( T(i) \) has been altered. We call this propagated coincidence, since the error starts to propagate to later assignments before coincidence occurs. These two concepts are related in that propagated coincidence for \( T(i) \) occurs only when immediate coincidence affects the error propagated to each \( T(j) \). So, in the first case the effect of the error is masked at the triple in which the error exists, but in the latter case coincidence would have to occur in every subsequent triple in which \( T(i) \) is used. The difference between these two definitions is the relationship between where the original error occurs and where it ceases to affect the computation. This is where a data flow model can be helpful since the effects of an error will spread along the paths defined as data flow dependencies. Coincidental correctness can occur only if the error is masked somewhere along each possible data flow path, leaving from the original source of the error to the assignment statements for the output variables. Therefore, propagated coincidence would seem to be less likely than immediate coincidence, since the error propagation increases the number of occurrences of coincidence necessary for the final output values not to be affected.

The easiest way for coincidental correctness to occur is when the error does not even affect the value calculated for \( T(i) \), the output of the triple containing the error.
Therefore, we first analyze errors in which one of the three components of a triple is changed, and we formulate a small set of simple conditions necessary for coincidental correctness to occur within the triple itself. We will consider a correct triple of the form

\[ T(i) \leftarrow \text{<op} \ X \ Y > \]

where the operator is one of the set \((+,-,*,/\), and \(X\) and \(Y\) are the first and second operands respectively. In addition, \(X'\) and \(Y'\) denote altered and therefore incorrect operands.

The conditions necessary for coincidental correctness can be determined very easily. For example, consider the case in which the operator \(+\) is replaced by the operator \(*\). These two triples will produce the same value for \(T(i)\) only when \(X + Y = X*Y\), and we can transform this condition into the standard form \(X = Y/(Y-1)\). So we can conclude that if \(X \neq Y/(Y-1)\), no coincidental correctness of this type can occur. In addition, the change of operator is obviously symmetric, and the same condition is required for coincidental correctness when the error has replaced \(*\) by \(+\). So, we can conclude that an interchange of \(+\) and \(*\) will go undetected only when \(X = Y/(Y-1)\). Similar conditions can be formulated for every other possible operator error, and the total set is summarized below, where the necessary condition is listed in the column on the right.

<table>
<thead>
<tr>
<th>Error Case</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (\leftarrow) -</td>
<td>(Y = 0)</td>
</tr>
<tr>
<td>+ (\leftarrow) *</td>
<td>(X = Y/(Y-1))</td>
</tr>
<tr>
<td>+ (\leftarrow) /</td>
<td>(X = (Y**2)/(1-Y))</td>
</tr>
<tr>
<td>- (\leftarrow) *</td>
<td>(X = Y/(1-Y))</td>
</tr>
<tr>
<td>- (\leftarrow) /</td>
<td>(X = (Y**2)/(Y-1))</td>
</tr>
<tr>
<td>* (\leftarrow) /</td>
<td>(X=0) or (Y=1) or (Y=-1)</td>
</tr>
</tbody>
</table>

A similar analysis can be applied to an error in which one of the operands is altered or has been assigned an incorrect value. For example, if the triple \(< * X Y >\) is replaced by \(< * X' Y >\), the error in the first operand will not affect the value assigned to \(T(i)\) if \(Y=0\) since \(X*0 = X'*0\). Again all possibilities can be enumerated as follows, where we list the original triple, the new triple, and the conditions under which the value of \(T(i)\) is unchanged.
As we can see from this list there are extremely few situations in which an error in an operand will not affect the value calculated for T(i). This is particularly significant, since an operand error occurs whenever an error propagates to a subsequent triple.

We can now summarize the results for coincidental correctness with a single error in any triple. The list below contains all the conditions which must be avoided in order to preclude this type of coincidence for a triple containing the specified operator.

\[
\begin{align*}
+ & \quad X = Y/(Y-1) & - & \quad X = Y/(1-Y) \\
+ & \quad X = (Y**2)/(1-Y) & - & \quad X = (Y**2)/(Y-1) \\
+ & \quad Y = 0 & - & \quad Y = 0 \\
* & \quad X = Y/(1-Y) & / & \quad X = (Y**2)/(1-Y) \\
* & \quad X = Y/(Y-1) & / & \quad X = (Y**2)/(Y-1) \\
* & \quad X = 0 & / & \quad X = 0 \\
* & \quad Y = 0 & / & \quad Y = 1 \\
* & \quad Y = 1 & / & \quad Y = -1 \\
* & \quad Y = -1 & / & \quad Y = 1
\end{align*}
\]

From this short list of specific conditions it seems very unlikely that an error will not affect the output of a triple. In addition, if we monitor the execution of each test case, we can insure that these conditions do not occur for any triple executed along the path, and we thereby would greatly reduce the possibility that any of the test points are coincidentally correct.

As described previously, propagated coincidence can occur when the value of T(i) is affected by an error in its triple. The incorrect value for T(i) becomes an operand error in every triple using T(i) as an operand. Basically, the error in T(i) would have to be masked somewhere along each data flow path from T(i) to each of the assignment statements calculating an output variable. When the error propagates in this fashion, it always acts as an operand
error. We have seen that an operand error produces immediate coincidence only when we multiply it by zero or divide it into zero, and therefore these propagation errors are even less likely to produce coincidence. In any event this condition can be detected, and any test case in which it occurs can be replaced.

A single error occurring in any one triple has been shown to be very resistant to coincidental correctness, and the special conditions under which it could occur have been enumerated and can easily be avoided. However, since a transformation can contain more than one error and can be affected by a single error propagating along many data flow paths simultaneously, we must also consider the case in which two or even three components of the triple are incorrect. An exhaustive enumeration and analysis of all these cases has been performed similar to that described for single errors, and the results can be summarized by treating some representative cases. Basically, these multiple errors are of two types: either one operand and the operator are incorrect, or both operands are incorrect. In the latter case the operator may or may not be correct, but the effects are the same. In every possible error of the first type coincidental correctness can occur only when a specific relationship exists between the new value of the incorrect operand and the correct values of both operands. For example, consider the case in which \(< + X Y >\) is replaced by \(< * X' Y >\). The value of \(T(1)\) will be unaffected only when \(X + Y = X'Y\), which is equivalent to the condition \(X' = X/Y + 1\). So, this type of error produces immediate coincidence only in the event that the incorrect operand equals this specific function of both correct operands. There is only a single value for \(X\), other than the correct value, which causes the correct value to be computed for \(T(1)\).

The key characteristic of the second type of error is the fact that both operands are incorrect. For example, consider the case in which \(< + X Y >\) is replaced by \(< * X' Y' >\). The value of \(T(1)\) will be unaffected if \(X + Y = X'Y'\), or equivalently \(X' = (X + Y)/Y'\). Similarly, if the operator is not changed, we get \(X' = X + Y - Y'\). In either case there is an infinite number of ways in which to change both operands without affecting the result, e.g., \(3+4 = 2+5 = 1+6 = 3+7\), etc.. However, for any change in one operand there is only a single incorrect value for the other operand which will cause the errors to cancel one another. In other words, the errors in the two operands have to be synchronized, and out of the total errors possible, this event is still very unlikely. Every possible type of multiple error has been enumerated and analyzed in this
fashion, and in each case a condition similar to the two described above must be satisfied in order for the value of \( T(i) \) to remain unchanged. So, as in the case of single errors, it is very unlikely for multiple errors to produce coincidental correctness. In addition the assumption made for single errors is no longer necessary, since this analysis has shown that multiple errors are very unlikely to cancel each other. The only negative aspect of this multiple error analysis is that we cannot monitor the execution of test cases to detect the conditions required for coincidental correctness. Each of these conditions involves both the correct and incorrect values of one or both operands, and of course if an operand is incorrect, we do not know what the correct value is and therefore cannot determine if the condition is satisfied.

We have seen that the data flow structure of a transformation is very important in determining the propagation effects of an error. We can use this model to draw some general conclusions relating the likelihood of coincidental correctness to properties of the data flow structure of a path. First we consider a sequence of triples generated for a single assignment statement. For example consider the following two assignment statements and the associated sequences of triples.

\[
A = B + (C-D)/(E+1) - 3; \quad X = 2*Y + 3;
\]

\[
\begin{align*}
T(1) & \leftarrow \leftarrow - C \quad \L \quad T(1) & \leftarrow \leftarrow \ast 2 \quad Y \\
T(2) & \leftarrow \leftarrow + E \quad I \quad X & \leftarrow \leftarrow + T(1) \quad 3 \\
T(3) & \leftarrow \leftarrow \div T(1) \quad T(2) \\
T(4) & \leftarrow \leftarrow + B \quad T(3) \\
A & \leftarrow \leftarrow - T(4) \quad 3
\end{align*}
\]

We see that the result of each triple is used as an operand in only one subsequent triple, and this will always be true assuming that no attempt is made to rearrange the sequence of triples for efficiency and optimization. So, the data flow within an assignment statement is very simple, and each temporary variable directly affects only one other triple. An error in some triple for \( T(i) \) can produce immediate coincidence, or it can propagate as an operand error in the subsequent triple for \( T(j) \) which uses \( T(i) \). Coincidence can occur in the triple for \( T(j) \), or again it can propagate to the \( T(k) \) which use \( T(j) \). Since the result of each triple is only used in one subsequent triple, there can only be a single line of error propagation, and it can be halted at any triple along this single sequence. So, the likelihood of coincidental correctness within a single assignment
statement depends on the length of the sequence of triples. For example, for the short assignment above on the right, an error affecting \( T(1) \) can only be masked in the triple computing variable \( X \). However, for the longer sequence on the left, propagated coincidence for an error in \( T(1) \) occurs if immediate coincidence affects any one of the triples for \( T(3), T(4), \) or \( A \). So, we can make the general observation that if an error exists in an assignment statement, the likelihood of the assigned value being coincidentally correct depends on the average length of the data flow paths from each triple to the triple for the final variable assignment. We can draw this conclusion because of the sequential nature of the data flow within an assignment statement, and of course the likelihood of an error existing at all depends on the length and complexity of the assignment statement.

