A Domain Strategy for Computer Program Testing

LEE J. WHITE, MEMBER, IEEE, AND EDWARD I. COHEN

Abstract—This paper presents a testing strategy designed to detect errors in the control flow of a computer program, and the conditions under which this strategy is reliable are given and characterized. The control flow statements in a computer program partition the input space into a set of mutually exclusive *domains*, each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. The testing strategy generates test points to examine the boundaries of a domain to detect whether a domain error has occurred, as either one or more of these boundaries will have shifted or else the corresponding predicate relational operator has changed. If test points can be chosen within ϵ of each boundary, under the appropriate assumptions, the strategy is shown to be reliable in detecting domain errors of magnitude greater than ϵ. Moreover, the number of test points required to test each domain grows only linearly with both the dimensionality of the input space and the number of predicates along the path being tested.

Index Terms—Control structure, domain errors, path-oriented testing, software reliability, software testing, test data generation.

I. Introduction

THE purpose of this paper is to present a methodology for the automatic selection of test data for computer program testing. Computer programs contain two types of errors which have been identified as computation errors and domain errors by Howden [9]. A *domain error* occurs when a specific input follows the wrong path due to an error in the control flow of the program. A path contains a *computation error* when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. The proposed testing strategy has been designed to detect domain errors, and the conditions under which this strategy is reliable are given and characterized. A byproduct of this domain strategy is a partial ability to detect computation errors. This study and proposed methodology are described in greater detail in White *et al.* [12], [13] and in Cohen [3], [4].

There are limitations inherent in any path-oriented testing strategy, and these also constrain the proposed domain strategy. One such limitation might be termed *coincidental correctness*, which can occur when a specific test point follows an incorrect path, and yet the output variables coincidentally are the same as if that test point were to follow the correct path. This test point would then be of no assistance in the detection of the domain error which caused the control flow to change. Note that the problem of coincidental correctness also complicates testing for computation errors; no path-oriented test generation strategy can circumvent this problem.

Another inherent testing limitation has been previously identified by Howden [9] as a *missing path error*, in which a required predicate does not appear in the given program to be tested. Especially if this predicate were an equality, Howden has indicated that no path-oriented testing strategy could systematically determine that such a predicate should be present.

An important assumption in our work is that the user or an "oracle" is available who can decide unequivocally if the output is correct for the specific input processed. The oracle decides only if the output values are correct, and not whether they are computed correctly. If they are incorrect, the oracle does not provide any information about the error and does not give the correct output values.

In the next section, preliminary concepts are defined and discussed. Some assumptions must be made concerning the language in which the given computer program is written, and the ramifications of certain language constructs are explored. The important concepts of program path and path predicates, together with domains, are defined and characterized. The case of linear predicates is given particular emphasis since, in that situation, the domains assume the simple form of convex polyhedra in the input space.

II. Programming Language Assumptions

In order to investigate domain errors, we need to consider the language in which programs will be written. The control structures should be simple and concise, and should resemble those available in most procedure-oriented languages; examples will be stated in an Algol- or Pascal-like language. For simplicity, we assume a single real-valued data type, and this is converted to integer values for use as DO-loop indices.

A number of programming language features are assumed not to occur in the programs we are to analyze for domain errors. The first feature is that of arrays; despite the fact that arrays commonly occur in programs, a predicate which refers to an element of an input array can cause major complications (Ramamoorthy [11]). A second class of language features which will be temporarily excluded in our analysis is that of subroutines and functions. Side effects and parameter passing

Manuscript received February 25, 1979; revised July 24, 1979. This work was supported in part by the Air Force Office of Scientific Research under Grant 77-3416. Portions of this paper previously appeared in the Infotech State-of-the-Art Report "Software Testing," Infotech International Ltd., Maidenhead, England, 1978.

L. J. White is with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210.

E. I. Cohen is with the Poughkeepsie Programming Center, IBM Corporation, Poughkeepsie, NY.

can pose special problems for the computation of both control flow and environment. These two classes of language features are admittedly very important, and further research is needed to determine whether these features pose any fundamental limitations to the domain testing strategy.