More interesting observations can be made about the data flow between assignment statements since the variable assigned a value can be used in any number or subsequent assignment statements. We define the usage index of a program variable for a particular path as the total number of occurrences of the variable on the right hand sides of all assignment statements on the path, where each occurrence in an assignment statement using the variable more than once is counted. As described previously, an error in the assignment statement for \( V(i) \) can produce immediate coincidence or can cause an error which propagates to subsequent assignment statements. If the error is not masked in one of these occurrences, it propagates even further to each use made of that variable. Therefore, the effects of the original error can become widespread very quickly, thus greatly reducing the likelihood of coincidental correctness by the time the output variables are finally calculated.

The extent of error propagation depends on the usage index of the variable involved. If the incorrect variable has a high usage index, the error obviously propagates very quickly, and its effects spread throughout the data flow graph. In addition, if the variable indirectly affects many other variables, the error propagation spreads even faster after the variables which are directly dependent are affected. So, it appears that the likelihood of coincidental correctness is higher for a path whose data flow structure is basically sequential with a lesser degree of data dependence between assignments. Conversely, a path whose data flow is very complicated and which contains assignment statements with high usage indexes is less likely to produce coincidentally correct results.
The size of the output data space is an important factor affecting the likelihood of obtaining coincidentally correct output values. For example, if the output consists of only a single bit, a transformation selected at random can be expected to produce correct output half the time on the average. The output space is \(2^M\)-dimensional, where \(M\) is the number of output variables produced. Also, the number of different values each output variable can have depends on the size and type of data representation used. A transformation is incorrect regardless of how many of the \(M\) output variables are actually computed incorrectly. Therefore, the number of values computed is not the most important factor, and in this discussion we focus on the effect of the precision of the data representation used. We assume that the same data representation is used for all variables, and we will show that the higher the precision of the representation the less likely is coincidental correctness.

We have formulated necessary conditions for coincidental correctness to occur in a triple, and we will show how a low precision data representation tends to increase the likelihood that these conditions will be satisfied. For example, an error in which the triple \(< + X Y >\) is changed to \(< * X Y >\) will be masked when \(X = Y/(Y-1)\). Using a 32-bit data representation and assuming that all values of \(X\) and \(Y\) are equally likely, we have a total of \((2)^{30}\) possible \((X,Y)\) pairs, and for each value of \(Y\), there is obviously only one value of \(X\) which satisfies the condition \(X = Y/(Y-1)\). So we can expect the condition to be satisfied on the order of once in every four billion executions of the triple. The likelihood of coincidental correctness in this triple doubles every time we reduce the size of the representation by one bit, and for a very small representation of four bits, we expect coincidental correctness once in every 16 executions of the triple. A similar analysis can be done for the condition formulated for each type of change to the triple, and similar results would be obtained. So, in general, coincidental correctness seems to be less likely when the data representation allows a wider range of values to be used. This result suggests the possibility that coincidental correctness can be detected if we can somehow expand the size of the output space, essentially increasing its precision.

Our basic strategy is to expand the output space by adding the values calculated by intermediate operations to the set of output variables. To interpret this extra information, we assume that the "oracle" can determine unequivocally whether all these intermediate results are
correct. Adding extra information such as intermediate values to the output is a standard technique used in debugging computer programs. In debugging, the extra information is used to identify the source of the error, but in trying to detect coincidental correctness it provides details of how the output variables are computed. We refer to the different types of information which can be appended to the output of the program as traces, and we will define a series of them, each of which adds more information and consequently improves our ability to detect coincidental correctness.

Of course the basic information which always must be produced is the set of output values for the path. Obviously, by definition, coincidental correctness can never be detected using only these basic output values, and we call this an "O-level trace". We can extend the output to include the values assigned to each program variable V(i), giving us a "V-level trace". If an error in the assignment statement for V(i) produces propagated coincidence rather than immediate coincidence, it will be detected at the V-level because of the incorrect value of the program variable V(i). However, immediate coincidence occurs at the assignment statement for V(i), and the coincidence exists below the V-level and would still go undetected. An assignment statement consists of triples, each producing some temporary value T(i). Therefore, we next define a "T-level trace" by including the values calculated for all these temporary values in the output. Again, if an error in some triple affects the result of that triple, but its effects are masked in some subsequent triple, the error will be detected at the T-level. However, the coincidence can occur at a level even below this detailed trace, since the value of T(i) may be correct even when there is an error in the triple calculating T(i). Therefore, we define the final full computation trace as one consisting of the values of the three components of each triple generated. This "F-level trace", where F signifies that the full amount of information possible is included, must be used in order to guarantee that the output values have in fact been computed correctly rather than by coincidence. However, since the user must interpret the testing results in practice, it will be very difficult to make use of this degree of detailed information.

We have defined a series of information levels which progressively increases our ability to detect coincidental correctness. Each level provides more detail about how the computation is performed and includes all values present in previous levels, e.g., each program variable is produced by some triple. One interesting conclusion is that we must
view the full details of every individual arithmetic operation in order to detect all instances of coincidental correctness. Of course, we make no claims as to the cost effectiveness of these traces, since the amount of extra work required by the user may be totally unacceptable. However, this discussion does provide further insight into the mechanics of the coincidental correctness phenomenon.

In summary, this analysis seems to indicate that coincidental correctness is unlikely in many practical situations, e.g., over the set of arithmetic operators (+,-,*,/). We have formulated a concise list of specific operand relationships, one of which must be true in order for single errors to cause coincidental correctness. The execution of each test case can be monitored for these conditions, further reducing the likelihood of coincidental correctness affecting our test data. In addition we have considered the relationship between coincidental correctness and the precision of the output variables, and this allowed us to formulate procedures to increase the precision of the output space. However, it was shown that this increased precision can preclude the possibility of coincidental correctness only when the complete details of a computation are verified. Finally, a data flow model allowed us to draw some general conclusions about data flow characteristics which would indicate that a path is more susceptible to coincidental correctness. Based on this analysis, we conclude that the theoretical problem of coincidental correctness does not significantly diminish the effectiveness of the domain testing strategy for most practical programs.

7.5 An Optimal Domain Testing Strategy

In developing the basic domain testing results, it was shown that the number of required test points could be reduced by using an extreme point to test many adjacent borders. In addition significant savings were shown to be possible by sharing the test points for a common border between adjacent domains. These observations lead to the conclusion that when the domains of an input space are tested one at a time, many more points than are really necessary might be generated. Therefore, in this section we address the problem of defining a minimal set of points required to test all borders in an input space. By taking a global view of the problem, we will be able to maximize the degree of test point sharing possible between domains, but
This optimal strategy may be so complex that the extra work it entails may not be worth the reduction in the number of test points. However, an optimal strategy is also important in evaluating the efficiency, i.e., number of extra test points, of any other variation of the basic domain testing strategy. Therefore, our goal is to define a minimal set of domain test points such that any domain error, which is detected by a complete set of domain points, is also detected by this minimal set.

In this discussion we treat a very simple case to allow a detailed treatment of the various aspects of the problem. We analyze the problem for a two-dimensional continuous input space, and we assume that both the given and the true borders are linear and are produced by simple inequalities. To insure that the test points are effective, we assume that adjacent domains do not compute the same function. In addition, as in previous discussions, we assume the existence of an oracle and that coincidental correctness does not occur for any test point. So, essentially we are considering the simplest case possible, as first treated in section 6.4. Finally, in order to clarify these results, we discuss predicates in a program consisting of a sequence of unnested IF constructs.

We need to define some concepts which will be important in describing an optimal testing strategy. As used previously, a border segment is that part of a domain boundary produced by a single predicate. In a program consisting of a sequence of unnested IF constructs, we can number the predicates \( P(I) \) through \( P(Q) \), where \( Q \) is the number of IF constructs in the program. Each predicate, \( P(I) \), can have from one to \( 2^{\ast}(i-1) \) interpretations, depending on how and where the program variables used in the predicate have been calculated. The upper limit is determined by the number of paths from the beginning of the program to the predicate in question, and the number of interpretations can vary over this entire range because some or all of the program variables may be computed in code segments common to many paths. (*)

The following simple program demonstrates these possibilities.

(*) A complete discussion of predicate interpretations is provided in section 3.5.
The first predicate, $P(1)$, can have only a single interpretation, since each variable used in it can be calculated in only one way. Predicate $P(2)$ has two interpretations, since even though variable $D$ is assigned the value $C+2$ on both paths, the value of $C$ depends on the branch followed in the first IF construct. In general, the third predicate of a program can have up to four interpretations, since there are four possible paths before the predicate is encountered. However, in this program, predicate $P(3)$ has only two interpretations since the computation of $C$ does not depend on which branch is taken in the second IF construct. However, if the third predicate is in error and should be $E \leq 0$, the correct predicate would have four interpretations, the maximum number.

A composite border consists of all border segments produced by a single interpretation of a predicate. A single predicate interpretation in general can affect many paths. For example, the first predicate in the program, $P(1)$, can have only one interpretation, but it affects every domain in the input space since it must be on every path. So, in general, a composite border spans many domains and consists of many border segments. In addition, a composite border is subdivided into two sets of overlapping border segments, each set corresponding to the domains on either side of the composite border. An example of this can be seen in figure 41, where the composite border $AB$ consists of border segments $AD$ and $DB$ on one side and $AF$ and $FB$ on the
In formulating the degree of test point sharing possible, it will be very important to determine exactly how a shift in one border segment implies that other related border segments must also be shifted. If a domain error is caused by an error in the predicate itself, we would expect all composite borders produced by the many interpretations of the predicate to be shifted. However, since a domain error can also be caused by an assignment statement error affecting a variable used in only some of the interpretations of the predicate, it is possible for a single composite border to shift while other composite borders produced by the same predicate remain unchanged. So it would seem that a composite border must shift as a whole since it is produced by a single predicate interpretation, but unfortunately this is not always the case.

A predicate can have anywhere from a single interpretation up to a different interpretation for every possible path defined from the beginning of the program to the point at which the predicate is encountered. However, if the predicate is incorrect, it is possible that the correct predicate has more or fewer interpretations than the given predicate. If it has fewer interpretations, correcting the error would combine some of the composite borders into a single composite border. This case would not pose a problem since by testing each composite border we would detect those that had shifted more than 6. However, if the correct predicate has more interpretations than the given predicate, some composite border should actually be divided into many composite borders. Since each of these true composite borders can shift independently of the others, we must test each section of a composite border which corresponds to a potentially different interpretation. So, we define an independent border as that part of a composite border which may correspond to a different predicate interpretation. Of course, a composite border may also be an independent border if its interpretation corresponds to only one of the possible paths defined up to the predicate. For example, the first predicate of every program can have at most one interpretation. Therefore, the single composite border produced by the first predicate must always be an independent border.

These concepts are important, and their differences seem to be very complicated and subtle. Therefore, the following short program will provide concrete examples of these ideas.
The input space structure for this program is diagrammed in figure 40. Border AB is the single composite border produced by the first predicate, I ≤ 3, and AB is also an independent border. The second predicate, K ≥ 2*I + J, has only a single interpretation since K is assigned the same value for both paths through the first IF construct, and CE is the single composite border produced. However, the predicate could potentially have two interpretations, one for each path through the first IF construct. Since border AB defines the regions for these two paths, CD and DE are the two independent borders for the second predicate. In general, independent borders are bounded by the intersections of the composite border with borders produced by previous predicates.

We can also use this example to show how an independent border can actually shift independently of the rest of the composite border. Consider the case in which the predicate in the second IF construct, K ≥ 2*I + J, is incorrect and should actually be J ≥ I. Since the correct predicate contains the variable L, it now has two interpretations, J ≥ I for the path through the THEN branch of the first predicate, and J ≥ -3 for the ELSE branch. Since J ≥ I is also the interpretation of the given predicate, K ≥ 2*I + J, only part of the composite border shifts. In figure 40, independent border DE is shifted and should actually be positioned as FG, shown in figure 41. If we test composite border CE in figure 40 as a single entity, the domain error may not be detected. In this example the error will be detected only if we choose the OFF point along segment DE of composite border CE. So, we must test each independent border with the full complement of three test points. Also we can now see how a composite border consists of two sets of overlapping border segments. In figure 41,
Figure 40: Composite Borders and Independent Borders

Figure 41: Shift of A Single Independent Border
composite border AB consists of border segments AF and FB for the domains on the open side and segments AD and DB for the domains on the closed side.

We can take full advantage of the fact that an entire independent border, spanning many border segments, shifts as a unit or is correct as a whole. In this strategy, the only requirement is that an ON-OFF-ON sequence be tested for each independent border. We can fully share test points between all the border segments on both the open and closed sides of an independent border. This is a significant reduction, but we may be able to do better, since we have complete freedom in choosing the three test points for each independent border. Of course, the UFF point must lie between the ON points and be within the region bounded by the extensions of the adjacent borders.

If an independent border consists of two or more border segments on the open side, some other border must intersect it on this side. In this case we may be able to use the UFF point for this border as an ON point for the intersecting border. We can also use the same type of sharing on the closed side of the independent border. If there is a border intersecting on the closed side, we can use the intersection point as an ON test point for both borders. So, the optimal strategy allows three types of sharing: a single set of three test points is shared by all border segments defined on both sides of an independent border, under certain conditions the OFF test point can be chosen so that it serves as an ON test point of another independent border intersecting on the open side, and in certain cases one or both ON points can also serve as ON points for other independent borders intersecting on the closed side. Of course to share all three points the intersections on either side would have to be positioned so that the three points form an ON-OFF-ON sequence for the independent border in question.

In figure 42 we reproduce the simple input space of figure 41, and a set of test points defined by the optimal strategy is demonstrated. We see that independent border AB is intersected by independent border CD on the closed side, and the intersection point at D is used as an ON test point for both. Similarly, the endpoint of FG near point F serves as an ON point for FG and as an UFF point for AB. Since both CD and FG are not intersected by any other borders, no further sharing of their test points is possible.

With the set of assumptions stated at the beginning of this section, we define an optimal set of domain test points as the smallest set containing points satisfying the
Figure 42: An Optimal Set of Test Points
The ON-UFF-ON sequence requirements for each independent border in the input space structure. This is the minimum set of points, since if any point is eliminated from the set, at least one independent border would not be tested with an ON-UFF-ON sequence. By definition, this independent border can shift without affecting any other independent border. Therefore, without the required ON-UFF-ON sequence, certain shifts of this independent border would not be detected.

With the assumption that at most two borders intersect in a single point, we can now analyze the best and worst cases to formulate bounds on the number of test points required by the optimal strategy. The best case is when each independent border has two closed side intersections which bracket an open side intersection, since we would be able to utilize every test point for two purposes. However, we know that this lower bound is unattainable since the independent borders produced by the last predicate in the program will not be intersected, other than at the endpoints which intersect existing borders generated by previous predicates. Therefore, since we cannot possibly share the UFF point of these borders, the lower bound will be expressed using a strict inequality. In the worst case we cannot share any points of any independent borders, and for example, this will be the case for a simple program with only one predicate. So, the following bounds can be formulated, where IB is the total number of independent borders, and TP is the number of test points required.

$$(1.5 \times IB) < TP \leq (3 \times IB)$$

For example in figure 42 we have three independent borders and seven test points. The most we would need for three independent borders is nine points, and we saved two points because the test points at D and near F each serve two purposes.

The optimal strategy provides significant savings in comparison to the basic domain testing strategy. In figure 43, we again reproduce the input space diagram used in the previous example. In this diagram we indicate a complete set of domain test points, which are generated by testing each of the four domains separately. In addition, each extreme point which is enclosed by a box, indicates that the point would actually be generated twice. This clearly shows the significant savings possible with the optimal strategy, since the number of test points is reduced from 20 to 7 even in this simple example.
We must now consider the difficulty of actually generating a set of test points as defined by the optimal strategy. In defining the minimal set of test points we were able to view the full structure of the input space, but we must try to devise a reasonable strategy to actually select this set of points. In the general testing process one path is tested at a time, or equivalently one domain at a time. Obviously, this procedure cannot be used to select an optimal set of points since we cannot identify the intersection points which must be used as test points, or even the independent borders themselves. We could try to test each predicate in sequence. In this case we could identify each independent border since an independent border is bounded by the borders produced by previous predicates, but again we would not be able to identify intersection points with borders produced by subsequent predicates. So, the only way to generate a set of points for the optimal strategy is to assume that the complete structure of the input space is known. However, even with this assumption, it is very difficult to select a minimal set of points.

The difficulty of this problem can be demonstrated with a small example. The diagrams in figure 44 show a set of independent orders, AB, CD, EF, GH, IJ, KL, and MN, each of which must be tested with an ON-OFF-ON sequence. We see that AB is intersected by four other borders, and obviously we should use point C and a point on EF near F as shared test points. However, we must decide whether to use G or K as the other ON point for AB. In the upper diagram we have selected K as the shared test point and have indicated the rest of an optimal set. In the lower diagram point G has been selected, and we see that the total number of test points is 17 rather than the 16 required in the first case. The difference is that we can still share the two ON points for GH with other borders, IJ and MN, and thus point G is not needed, but the points for KL cannot be shared with other intersecting borders. So we see that in selecting points for AB we have to consider the structure of those borders intersecting AB and also the borders intersecting those borders. In general we may have to widen the scope of our analysis to the entire input space to make a decision for a particular independent border which will lead to the optimal set of test points. So, to select a truly optimal set of test points we may have to analyze the partitioning structure of the entire input space, and obviously this is a very complex and difficult problem.

In summary, we have studied an optimal set of test points, but it does not seem practical to try to achieve this optimum. However, this optimal strategy is a very important result even though we may not want to implement
Figure 44: Difficulty of Generating Optimal Test Points
it. First, it provides a theoretical minimum cost to which any other contemplated strategy can be compared, and this would be valuable in evaluating the efficiency of any variation of the domain testing strategy which might be devised. Second, this analysis has outlined many possible types of test point sharing, and each of these may be useful if implemented in some practical strategy. Lastly, these results provide significant insights into the relationship between a single error and the expected extent of its effects. In particular, we now see why a single error can affect many related domains, and this insight may prove useful in future research on the problem of test path selection.

7.6 Examples of Error Detection Using Domain Test Points

The domain testing strategy has been described and evaluated using somewhat complicated algebraic and geometric arguments. In this section we hope to complement those discussions by demonstrating how a specific set of domain test points for a short sample program actually detects specific examples of different types of programming errors. In discussing each error we will focus on a specific domain affected by the error, and a careful analysis of its effect on the domain will allow us to identify those domain test points which detect the error. These errors are designed to demonstrate various features of domain testing, and therefore neither the given nor the correct program is meant to represent any common, recognizable function. Each error could easily occur, and we feel that they are representative of the typical errors found in most programs.