Since input/output processing is so closely linked to a machine or compiler environment, we will assume that all I/O errors have previously been eliminated. Thus, only the most elementary I/O capabilities are provided; input is provided by a simple READ statement, and output is accomplished with a simple WRITE statement. 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; all other variables are called *program variables.*

The general predicate form used for control flow 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 ($<, >, =, \leqslant, \geqslant, \neq$). If a predicate consists of two or more relational expressions with Boolean operators, then it is a *compound predicate.* A *simple predicate* consists of just a single relational expression.

III. Predicate Interpretations

Every branch point of a computer program is associated with a predicate which evaluates to true or false, and its value determines which outcome of the branch will be followed. The *path condition* is the compound condition which must be satisfied by the input data point in order that the control path be executed. It is the conjunction of the individual predicate conditions which are generated at each branch point along the control path. Not all the control paths that exist syntactically within the program are executable. If input data exist which satisfy the path condition, the control path is also an *execution path* and can be used in testing the program. If the path condition is not satisfied by any input value, the path is said to be *infeasible*, and it is of no interest in testing the program.

A simple predicate is said to be *linear* in variables $V_1, V_2, \cdots, V_n$ if it is of the form

$$A_1 V_1 + A_2 V_2 + \cdots + A_n V_n \text{ ROP } K$$

where K and the A_i are constants and ROP represents one of the relational operators ($<, >, =, \leqslant, \geqslant, \neq$). A compound predicate is linear when each of its component simple predicates is linear.

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 *predicate interpretation.* A particular interpretation is equivalent to the original predicate in that input variable values satisfying the interpretation will lead to the computation of program variables which also satisfy the original 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. The following program segment provides example predicates and interpretations.

```
READ A, B;
IF A > B
   THEN C = B + 1;
   ELSE C = B - 1;
D = 2 * A + B;
IF C ≤ 0
   THEN E = 0;
   ELSE
         E = 0;
         DO I = 1, B;
            E = E + 2 * I;
         END;
IF D = 2
   THEN F = E + A;
   ELSE F = E - A;
WRITE F;
```

In the first predicate, $A > B$, both A and B are input variables, so there is only one interpretation. The second predicate, $C \leqslant 0$, 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 $B + 1 \leqslant 0$ or, equivalently, $B \leqslant -1$. When the ELSE $C = B - 1$ branch is taken, the interpretation is $B - 1 \leqslant 0$ or, equivalently, $B \leqslant 1$. Within the second IF-THEN-ELSE clause, a nested DO loop appears. The DO loop is executed:

no times if $B < 1$
once if $1 \leqslant B < 2$
twice if $2 \leqslant B < 3$
etc.

Thus, the selection of a path will require a specification of the number of times that the DO loop is executed, and a corresponding predicate is applied which selects those input points which will follow that particular path. Even though the third predicate, $D = 2$, appears on multiple paths, it only has one interpretation, $2*A + B = 2$, since D is assigned the value $2*A + B$ in the same statement in each path.

IV. Importance of Linear Predicates

The domain testing strategy becomes particularly attractive from a practical point of view if the predicates are assumed to be linear in input variables. It might seem to be an undue limitation to require that predicate interpretations be linear for the proposed strategy. In fact, however, as the following discussion shows, this represents no real limitation for many important applications.

A number of authors have provided data to show that simple programming language constructs are used more often than complex constructs. Knuth [10] studied a random sample of Fortran programs and found that 86 percent of all assignment statements were of the forms

TABLE I
PREDICATE STATISTICS FOR 50 COBOL PROGRAMS

	Total	Avg.	Range
Total Lines	12 628	253	31-1287
Procedure Division Lines	8139	163	13-822
Total Predicates	1225	25	0-115
Linear Predicates	1070	21	0-104
Nonlinear Predicates	1	0.02	0-1
Input-Independent Predicates	154	3	0-28
Predicates with 1 Variable	945	19	0-97
Predicates with 2 Variables	125	2.5	0-20
Equality Predicates	779	15.5	0-76

$$V_1 = V_2,$$
$$V_1 = V_2 + V_3,$$

or

$$V_1 = V_2 - V_3.$$

Also, 70 percent of all DO loops in the programs contained fewer than four statements. Knuth also conducted extensive dynamic tests, and found that less than 4 percent of a program accounts for more than half its running time. Elshoff [5], [6] studied 120 production PL/I programs and showed similar results, including the fact that 97 percent of all arithmetic operators are + or -, and 98 percent of all expressions contain fewer than two operators.