The short example program reads two values, I and J, and produces a single output value M. Therefore, the input space is two-dimensional, and the following min-max constraints have been chosen so that the input space diagram would not be too large or complicated.

\[ I \geq -8 \quad I \leq 8 \quad J \geq -5 \quad J \leq 5 \]

In addition, since this is a two-dimensional space, we also test extreme points for the border segments produced by the min-max constraints in order to be able to detect as many missing path errors as possible.
The domain testing strategy for two-dimensional linear borders, as described in the previous chapter, requires that an ON-OFF-ON sequence of test points be selected for each border segment. In addition the OFF points must satisfy the constraints corresponding to the adjacent borders. Any set of points meeting these requirements is a valid set of domain test points, and the strategy does not specify a particular selection algorithm. In this section we have selected the test data by inspection so that they satisfy the domain testing requirements. Extreme points are used as ON test points, and we have tried to select OFF points near the middle of the border segments. Of course, a specific algorithm for selecting domain test points will be required in practice, and a possible implementation scheme for generating domain test data is described in chapter 8.

Finally, even though the input space is assumed to be continuous, the coordinates of each test point are specified to an accuracy of ±.2 in order to simplify the diagrams and discussions. Of course, in an actual implementation each OFF point would be chosen much closer to the border.

The sample program is listed below, and it consists of three simple IF constructs, the first two of which are inequalities and the last of which is an equality. Again this program is not meant to compute any common or recognizable function, and it has been designed to serve our specific purposes of exposition.

```plaintext
READ I,J

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

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

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

WRITE M
```
The input space structure is diagrammed in Figure 45, where the solid diagonal border across the entire space is produced by the first predicate, the dashed horizontal border and short vertical border at \( I=0 \) are produced by the second predicate, and the vertical equality border at \( I=5 \) corresponds to the third predicate. In addition, domain test points have been indicated for the two domains which we will need to discuss.

First we consider a transformation error and the possibility of coincidental correctness for path TEE, which follows the THEN branch in the first IF construct and the ELSE branch in the second and third IF constructs. The domain for this path is designated by TEE in Figure 45, and the following test points have been selected for this domain:

\((-8,2)\) \((-8,-5)\) \((-4,-5)\) \((-1,-2.2)\) \((3,2)\) \((-3,1.8)\)

This transformation error is caused by an incorrect assignment statement in the ELSE clause of the second IF construct, where \( L = J - 1 \) should actually be \( L = I - 2 \). Since \( L \) is not used in the third predicate, we know that a domain error cannot be produced by this incorrect assignment statement. The sequence of assignment statements below on the left, followed by the transformation computed for \( M \), represents the given transformation, and that on the right is the correct transformation, labeled \( M' \).

\[ K = I + J - 1; \]
\[ L = J - 1; \]
\[ M = L + 2*K - 1; \]
\[ M = 2*I + 3*J - 4; \]

\[ K = I + J - 1; \]
\[ L = I - 2; \]
\[ M = L + 2*K - 1; \]
\[ M = 3*I + 2*J - 5; \]

The incorrect transformation for \( M \) is applied to all points in the domain for path TEE, and the error should be detected by those test points which lie in this domain. Three of the selected test points lie in the domain being tested, and they are listed below, followed by the value computed by the given transformation and the value which would be computed by the correct transformation.

\((-3,1.8)\) \( M = -4.0 \) \( M' = -10.4 \)
\((-4,-5)\) \( M = -27 \) \( M' = -27 \)
\((-8,-5)\) \( M = -35 \) \( M' = -39 \)
Figure 45: Sample Input Space and Domain Test Points
So we see that we have a case of coincidental correctness at one of the points since the given and true transformations compute the same value for \((-4, -5)\). However, the error will be detected at \((-3, 1.8)\) and \((-8, -5)\) since the results produced by the two transformations differ. Even though we contrived this error to produce coincidental correctness, the error is still detected by some of the other test points which really have been selected to detect domain errors.

Path TEE can also be used to study examples of domain errors. We consider two example programming errors, one in which only the domain is defined incorrectly and another in which both the domain and the transformation are incorrect. In the first case, the predicate in the second IF statement, \(K \geq I + 1\), is incorrect and should be \(K \geq I + 2\). Since the domain error is caused by an incorrect predicate rather than an incorrect assignment statement, we know that the transformation is not affected. The given predicate is interpreted as \(J \geq 2\) since \(K = I + J - 1\), but the correct predicate would be interpreted as \(J \geq 3\). So the border has shifted from \(J = 3\) to \(J = 2\), and the correct input space structure is shown in figure 40. The three points, \((-8, 2)\), \((-3, 1.8)\), and \((3, 2)\) have been selected to test this border, and since the shift reduces the size of the domain, we expect both ON points, viz., \((-8, 2)\) and \((3, 2)\) to detect the error. These points follow path TEE, and \(M\) is computed as \(3*I + 2*J - 2\) on this path. However, these points should actually follow path TEE, and the transformation on this path is \(M' = 2*I + 3*J - 4\). The output values actually computed and those which should be computed for these two points are listed below.

\[
\begin{align*}
(-8, 2) & \quad M = -22 & M' = -14 \\
(3, 2) & \quad M = 11 & M' = 8
\end{align*}
\]

Obviously, the output produced for each of these test cases is not as expected, and the error will be detected. One final aspect of this error is that since the predicate itself is in error, all of its interpretations are affected. In fact, in figure 40 the other border produced by this predicate, the dashed vertical border at \(I = 0\), is also incorrect and should actually be positioned at \(I = 1\). Therefore, the error can also be detected by testing paths ETE and EEE, since their domain boundaries contain this border.

A programming error in the assignment statement \(K = I + J - 1\) in the THEN clause of the first IF construct produces a domain error in which the transformation for path TEE is also incorrect, since \(K\) is used both in the
Figure 46: Correct Sample Input Space for Domain Errors
calculation of \( M \) and in the second predicate. If this assignment statement is incorrect and should actually be \( K = I + J - 2 \), the second predicate, \( K \geq I + 1 \) is interpreted as \( J \geq 2 \), but it should actually be interpreted as \( J \geq 3 \). This is the same border shift as in the previous case, and again test points \((-3,2)\) and \((3,2)\) will detect the border shift. In addition, the given transformation for path TEE is \( M = 2*I + J*J - 4 \), but in this case the correct transformation is \( M' = 2*I + 3*J - 6 \). Again, since the incorrect transformation is applied to all points in the domain being tested, we expect those test points which lie in this domain to detect the error. These points and their corresponding output values are listed below.

\[
\begin{align*}
(-3, 1, 0) & \quad M = -4.0 \quad M' = -6.0 \\
(-4, -5) & \quad M = -27 \quad M' = -29 \\
(-8, -5) & \quad M = -35 \quad M' = -37
\end{align*}
\]

In this case coincidental correctness does not occur at any of the test points, and each of them will detect the error. The correct input space in this case is the same as in the previous example and is shown in figure 46. Therefore, the domain error would also be detected by testing paths ETE and EEE. In addition, the transformation error would also be detected by testing domains TTE and TTT, since the incorrect assignment statement is also used in the transformation for those paths. These examples clearly show that an error may affect many domains, since in general the incorrect statement is executed on many paths. Therefore, paths with common elements tend to be affected by the same errors, and this fact may prove useful in future research on the problem of test path selection.

We can also examine the effect of missing path errors on the domain for path TEE. Suppose that the THEN clause of the first IF construct is incorrect, and that under certain conditions we should compute the value of \( K \) using the simple assignment statement \( K=J \) rather than \( K = I + J - 1 \). In this example, the THEN clause should be replaced by the following code.

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

This is a missing path error in which the missing predicate
is an inequality. In figure 47 we show the input space with
the additional border corresponding to the missing
predicate. We see that the part of domain TEE to the right
of this new order, including the border itself, should
still be computed using the original assignment statement
\( K = I + J - 1 \), and therefore all test points in this
subdomain will be unaffected. However, the subdomain to the
left of the new order should use \( K = 3 \), and the correct
transformation is \( K' = J + 4 \). The test point \((-8, -5)\) is in
this subdomain and should detect the missing path error.
When this point is computed on the original path, the output
value produced is \( M = -35 \), but the correct output should be
\( M' = J + 4 = -1 \). Therefore, this point will detect the
missing path error, and we can see why the extreme points of
order segments corresponding to min-max constraints are
tested in a two-dimensional space. Of course, as described
previously, we cannot guarantee that this type of missing
path error will be detected in a general N-dimensional input
space, since the cost of testing every extreme point can
grow very rapidly.

A missing path error of reduced dimensionality occurs
if the missing predicate is of the form:

\[
\text{THEN IF } 2*J = -5*I - 40. 
\]

In this case the subdomain for the missing path is only the
new order, and we can see that none of our test points are
affected. In general, we would have to be extremely
fortunate for one of the selected domain test points to lie
on this missing order. Finally, the very worst case occurs
if the missing predicate is compound, such as:

\[
\text{THEN IF } (2*J = -5*I - 40) \text{ AND } (I = -7) 
\]

In this case the subdomain for the missing path is the
intersection of the hyperplanes defined by the two
equalities. Since this is a two-dimensional example, the
intersection is just the single point \((-7, -2.5)\), and again
our chances of having selected this point are essentially
zero. The subdomain for the missing path may also be as
small as a single point for higher dimensional input spaces
if the missing predicate consists of more than two
equalities. So, we can see the difficulty of detecting both
missing path errors of reduced dimensionality and those
corresponding to compound missing predicates. Finally, we
should note that the missing predicate affects all paths
following the THEN branch of the first IF construct.
However, in this example missing path errors are not
produced for those other domains since the missing predicate
is either redundant, dominated by the min-max constraint.
Figure 47: Correct Sample Input Space for Missing Path Error
The last errors to examine are shifts in equality and nonequality borders. Therefore, we now consider path ETT, since the equality predicate is not redundant for this path, and in Figure 45 we have indicated the set of test points selected for this domain. The two test points at the upper end of the domain, \((5,4)\) and \((5,3.8)\) constitute a set of one ON point and an OFF point to test the border defining the upper end of the domain. In this case the effective dimensionality of the domain, \(d-L\), is one since the input space is two-dimensional and the domain is constrained by one equality predicate. So the border at the upper end is a single point, and therefore only one ON point is required. The test point at the lower end, \((5,-5)\), is an extreme point produced by the \(\min-\max\) constraint \(J \geq -5\). The test points \((4,d,0)\) and \((5,2,0)\) are the two required OFF points, and \((5,0)\) is the extra ON point, which is needed to preclude all relational operator errors in the predicate. Consider a case in which this equality predicate, \(I = 5\), is incorrect and should actually be \(I = 5 - J\), as shown in Figure 48. In this case we see that the points at either end of the line segment, \((5,-5)\) and \((5,3.8)\) follow path ETT while they should follow ETI, but the other points are unaffected. So, these two points will be processed using the transformation \(M = 4*I + 3\), while they should be computed using \(M' = 5*I + 2\), and the following values will be calculated.

\[
\begin{align*}
(5,-5) & \quad M = 23 \\
(5,3.8) & \quad M = 23
\end{align*}
\]

\[
\begin{align*}
M' & = 27 \\
M' & = 27
\end{align*}
\]

Obviously, the output values are affected by the domain error, and both points detect the error. In addition we can clearly see that the equality border could not shift and still contain the test points lying on it.

The third predicate also produces a nonequality constraint in the domain for path ETI, and the error used in the previous example can also be analyzed as a nonequality error. In this domain the nonequality border can be tested with the same set of points as before, except that the point \((5,4)\) would not be needed in this case. Again, points \((5,-5)\) and \((5,3.8)\) serve as the two ON points for the nonequality border, \((4,d,0)\) and \((5,2,0)\) are the two OFF points, and \((5,0)\) is the extra ON point. This nonequality error is detected just as before since the same two points are in the wrong domain. This example clearly demonstrates the close relationship between equality and nonequality
Figure 48: Sample Errors for an Equality Predicate
errors. Furthermore, testing a domain constrained by an equality predicate may mean that the associated nonequality constraint need not be tested for the related domain. This is another example of possible directions for research on the problem of test path selection.

In summary, this section has provided specific examples of many of the types of errors discussed in previous chapters. The effectiveness of domain test points has been demonstrated for transformation errors, even with instances of coincidental correctness, domain errors for equality, inequality, and nonequality orders, domain errors for which the transformation is also incorrect, and a major class of missing path errors. In addition, the difficulty of detecting missing path errors of reduced dimensionality has been clearly demonstrated. These specific examples provide ample evidence of the power of domain testing and complement the detailed geometric and algebraic arguments presented earlier.
CHAPTER 8
AN IMPLEMENTATION SCHEME FOR LINEAR BORDERS

In this chapter we describe an implementation of the domain testing strategy using linear programming techniques. Section 1 documents our study of COBOL programs in which the importance of the linear case is demonstrated. A brief overview of the implementation plan is provided in section 2. The first stage of the plan transforms the predicates into a standard form and processes equality constraints, and this initial processing is described in section 3. In the next section we describe the use of linear programming techniques to generate test points for inequality borders, and this discussion addresses many of the central issues of the practical implementation of domain testing. Finally, in section 5 we discuss the generation of test points for equality and nonequality borders.

8.1 The Importance of the Linear Case

Efficient implementations of domain testing are possible for the linear case, but more powerful mathematical techniques are needed in order to apply the methodology to nonlinear borders. Therefore, we have tried to determine the relative importance or frequency of linear predicate interpretations in the type of programs typically found in a production environment. This section presents a summary of the study undertaken and draws some conclusions based on the data obtained.

Many studies of program profiles have been reported, but none of them were designed to directly evaluate the linearity of predicate interpretations. Knuth [KNUT71] has studied FORTRAN programs, Elshoff [ELSH75, ELSH76] PL/I programs, and Gordon and Salvadori [GORD75, SALVA75] have reported their findings for COBOL programs. A common finding in all these studies is that extremely short and simple arithmetic expressions predominate in both assignment
statements and predicates. (*) Also, it was found that the arithmetic operators which cause nonlinearities, such as multiplication and division, are used much less frequently than addition and subtraction. However, other than this observation, no data have been reported which can be used to determine the relative frequency of linear and nonlinear predicate interpretations.

A random sample of 50 COBOL programs were taken directly from production data processing applications for our study. In this static analysis each predicate appearing in a program is classified according to various characteristics, but no attempt has been made to determine its dynamic behavior, such as the number of paths it might affect or the number of times it might be encountered on a particular path. Each predicate is classified according to whether it is linear or nonlinear, and the number of input variables used in the predicate has also been recorded. In addition, we tabulated the number of predicates which are input-independent, since these predicates do not produce any input constraints. (**) The number of equality predicates is also reported since these predicates are very beneficial in reducing the number of test points required for a domain. These data are summarized in table 2.

The most important result is that only one predicate out of the 1225 tabulated in the study can possibly produce a nonlinear border. In addition, nearly three out of four predicates are equality constraints, and therefore the number of test points required for a domain will be substantially less than expected. Even though each of these equality predicates produces nonequality constraints for some paths, the expected reduction is still very significant. The predicates are also very simple since the bulk of them refer to only one input variable, and no predicate in this sample uses more than two input variables.

In conclusion, while this study by no means represents an exhaustive survey, we believe the sample is large enough to indicate that nonlinear predicate interpretations are rarely encountered in data processing applications. This seems to reflect the fact that very little numerical computation is done in this type of programming, and consequently there is little chance of a variable being

(*) A more detailed discussion of these findings is provided in section 2.1.

(**) Section 4.4 contains a detailed discussion of these input-independent predicates.
Table 2: Predicate Statistics for COBOL Programs

<table>
<thead>
<tr>
<th>Category</th>
<th>Total</th>
<th>Ave.</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Lines</td>
<td>12,628</td>
<td>253</td>
<td>31-1,287</td>
</tr>
<tr>
<td>Procedure Division Lines</td>
<td>8,139</td>
<td>163</td>
<td>13-822</td>
</tr>
<tr>
<td>Total Predicates</td>
<td>1,225</td>
<td>25</td>
<td>0-115</td>
</tr>
<tr>
<td>Linear Predicates</td>
<td>1,070</td>
<td>21</td>
<td>0-104</td>
</tr>
<tr>
<td>Nonlinear Predicates</td>
<td>1</td>
<td>0.02</td>
<td>0-1</td>
</tr>
<tr>
<td>Input-Independent Predicates</td>
<td>154</td>
<td>3</td>
<td>0-28</td>
</tr>
<tr>
<td>Predicates with 1 Variable</td>
<td>945</td>
<td>19</td>
<td>0-97</td>
</tr>
<tr>
<td>Predicates with 2 Variables</td>
<td>125</td>
<td>2.5</td>
<td>0-20</td>
</tr>
<tr>
<td>Equality Predicates</td>
<td>779</td>
<td>15.5</td>
<td>0-76</td>
</tr>
</tbody>
</table>
calculated as a nonlinear function of the input variables. Therefore, our linear domain testing results can be applied to this large class of programs and should prove useful in improving the reliability of these extremely important software systems.

8.2 Overview

In this section we describe the overall problem of domain test data generation and outline the goals of the process. A brief overview of the implementation scheme is presented, and each stage of the process is briefly described.

We assume that the path to be tested has already been selected in some way, and that a complete set of predicate interpretations has been produced using symbolic execution, as described in section 3.4. In addition we assume that all input-independent predicate interpretations are eliminated from the set of predicates during the symbolic execution of the path. Of course, the most important assumption is that each predicate interpretation is linear in terms of the input variables. Since we plan to use a standard linear programming algorithm, we assume that all variables are real-valued, and we also assume that subscripted variables are not used since in this presentation we do not want to address the problem of indeterminate array references.

The feasibility of the path selected for testing has not yet been ascertained. Consequently there may be no solution to the system of constraints, and this condition must be recognized by the test data generation process. In addition, some of the predicates may be redundant, and we must also be able to identify this condition. Also, since the predicates are generated and interpreted in the order in which they are encountered on the path, any required sorting of the predicates will have to be performed in the initial stage of the processing.

The eventual output of the test data generation process is a complete set of domain test points or an indication that the selected path is infeasible. For this discussion we assume that no other information is to be produced, but in an actual implementation an identification of redundant predicates and an identification of the points generated to test each predicate may prove to be useful. The generated test data must contain the necessary number, i.e., (N-L), of
ON points and a single OFF point for each nonredundant inequality order and the \((N-L+1)\) ON points and pair of OFF points for each nonredundant equality and nonequality predicate. As used previously, \(N\) is the dimensionality of the input space, which is determined by the number of input variables used by the program, and \(L\) is the number of independent or nonredundant equality constraints for the particular test path.