An experiment of particular relevance to the present context is reported in Cohen [4] using typical data processing programs since program functions and programming practice tend to be reasonably uniform in this area. A random sample of 50 Cobol programs was taken directly from production data processing applications for this study. In this static analysis, 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. It should be noted that linear predicates do not always correspond to linear predicate interpretations, for this depends upon the dynamic assignment of program variables used in these predicates; however, for these programs, the predicate interpretations were also linear in nearly every instance. The number of input-independent predicates were also tabulated 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 I.

The most important result is that only one predicate out of the 1225 tabulated in the study is nonlinear. The predicates are also very simple since most 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. It is clear that any testing strategy restricted to linear predicates is still viable in many areas of programming practice.

V. INPUT SPACE STRUCTURE

An *input space domain* is defined as a set of input data points satisfying a path condition, which consists of the conjunction of predicate interpretations along the path as defined in Section III. For simplicity in this discussion, each of these predicates is assumed to be simple. The input space is partitioned into a set of domains. Each domain corresponds to a particular execution path in the program and consists of the input data points which cause the path to be executed.

The boundary of each domain is determined by the predicate interpretations in the path condition and consists of *border segments*, where each border segment is the section of the boundary determined by a single simple predicate in the path condition. Each border segment can be open or closed, depending on the relational operator in the predicate. A *closed border segment* is actually part of the domain and is formed by predicates with $\leqslant$, $\geqslant$, or = operators. An *open border segment* forms part of the domain boundary, but does not constitute part of the domain, and is formed by $<$, $>$, and $\neq$ predicates.

The total number of predicates on the path is only an upper bound on the number of border segments in the domain boundary since certain predicates in the path condition may not actually produce border segments. An *input-independent predicate interpretation* is one which reduces to a relation between constants, and since it is either true or false regardless of the input values, it does not further constrain the domain. This case arises when a predicate is required for one or more paths, and yet lies along other paths for which it reduces to a tautology and is input-independent. *A redundant predicate* interpretation is one which is superceded by the other predicate interpretations, i.e., the domain can be defined by a strict subset of the predicate interpretations for that path. In general, the determination of which predicate interpretations are redundant is a difficult problem.

The general form of a simple linear predicate interpretation is

$$A_1X_1 + A_2X_2 + \cdots + A_NX_N \text{ ROP } K$$

where ROP is the relational operator, X_i are input variables, and A_i, K are constants. However, the border segment which any of these predicates defines is a section of the surface defined by the equality

$$A_1X_1 + A_2X_2 + \cdots + A_NX_N = K$$

since this is the limiting condition for the points satisfying the predicate. In an N-dimensional space, this linear equality defines a hyperplane which is the N-dimensional generalization of a plane.

Consider a path condition composed of a conjunction of simple predicates. These predicates can be of three basic types: equalities (=), inequalities ($<$, $>$, $\leqslant$, $\geqslant$), and nonequalities ($\neq$). The use of each of the three types results in a markedly different effect on the domain boundary. Each equality constrains the domain to lie in a particular hyperplane, thus reducing the dimensionality of the domain by one. The set of inequality constraints then defines a region

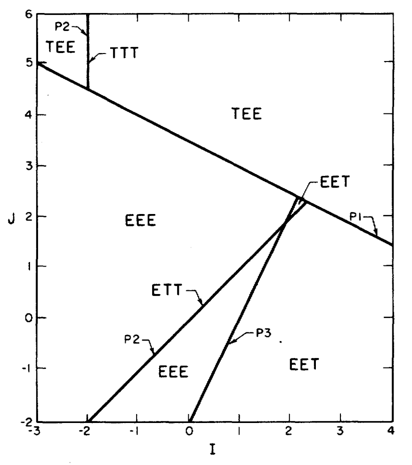


Fig. 1. Input space partitioning structure.

within the lower dimensional space defined by the equality predicates.

The nonequality linear constraints define hyperplanes which are not part of the domain, giving rise to open border segments as mentioned earlier. Observe that the constraint $A \neq B$ is equivalent to the compound predicate $(A < B)$ 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.