In this initial description we are not concerned with the efficiency of the implementation, and we make no claims of optimality in terms of the number of test points generated. Our goal in this chapter is to discuss some of the central issues which need to be addressed if the required set of domain test points is to be generated using linear programming techniques. An actual implementation of a working system is one area for future research, and questions of efficiency will be addressed at that time.

The overall scheme is divided into three distinct stages, each of which will be discussed in detail in a later section. The first stage performs all necessary preprocessing of the predicates in order to meet the format requirements of the standard linear programming techniques. In addition, at this time the set of equality constraints are used to eliminate as many input variables as possible from the total set of predicates, thus producing a smaller system to solve in later stages. The second stage generates the test points for the complete set of inequality predicates, and during this stage linear programming is used extensively. Finally, the third stage generates any additional test points needed for the equality predicates themselves and for all nonequality predicates. We must try to recognize redundant predicates as early as possible, since not only do we need not test these predicates but we cannot test them since no point in the domain can serve as an ON point for a redundant predicate. In addition, infeasibility must also be recognized as quickly as possible, since any effort spent on processing a set of predicates for an infeasible path is obviously wasted. Figure 49 provides a high level block diagram of these three stages in the overall test data generation scheme.

8.3 Initialization and Processing of Equality Constraints

The first stage of the process inputs the set of predicate interpretations encountered along the test path.
TRANSFORM PREDICATES INTO STANDARD FORM

AND

PROCESS EQUALITY CONSTRAINTS

GENERATE DOMAIN TEST POINTS FOR INEQUALITY PREDICATES USING LINEAR PROGRAMMING TECHNIQUES

GENERATE ADDITIONAL POINTS FOR TESTING EQUALITY AND NONEQUALITY PREDICATES

Figure 49: The Three Stages of Domain Test Data Generation
So, all preliminary initialization and preprocessing must be done at this time. In addition the set of equality constraints are used to eliminate as many input variables as possible from the total set of predicates, and we will be able to eliminate L of them, where L is the number of independent equality constraints.

Since the predicate interpretations are not in standard order or form, the first step in this stage of the processing is to separate them into three distinct sets: equalities, inequalities, and nonequalities. At the same time we will standardize their format by transposing all variable terms to the left hand side and combining all constant terms on the right hand side, e.g., \( 6 + 3X < 2Y + 4 \) would become \( 3X - 2Y < -2 \). Each of these sets of predicates will eventually be stored in matrix form, where each column in the matrix holds the coefficients of a variable or the right hand side constant term. However, before we store the inequalities we must transform them so that each is expressed using the relational operator \( \leq \). First, any predicates using \( > \) or \( \geq \) can be transformed into an equivalent form by multiplying both sides by \(-1\), e.g., \( 2X - Y \geq -3 \) is equivalent to \(-2X + Y \leq 3\). Then each predicate using the strict inequality \(<\) can be replaced by an equivalent predicate using \( \leq \) by subtracting some very small positive quantity, \( \epsilon \), from the right hand side, e.g., \( X < 3 \) would become \( X \leq 3 - \epsilon \) (*) So now we have three separate sets of predicates: equalities, nonequalities, and inequalities expressed as \( \leq \) constraints. At this time we can append the full set of min-max constraints to the set of inequalities. Again, each min-max constraint must conform to the standard form, e.g., the pair of constraints \( X \geq -10 \) and \( X \leq 10 \) would be stored as \(-X \leq 10 \) and \( X \leq 10 \). In addition, an indicator or flag must be stored with each min-max constraint since in later processing stages we must be able to differentiate between them and constraints generated by actual path predicates. The standard linear programming technique assumes that all variables are nonnegative, but in general the test data for a program may be positive or negative. At this time we must replace each variable by the difference of two nonnegative variables, e.g., \(-7 = 0 - 7\). This transformation satisfies the nonnegativity assumption, but we note that it is costly since the number of variables in the system is doubled.

(*) This is a standard procedure employed in test data generation using linear programming techniques. See, for example, Clarke [CLARL76], p. 221.
The next step is to use the set of equalities to eliminate input variables from all three sets of constraints. So, an equality is first solved for one of the input variables, e.g., $3X - Y = 2$ becomes $Y = 3X - 2$, and this variable is then replaced in all remaining constraints, including inequalities, nonequalities, and the other equalities. Of course we must also store the equation for each variable eliminated in this way in some other matrix, so that values for this input variable can be calculated in stages 2 and 3.

Each time we eliminate a variable it is possible that a redundant predicate or the existence of infeasibility will become apparent. For example, if we use the equation $X = Y + 1$ to eliminate the variable $X$ in the constraint $X - Y \leq -1$, we would get $Y + 1 - Y \leq -1$ or in standard form $0 \leq -2$. Since this is obviously an unsatisfiable condition, we know that the set of constraints is not solvable, and therefore the path is infeasible. Similarly, if the constraint had been $X - Y \leq 2$, it would reduce to $0 \leq 1$ after eliminating $X$, and in this case we can conclude that it is redundant.

We can formulate a simple set of conditions for each of the three types of constraints which will signal infeasibility and redundancy. The key condition is that after eliminating a variable, the constraint reduces to a relationship between constants. Since all constant terms are combined on the right hand side in the standard form and all left hand side variable terms are zero, a constraint will reduce to a relationship between zero and some constant term. Basically, if this relationship is true, the predicate is redundant, and if it is false, the path is infeasible. So, for equalities, if the right hand side is zero, the equality is redundant, and any other value indicates infeasibility. Conversely, for nonequalities, if the right hand side is zero, the path is infeasible, and any other value indicates redundancy. Finally, since each inequality uses $\leq$ as the operator, the path is infeasible when the right hand side is negative, and redundant otherwise. Of course, the test data generation process should be aborted if infeasibility were to be detected, and any constraint which is found to be redundant would be dropped from the set of predicates.

In summary, stage 1 inputs an unordered set of predicates and produces three things: a set of inequalities in the standard form required to apply linear programming, a set of nonequalities, also stored in a standard form, and a list of the values for the input variables which have been eliminated. Figure 50 shows the input and outputs of
Unordered Set of Linear Predicate Interpretations

PHASE 1

Set of Inequality Constraints in Standard Form
Set of Nonequality Constraints in Standard Form
Set of Values for Input Variables Eliminated

Figure 50: Schematic Diagram of Stage 1 Processing
stage 1. Finally, even though we check for infeasibility and redundancy in this stage, we cannot detect all occurrences of these conditions, and we must still check for it in later stages of the test data generation process.

8.4 Generating Test Points for Inequality Predicates

In this section we describe a method for generating both ON and OFF test points for the set of inequality constraints. This second stage of the process uses linear programming extensively, and many of the central issues of domain test data generation are addressed in this section. We first give a brief description of standard linear programming techniques, and then we discuss the use of this algorithm in generating the required set of ON test points. Our discussion will focus on the features of linear programming which are important for this purpose, and a full description of this technique is provided by Hadley [HADL62].

Linear programming techniques, such as the simplex method, are designed to find a solution which both satisfies a set of linear constraints and optimizes some linear objective function, commonly a cost or profit function. Viewed geometrically, it first finds an extreme point of the convex set of feasible solutions, and then it moves from extreme point to extreme point such that the value of the objective function improves with each move. In a finite number of moves an extreme point is reached such that no move can be made which improves the value of the objective function. This extreme point represents the optimal solution, and the process comes to a halt. Since we wish to find extreme points of the domain to use as ON points, each basic feasible solution, i.e., extreme point, found by the simplex method can serve as a test point. The basic difference between generating domain test data and finding an optimal solution is that rather than moving from one extreme point to another and stopping when the extreme point representing the optimal solution is reached, we must generate a set of extreme points providing a total of \((N-L)\) test points for each border. Therefore, we must modify this technique so that the extreme points visited are useful test points for those borders which do not already have the required number of ON points and so that in a finite number of steps the testing requirements for all borders are satisfied. Since we do not have to optimize any objective function, we can arbitrarily manipulate the objective
function to force the procedure to find extreme points which lie on the borders for which we do not already have \((N-L)\) test points.

The simplex method transforms the set of inequalities into a set of equality constraints by adding a slack variable to each inequality. For example, the constraint \(X + 2*Y \leq 6\) would be transformed into \(X + 2*Y + Z = 6\), where \(Z\) is a slack variable. Geometrically, the value of a slack variable for a particular solution point represents the distance of the point from the border corresponding to the slack variable. We should note that since a slack variable must have a nonnegative value, we need not replace it by the difference of two variables. Consider the following set of three inequalities.

\[
\begin{align*}
X + 2*Y &\leq 6 \\
3*X - Y &\leq 5 \\
-X - 3*Y &\leq -2
\end{align*}
\]

This system would be transformed into an equivalent system of equations as follows, where \(Z_1, Z_2,\) and \(Z_3\) are slack variables.

\[
\begin{align*}
X + 2*Y + Z_1 & = 6 \\
3*X - Y + Z_2 & = 5 \\
-X - 3*Y + Z_3 & = -2
\end{align*}
\]

A solution to this set of equations would be

\((X, Y, Z_1, Z_2, Z_3) = (2, 2, 0, 1, 6)\)

From the values of the slack variables we know that this solution point lies one unit from the border produced by the second inequality, six units from the border produced by the third inequality, and it lies right on the border produced by the first inequality. This geometrical interpretation of the slack variables is very important since it allows us to determine the borders on which each generated point will lie. An extreme point serves as an ON test point for each border whose corresponding slack variable has a value of zero. In this way we can keep track of which borders still require more ON test points.