The following example should clarify the concepts discussed above.

```
      READ I, J;
      C = I + 2*J - 1;
(P1)  IF C > 6
         THEN D = C - I;
         ELSE D = C + I - J + 2;
(P2)  IF D = C + 2
         THEN E = I;
         ELSE E = 3;
(P3)  IF E ≤ D - 2*J
         THEN F = I;
         ELSE F = J;
         WRITE F;
```

Fig. 1 shows the corresponding input space partitioning structure for this program. The input space is in terms of inputs I and J, and is arbitrarily constrained by the following min-max conditions:

$$-3 \leq I \leq 4, \quad -2 \leq J \leq 6.$$

Each border in Fig. 1 is labeled with the corresponding predicate, and each domain is labeled with the corresponding path. The path notation is based upon which branch (THEN T or ELSE E) is taken in each of the three IF constructs, P1, P2, and P3. A three-character code then uniquely specifies the path, e.g., TEE.

The first predicate P1, $C > 6$, will be interpreted as $I + 2*J > 7$ since $C = I + 2*J - 1$. This single interpretation P1 is seen in Fig. 1 as a single continuous border segment across the entire input space. The second predicate P2 demonstrates the effects of both equality and nonequality predicates. Domains for paths through the THEN branch are constrained by the equality, and this reduction in dimensionality is seen in the fact that these domains consist of the points on the solid line segments ETT and TTT. Paths through the ELSE branch are specified by a nonequality predicate, and the corresponding domains consist of the two regions on either side of the solid line segments; in Fig. 1 these two regions are TEE and EEE, respectively, depending upon which branch is taken through predicate P1. Predicate P2 has two interpretations, depending upon the value assigned to D and produces two discontinuous border segments ETT and TTT.

The third predicate P3 might have four different interpretations, but only one border segment appears in the diagram. The other three interpretations do not produce borders since they are either redundant or correspond to infeasible paths. With three IF constructs, we have eight control paths, but the diagram contains only five domains since three of the paths are infeasible. Also, many of these domains have fewer than three border segments because of redundant and predicate interpretations. From this example, we can conclude that the input space partitioning structure of a program with many predicates and a larger dimensional input space can be extremely complicated.

The foregoing definitions and example allow us to characterize more precisely domains which correspond to simple linear predicate interpretations. For a formal statement of the characterization, we need the following definitions. A set is *convex* if for any two points in the set, the line segment joining these paths is also in the set. A *convex polyhedron* is the set produced by the intersection of the set of points satisfying a finite number of linear equalities and inequalities.

For an execution path with a set of simple linear equality or inequality predicate interpretations, the input space domain is a single convex polyhedron. If one or more simple linear nonequality predicate interpretations are added to this set, then the input space domain consists of the union of a set of disjoint convex polyhedra.

VI. Definitions of Types of Error

The basic ideas behind the classification of errors that we use are due to Howden [9], but our approach to defining them is somewhat more operational than that given in his paper.

From the previous sections, it is clear that a program can be viewed as

1) establishing an exhaustive partition of the input space into mutually exclusive domains, each of which corresponds to an execution path, and

2) specifying, for each domain, a set of assignment statements which constitute the domain computation.

Thus, we have a *canonical representation* of a program, which is a (possibly infinite) set of pairs $\{(D_1; f_1), (D_2; f_2), \cdots, (D_i; f_i), \cdots\}$ where D_i is the ith domain and f_i is the corresponding domain computation function.

Given an incorrect program P, let us consider the changes in its canonical representation as a result of modifications performed on P. It is assumed that these modifications are made using only permissible language constructs and results in a legal program.

Definition: A *domain boundary modification* occurs if the modification results in a change in the D_i component of some $(D_i; f_i)$ pair in the canonical representation.

Definition: A *domain computation modification* occurs if the modification results in a change in the f_i component of some $(D_i; f_i)$ pair in the canonical representation.

Definition: A *missing path modification* occurs if the modification results in the creation of a new $(D_i; f_i)$ pair such that D_i is a subset of D_j occurring in some pair $(D_j; f_j)$ in the canonical representation of P.

Notice that a particular modification (say a change of some assignment statement) can be a modification of more than one type. In particular, a missing path modification is also a domain boundary modification.

The errors that occur in a program can be classified on the basis of the modifications needed to obtain a correct program and consequent changes in the canonical representation. In general, there will be many correct programs and multiple ways to get a particular correct program. Hence, the error classification is not absolute, but relative to the particular correct program that would result from the series of modifications.

Definition: An incorrect program P can be viewed as having a *domain error* (*computational error*) (*missing path error*) if a correct program P^* can be created by a sequence of modifications, at least one of which is a domain boundary modification (domain computation modification) (missing path modification).

Several remarks are in order. The operational consequence of the phrase "can be viewed as" in the above definition is that the error classification is relative not only to a particular correct program, but also to a particular sequence of modifications. For instance, consider an error in a predicate interpretation such that an incorrect relational operator is employed, e.g., use of $>$ instead of $<$. This could be viewed as a domain error, leading to a modification of the predicate, or as a computation error, leading to a modification of the functions computed on the two branches. The fact that it might be more possible to change the relational operator rather than the function computations is a consequence of the language constructs, and it is not directly captured in the definitions of the types of error. In this paper, we would regard an error due to an incorrect relational operator as a domain error; it is a simpler modification to change the relational operator in the predicate than to interchange the set of assignment statements.

More specific characterizations of these errors can be made in the context of specific programming language constructs. In particular, the following informal description directly relates domain errors to the predicate constructs allowed in the language.

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 an incorrect 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. An incorrect predicate or assignment statement may affect many predicate interpretations, and consequently cause more than one border to be in error.

VII. The Domain Testing Strategy

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. Test points are generated for each border segment which, if processed correctly, determine that both the relational operator and the position of the border are correct. An error in the border operator occurs when an incorrect 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 or near the border are most sensitive to domain errors. A number of authors have made this observation, e.g., Boyer *et al.* [1] and Clarke [2].

It should be emphasized that the domain strategy does not require that the correct program be given for the selection of test points since only information obtained from the given program is needed. However, it will be convenient to be able to refer to a "correct border," although it will not be necessary to have any knowledge about this border. Define the *given border* as that corresponding to the predicate interpretation for the given program path being tested, and the corresponding *correct border* as that border which would be calculated in some correct program.

The domain testing strategy will be developed and validated under a set of simplifying assumptions:

1) Coincidental correctness does not occur for any test case.

2) A missing path error is not associated with the path being tested.

3) Each border is produced by a simple predicate.

4) The path corresponding to each adjacent domain computes a different function from the path being tested.

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

6) The input space is continuous rather than discrete.

Assumptions 1) and 2) have been shown to be inherent to the testing process, and cannot be entirely eliminated. However, recognition of these potential problems can lead to improved testing techniques. Assumptions 3) and 4) considerably simplify the testing strategy, for with them no more than one domain need be examined at one time in order to select test points, and as will be indicated shortly, a reduced number of test points will be required. As for the linearity assumption 5), the domain testing method has been shown to be applicable for nonlinear boundaries, but the number of required test points may become inordinate, and there are complex problems associated with processing nonlinear boundaries in higher dimensions. The continuous input space assumption 6) is not really a limitation of the proposed testing method, but allows points to be chosen arbitrarily close to the border to be tested. An error analysis has shown that pathological cases do exist in discrete spaces corresponding to integer data

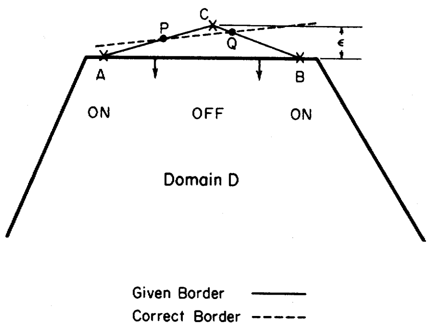


Fig. 2. Test points for a two-dimensional linear border.

types, but these occur only when domain size is on the order of the resolution of the discrete space itself; these results are available as [12].

VIII. Two-Dimensional Linear Inequalities

The test points selected will be of two types, defined by their position with respect to the given border. An *ON test point* lies on the given border, while an *OFF test point* is a small distance ϵ from, and lies on the open side of, the given 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.