Each extreme point represents a basic feasible solution to the set of constraints. In this case some of the variables are "basic variables" or are in the basis, and other variables are not in the basis. The important characteristic of a basic feasible solution is that only basic variables will have nonzero values. Therefore, if we can control which variables are in the basis, we can direct
the algorithm to find an extreme point which lies on a border lacking a sufficient number of ON test points. After finding an initial basic feasible solution, i.e., first extreme point, the simplex method generates the next basic feasible solution by replacing one of the variables in the basis with some nonbasic variable. This replacement produces a new basis which corresponds to a new basic feasible solution, and a new extreme point is thus generated. The variable to be inserted into the basis is chosen so that the objective function is improved and so that the new basis represents a feasible solution.

After a finite number of steps the simplex algorithm finds a basic feasible solution which optimizes the objective function. At each step it generates a new basic feasible solution which improves the value of the objective function. Basically, we will manipulate the objective function to force the algorithm to find an extreme point on a selected border, and we can also make use of any intermediate extreme points visited in finding the optimal solution. In generating a full set of domain test points we will provide the algorithm with a sequence of objective functions to optimize, and therefore we must check each extreme point to insure that it has not already been generated. In moving from one extreme point to another the simplex algorithm changes the basis by inserting a variable not currently in the basis and removing one of the basic variables. The variable to enter the basis is chosen to obtain the largest improvement in the objective function. Once this variable is selected, a specific variable must leave the basis in order to maintain a feasible solution. The central issue is how to manipulate the simplex algorithm so that specific variables can be forced into and driven out of the basis, since in this way we can force the algorithm to find an extreme point lying on a particular border.

We will maintain a vector of counters, each one corresponding to one of the constraints, and these counters will be used to record the number of extreme points already generated for each border. We initialize this vector to zero, and at this point we must be able to differentiate between constraints produced by path predicates and min-max constraints, since we do not need to test min-max borders. When an extreme point is generated, the counter for each border on which it lies will be incremented.

At the start of the process a basic feasible solution is not available, and the standard phase I of the simplex algorithm will be used to find this initial basic feasible solution. In this phase artificial variables are introduced into the system, and they are used to form an initial basis.
However, the values of these artificial variables must be driven to zero in order for a basic solution to be feasible. This is accomplished by using an artificial objective function in which the cost of each artificial variable is very high, while a zero cost is assigned to each real variable. During phase 1 the basis is changed so that all artificial variables are eventually driven out of the basis. This must occur since it is the only way the cost function can be reduced to its minimum value of zero.

At the end of phase 1 the feasibility of the system of constraints can be determined, and this is very important since it constitutes a practical solution to the infeasible path problem. Of course, if the path is found to be infeasible, the test data generation process will be terminated. In addition, if one or more artificial variables appear in the final basis at the zero level, some of the constraints may be redundant. Furthermore, by examining the values of the variables in the final basis, we can even determine the specific constraints which are redundant. (\*) So, at this point, the redundant predicates will be deleted from the system since they do not produce domain borders. Phase 1 also generates an initial basic feasible solution which constitutes our first extreme point and serves as the starting point for phase 2.

In phase 2 we select a border for which the required number of test points has not yet been generated, and a reasonable choice might be the border with the fewest test points. Our goal is to force the algorithm to find a point on the selected border, and we can do this by assigning a high cost to the slack variable for the selected border and zero cost to all other variables. After a finite number of steps, the simplex algorithm will find an extreme point on this border, since this is the only way to achieve the minimum cost of zero. In addition, each intermediate basic feasible solution may be used as a test point, if it lies on at least one border which still lacks a sufficient number of test points.

Once we have found one extreme point on the selected border, we may want to find other extreme points on the same border. Essentially, this means that we must find other basic feasible solutions, while preventing the corresponding slack variable from reentering the basis. At this point we will choose the variable to enter the basis without reference to any cost function. As long as the slack

\[ (*) \] For a more detailed discussion of the redundancy criteria, see HADLG62, p. 121-124.
variable corresponding to the selected border is not chosen as the variable to enter the basis, the next basic feasible solution must represent an extreme point lying on the same border. However, since duplications are possible, we must compare each point with the points already generated so that the same point is not counted twice. In particular, we must insure that the algorithm does not cycle within a small closed set of extreme points, thus never generating the total number of points required. If we keep track of the sequence of basic solutions already generated for a particular border, we can easily recognize the occurrence of a closed cycle. In this event we must choose a new variable to be inserted into the oasis so that the cycle is broken. Since each border must contain at least \((N-L)\) extreme points, we will always be able to break the cycle before another traversal of the full set of points is completed.

When a sufficient number of test points for this border has been generated, we again choose some other border lacking sufficient test points. An objective function is formulated which will cause the algorithm to find an extreme point on that border after a finite number of steps. This procedure can be repeated until a sufficient number of test points have been generated for each border.

We have described modifications to the simplex algorithm which allow us both to find an extreme point for any specific border and to generate a set of extreme points for that border, once the first one is found. Since each border must contain at least \((N-L)\) extreme points, possibly including intersection points with min-max borders, we know that the required set of test points exists. Therefore, the procedure is finite, although the total number of test points generated may be more than absolutely required.

Once the complete set of ON test points have been determined, the OFF point for each border can easily be generated. In calculating the OFF test point, we will use a matrix in which each row corresponds to one of the borders. During the generation of the set of ON test points, each row of this matrix will be used to store the sum of the generated ON points for the associated border. When each ON point is generated, we can determine which borders it lies on since the corresponding slack variable will have a value of zero. We can also determine if the point is a useful ON test point for a border, i.e., one of the first \((N-L)\), since we maintain a count of the number of ON test points already generated for each border. If the generated point is a useful test point for a particular border, we add its coordinates to the coordinates in the corresponding row of the OFF point matrix. Therefore, when all the ON points have been generated, each row of the OFF point matrix will
hold the sum of the coordinates of \((N-L)\) ON test points for the corresponding border. By dividing each element in a row by \((N-L)\), we can compute the coordinates of a convex combination of the ON points, and this point represents the projection of the OFF point on the border. Furthermore, since this convex combination is formed using equal weights for the ON points, the central point of the region defined by the ON points is selected. The OFF point is then found by projecting a small distance from this point toward the open side of the border in a direction normal to the border. The normal vector and the open side can easily be calculated from the equation defining the border. The distance from the border to the OFF point is \(d\) and depends on the data representation used. Of course, as a last step we must also make sure that the OFF point satisfies the constraints corresponding to all adjacent borders. This can easily be determined by checking it against all the domain constraints other than the one corresponding to the border for which the OFF point has been chosen.

One final point must be considered. Since each ON point must be part of the domain, it cannot lie on any of the nonequality borders. If it does, we cannot use it to test the inequality borders, and it would not be counted as a test point. However, such a point is still useful since it can serve as a test point for any of the nonequality borders it lies on. So, we would save it for the third stage, and therefore the work spent to generate it is not really wasted since one less point will be needed in stage 3.

In summary, we can utilize linear programming to generate the set of ON points for all inequality borders, and the OFF points can easily be found once the ON points are known. The key to the successful use of the simplex algorithm is the ability to generate an extreme point on a particular border by manipulating the cost function. In addition, the use of this technique allows us both to determine feasibility and to identify redundant constraints during the first phase of the simplex algorithm. We have not addressed the efficiency of this technique, but we believe that in an actual implementation the procedure can be tuned so that the number of extra points generated is not excessive.
8.5 Generating Test Points for Equalities and Nonequalities

The third and final stage of the domain test data generation process produces the additional points needed for the set of equality constraints and for each individual nonequality predicate. So we need to have access to the list of equality predicates and the list of nonequality predicates. In addition, since the test points for the nonequality predicates must also satisfy the inequality constraints, we also need to use these predicates in this stage.

To test the complete set of equality predicates we need \((N-L+1)\) independent ON points, two OFF points for each equality predicate, and an extra ON point between each pair of OFF points. However, the ON points generated for the inequality predicates can also serve to test the set of equalities. We have already generated \((N-L)\) ON points for each inequality border, and as long as there are at least two of these borders, we already have the required \((N-L+1)\) ON points. However, in the event that the domain is defined with one or less inequality predicates, we would have to generate some or all of these points, using linear programming just as we did in stage 2. Once we have these ON points we find a point which is a convex combination, and this acts as the extra ON point. Then we find the \(L\) pairs of OFF points by projecting a small distance from this point in both directions normal to each equality hyperplane. Since this single ON point is on the line segment connecting each pair of OFF points, it serves as the extra ON point for testing each equality. Therefore, in most cases we expect that we will only have to generate a total of \((2L+1)\) additional points to test the complete set of \(L\) equalities, and the generation of these points is very easily accomplished.

We must generate \((N-L+1)\) independent test points on each nonequality border, and in addition these points must also satisfy the set of inequalities. Essentially we have to solve the same linear programming system as in stage 2, except we apply each nonequality predicate individually as an equality constraint so that the points will lie both within the exterior boundary of the domain and on the nonequality border. So again we are using linear programming, and in fact we can utilize the system as it was formulated in stage 2. In addition any points generated in stage 2 which happen to lie on any of the nonequality borders can be utilized here. Once the set of ON points for a border are found, the OFF points can be generated as described above for equality predicates.
One final important observation is that a nonredundant nonequality predicate must also produce an equality constraint for some other path, as demonstrated in section 7.4. If the predicate is incorrect for any reason, both of these paths must be affected, and the error can be detected by testing either path. So, a nonequality border need not be tested if the related path on which it produces an equality constraint has been tested. Similarly, once a nonequality border is tested, the related path whose domain consists of this nonequality border need not be tested. So a nonequality border only has to be tested if we are not sure that the related path will be tested.