Fig. 2 shows the selection of three test points A, B, and C for a closed inequality border segment. The three points must be selected in an ON-OFF-ON sequence. Specifically, if test point C is projected down on line AB, then the projected point must lie strictly between A and B on this line segment. Also, point C is selected a distance ϵ from the given border segment, and will be chosen so that it satisfies all the inequalities defining domain D except for the inequality being tested.

It must be shown that test points selected in this way will reliably detect domain errors due to boundary shifts. If any of the test points lead to an incorrect output, then clearly is an error. On the other hand, if the outputs of all these points are correct, then either the given border is correct, or if it is incorrect, Fig. 2 shows that the correct border must lie on or above points A and B, and must lie below point C, for by assumptions 1) and 4), each of these test points must lie in its assumed domain. So if the given border is incorrect, then the correct border can only belong to a class of line segments which intersect both closed line segments AC and BC.

Fig. 2 indicates a specific correct border from this class which intersects line segments AC and BC at P and Q, respectively. Define the *domain error magnitude* for this correct border to be the maximum of the distances from P and from Q to the given border. Then it is clear that the chosen test points have detected domain errors due to border shifts except for a class of domain errors of magnitude less than ϵ. In a continuous space, ϵ can be chosen arbitrarily small, and as ϵ approaches zero, the line segments AC and BC become arbitrarily close to the given border, and in the limit, we can

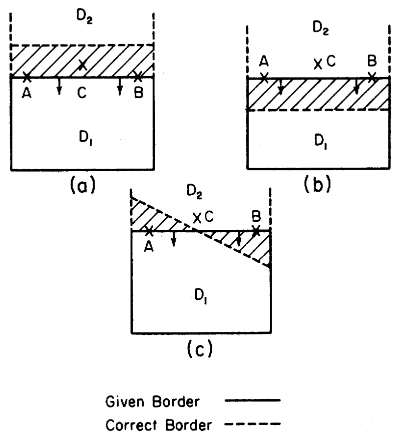


Fig. 3. The three types of border shifts.

conclude that the given border is identical to the correct border.

Fig. 3 shows the three general types of border shifts, and will allow us to see how the ON-OFF-ON sequence of test points works in each case. In Fig. 3(a), the border shift has effectively reduced domain D_1. Test points A and B yield correct outputs, for they remain in the correct domain D_1 despite the shifted border. However, the border has shifted past test point C, causing it to be in domain D_2 instead of domain D_1. Since the program will now follow the wrong path when executing input C, incorrect results will be produced. In Fig. 3(b), the domain D_1 has been enlarged due to the border shift. Here test point C will be processed correctly since it is still in domain D_2, but both A and B will detect the shift since they should also be in domain D_2. Finally, in Fig. 3(c), only test point B will be incorrect since the border shift causes it to be in D_1 instead of D_2. Therefore, the ON-OFF-ON sequence is effective since at least one of the three points must be in the wrong domain as long as the border shift is of a magnitude greater than ϵ.

Recall in Fig. 2 that we required the OFF point C to satisfy all the inequalities defining domain D except for the inequality being tested. The reason for this requirement is that some correct border segment may terminate on the extension of an adjacent border, rather than intersecting both line segments AC and BC as we have argued. In order to achieve this condition, C could always be chosen closer to the given border in order to satisfy the adjacent border inequalities.

We must also demonstrate the reliability of the method for domain errors in which the predicate operator is incorrect. If the direction of the inequality is wrong, e.g., $\leqslant$ is used instead of $\geqslant$, the domains on either side of the border are 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., $\leqslant$ is used instead of $<$. In this case, the only points affected lie on the border, and since we always test ON points, this type of error will always be detected. If the correct predicate is an equality, the OFF point will detect the error.

The domain testing strategy requires at most $3*P$ test points for a domain where P, the number of border segments on this boundary, is bounded by the number of predicates encountered on the path. However, we can reduce this cost by sharing test points between adjacent borders of the domain. The requirement for sharing an ON point is that it is an extreme point for two adjacent borders which are both closed or both open. The number of ON points needed to test the entire domain boundary can be reduced by as much as one half, i.e., the number of test points TP required to test the complete domain boundary lies in the following range:

$$2*P \leqslant TP \leqslant 3*P.$$

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.