In summary, this chapter has considered some of the central issues of an implementation scheme for domain test data generation. We do not claim that all of the details of a working implementation have been addressed. The purpose of this chapter has been to provide enough details to indicate the close similarity of the domain test data generation problem to the standard linear programming problem. In addition we have considered those aspects of the problem which require modifications to the linear programming algorithm. In particular, we have discussed the use of the simplex algorithm, which is an efficient and widely used technique in the generation of test data, and we have described how it could be used in this problem. We can therefore conclude that the domain testing strategy is a practical methodology for improving software reliability and can be implemented for the linear case using existing mathematical techniques.
CHAPTER 9
CONCLUSIONS

9.1 Contributions

The basic goal of this research is to replace the intuitive principles by which current program testing procedures have been designed by a methodology based on a formal treatment of the program testing problem. By formulating the problem in basic geometric and algebraic terms, we have been able to develop an effective testing methodology whose capabilities can be precisely defined. In addition, since program testing cannot be completely effective, we have identified the limitations of the strategy. Unfortunately, in many cases these limitations have proven to be theoretical problems inherent to the general program testing process.

The main contribution of this research is the development of the domain testing strategy. Under certain well-defined conditions the methodology is guaranteed to detect domain errors in linear borders greater than some small magnitude $\theta$. Furthermore, the cost, as measured by the number of required test points, is reasonable and grows only linearly with both the dimensionality of the input space domain and the number of path predicates. Domain testing also detects transformation errors and missing path errors in many cases, but the detection of these two classes of errors cannot be guaranteed.

Domain testing has also been extended to classes of nonlinear borders, and we have shown that the methodology generalizes to any class of functions which can be described by a finite number of parameters. Finite testing procedures cannot be effective for the universal class of all nonlinear functions, and our analysis has determined the cause to be the complete generality and unpredictability of this universal class rather than some innate property of nonlinearity. Unfortunately, the limited capability of available nonlinear programming algorithms makes the implementation of domain testing in this case infeasible at
present. However, the strategy is theoretically effective for classes of nonlinear borders, and when more powerful mathematical techniques are developed, we expect our techniques to be applicable to numerical software, which frequently contains nonlinear predicates and borders.

Coincidental correctness is a theoretical limitation inherent to the program testing process, and we have proven that it prevents any reasonable finite testing procedure from being completely reliable. In particular, the possibility of coincidental correctness means that an exhaustive test of all points in an input domain is theoretically required to preclude the existence of transformation errors on a path. Within the class of all computable functions there exist functions which coincide at an arbitrarily high number of points, and therefore coincidental correctness cannot be completely ignored. However, an analysis of this phenomenon indicates that two randomly chosen functions coincide at a set of points of measure zero, and this limitation may not be very significant in many practical situations. In addition, we have outlined various modifications of the domain testing strategy which can be used to detect certain occurrences of coincidental correctness.

The class of missing path errors, particularly those of reduced dimensionality, has proven to be another theoretical limitation to the reliability of any finite testing strategy. Although our methodology cannot be guaranteed to detect all instances of this type of error, it can be extended to detect some well-defined subclasses of missing path errors. Unfortunately, the extra cost of this modification may be unacceptably high. Our analysis of missing path errors has shown that the cause of the difficulty is that the program does not contain any indication of the possible existence of a missing path error. Therefore, without additional information, a reasonable testing strategy for this class of errors cannot be formulated.

Our research has also provided some general insights about computer programs which may be useful in other areas of software research. The notion of a partitioned input space is not original with this research, but we have developed a detailed understanding of the properties of this partitioning structure. In addition, error classification schemes similar to that which we have adopted have been used by other researchers, but our detailed analysis has identified some important characteristics of these error types which have not been previously reported.
The domain testing strategy is a significant step in the solution of the program testing problem, even though its practical application is currently limited to linear predicates. The statistics which were gathered during our study of COBOL programs indicate that a large portion of predicate interpretations are linear for this major class of practical programs. Therefore, our domain testing results for the linear case are applicable to the important class of programs generally known as data processing applications and should help to improve the reliability of this common type of software.

The domain testing strategy requires a reasonable number of test points for a single path, but the total cost may be unacceptable for a large program containing an excessive number of paths. Therefore, domain testing does not seem to be feasible for programs beyond a certain size or complexity. Many large software systems are currently designed as collections of small program modules rather than as large monolithic programs, and the success of these systems indicate that this design methodology will become more prevalent in the near future. So, although domain testing is limited to reasonably sized programs, we expect that it will be applicable even to these large modular systems.

The domain testing strategy utilizes many test points but provides little information beyond the indication of the existence of an error. Therefore, we believe that this technique is more appropriate for the validation of programs in the final stage of development than for the preliminary debugging of programs. Its eventual acceptance by practitioners will depend both on its ability to detect errors and on its ease of use. In this research we have analyzed the error detection capabilities of domain testing. However, experience with a working system is necessary to determine the best way to utilize this methodology, and in the next section we outline some promising directions for future work in this area.

In conclusion, programs have always been tested in one way or another, but a foundation of scientific principles has been lacking. To our knowledge domain testing is the first methodology which is both based on a formal treatment of the program testing problem and capable of efficient practical implementations. We hope that these results can be used both to improve the reliability of important classes of software and to provide a basis for further research in this area.
9.2 Future Research

Formal research on the problem of program testing has just recently begun, and many important issues in this area are currently being investigated. The experience and understanding gained in our work has allowed us to identify promising directions for future study of the program testing problem. In addition, this section also outlines some long term goals for general software reliability research.

The cost of the domain testing strategy applied to a single path is very reasonable in terms of the number of test points required, and it grows only linearly with both the dimensionality of the input space domain and the number of predicates on the path. However, for a large program the number of possible paths can be very high, and therefore the total cost of testing all of these paths may be unacceptable. In particular, this may occur for large programs with complicated control structures containing many iteration loops. In order to provide a flexible testing technique whose cost can be adjusted to fit the available resources, we plan to investigate various ways of reducing the number of required test points while maintaining as much testing effectiveness as possible. A "partial domain testing strategy" will be investigated in which only a subset of the domain test points are actually generated for each path. In this study, an ordering on the full set of domain test points will be determined so that the most effective points are selected first, and the resulting decrease in effectiveness will be quantified. For example, we might want to concentrate on testing those borders of the domain produced by more complicated predicates, since these predicates must have required more mental effort to understand and describe. Another possible approach is to test all the domain borders but to select fewer test points for each border than specified by the domain testing strategy. This modification will provide a flexible strategy which will maximize testing effectiveness across a wide range of cost levels.

The number of possible paths in a complex program may be too large for practical purposes. A "path selection strategy" must be developed which can be used to choose the best paths for testing. A figure of merit must be defined which reflects the value of a path as a candidate for testing, and this measure must consider both the benefits and the cost of testing each path. For example, a very long and complicated path containing many assignment statements and predicates tests many aspects of the program, but a larger number of test points are consequently required.
Therefore, the best candidate for testing might be a long path consisting of many assignment statements but few predicates. Also, in selecting the next path for testing we must consider now the set of paths already chosen affects our current selection. We have seen that a single error may affect many different paths, and the error can be detected by testing any one of these paths. We must investigate this relationship between paths so that the selected paths do not all test the same things while other aspects of the program remain untested. The overall goal of path selection is to define criteria such that the set of selected test paths is most representative of the entire set of program paths.

We have described a domain test data generation scheme for linear orders, but it has not yet been implemented. A working domain testing system for linear orders would provide needed practical experience, and it could be used as an experimental facility in any future research. In addition, an experimental system would provide an opportunity to study the human factor problems of programmers actually working with the system.

The domain testing strategy has been analyzed for a sample programming language defined with a restricted set of data types and operators. A natural extension is to generalize the technique to other data types, and in particular to identify any specific data type which might cause serious obstacles to effective testing. In this way we hope to be able to gain valuable experience which would eventually be useful in defining new programming languages in which programs would be easier to test and therefore more reliable.

Program testing cannot be completely effective, and therefore we need to develop a measure of testing effectiveness. This measure would provide an estimate of how completely the specific set of test data generated is expected to test the program. It could be based on how well the selected test paths represent the complete set of program paths and on how completely each path has been tested. A user would have some indication of how much confidence to place in the reliability of his program and could then make an informed decision on whether further testing were warranted.

We have assumed that an "oracle" exists which can always determine whether a specific test case has been computed correctly or not. In reality, the programmer himself must make this determination, and the time spent examining and analyzing these test cases is a major factor in the high cost of software development. One possible
avenue for future research would be to automate this process by using some form of input-output specification. If the user provides a formal description of the expected results, the correctness of each test case can be decided automatically by determining whether the output specification is satisfied. This would reduce the cost of testing tremendously, and these new testing techniques would gain acceptance more quickly since the tedious task of verifying test data would be eliminated. In addition, any extra information supplied by the user might be useful in specifying special processing requirements which would indicate the existence of a possible missing path error.

The long term goal of research in the area of software quality assurance is a set of tools which will help us to produce more reliable software, and the study of one of these techniques does not preclude the use of any other. We expect these methodologies to be complementary rather than competitive, and a hybrid system utilizing a variety of approaches should be more successful than any technique implemented separately. One long term goal of our research is the integration of the program testing methodology with the more theoretical approach of program verification, and it is hoped that each technique will overcome some of the obstacles encountered by the other. For example, the information generated during program verification, e.g., input-output assertions and loop invariant conditions, might prove useful in identifying potential missing path errors. Also, the use of specific test data has already been proposed as an aid for guiding a program verifier in producing a correctness proof.
LIST OF REFERENCES