IX. N-Dimensional Linear Inequalities

The domain testing strategy developed for the two-dimensional case can be extended to the general N-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-dimensional hyperplane is determined by N linearly independent points. So, whereas in the two-dimensional case we had to identify only two points on the correct border, in general we have to identify N points on the correct border and, in addition, these points must be guaranteed to be linearly independent. We might note that any N extreme points of the given border are automatically independent.

The validation of domain testing for the general linear case is based on the same geometric arguments used in the two-dimensional case. The key to the methodology is that the correct border must intersect every OFF-ON line segment, assuming that the test points are all correct. Since we must identify a total of N points on the correct border, N OFF-ON line segments are needed, and we can achieve this by testing N linearly independent ON test points on the given border and a single OFF test point whose projection on the given border is any convex combination of these N points. In addition, as in the two-dimensional case, the OFF point must also satisfy the inequality constraints corresponding to all adjacent borders.

Even though we do not know these specific points at which the correct border intersects the ON-OFF segments, we do know that these points must be linearly independent since the ON points are linearly independent. The OFF point is a distance ϵ from the given border, and in the limit as ϵ approaches zero, each OFF-ON line segment becomes arbitrarily close to the given border. However, as in the two-dimensional case, the ϵ-limitation means that only border shifts of magnitude greater than ϵ will be detected.

The domain testing strategy requires at most $(N + 1) * P$ test points per domain where N is the dimensionality of the input space in which the domain is defined and P is the number of border segments in the boundary of the specific domain. However, we again can reduce this testing cost by using extreme

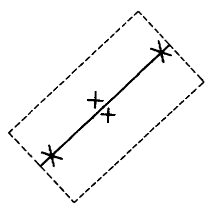


Fig. 4. Test points for an equality border.

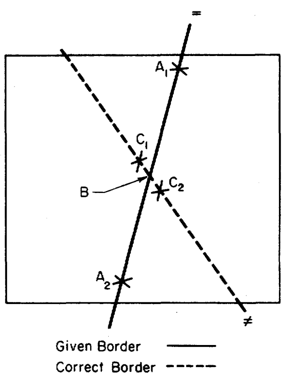


Fig. 5. A pathological case in domain testing for an equality predicate.

points as ON test points, and by sharing test points between adjacent domains.

X. Equality and Nonequality Predicates

Equality predicates constrain the domain to lie in a lower dimensional space. If we have an N-dimensional input space and the domain is constrained by L independent equalities, the remaining inequality and nonequality predicates then define the domain within the (N-L)-dimensional subspace defined by the set of equality predicates.

In Fig. 4, we see the equality border and the proposed set of test points. In a general N-dimensional domain, let us first consider a total of N ON points on the border and two OFF points, one on either side of the border. As before, the ON points must be independent, and the projection of each OFF point on the border must be a convex combination of the ON points.

Given an incorrect equality predicate, the error could be either in the relational operator or in the position of the border or both. The proposed set of test points can be shown to detect an operator or a position error by arguments analogous to those previously given. This set of points is also adequate for almost all combinations of operator and position errors, except for the following pathological possibility. Let us assume that the border has shifted and the correct predicate is a nonequality. If both OFF points happen to lie on the correct border while none of the ON points belongs to this border, the error would go undetected. This singular situation is diagrammed as the dashed border in Fig. 5 where A_1 and A_2 are the ON points and C_1 and C_2 are the OFF points. This problem can be solved by testing one additional point selected so that it lies both on the given border and the correct border for this case, i.e., at the intersection point of the given border with the line segment connecting the two OFF points. This additional point is denoted by B in the figure.

Each equality predicate can thus be completely tested using a total of (N + 3) test points. By sharing test points between all the equality predicates, this number can be considerably reduced, but the reduction depends upon values of N and L. In addition, since testing the equality predicates reduces the effective dimensionality to (N-L) for each of the inequality and nonequality borders, and the equality ON test points can be shared, even further reductions are possible.

For the case of a nonequality border, the testing strategy is identical to that of the equality border just discussed. The arguments for the validity of the strategy are analogous to those in previous cases. Again in this case, the pathological possibility discussed in connection with the equality predicate can occur, and can be handled in the same way. The major difference is that while test points can be extensively shared between equality and inequality borders, in general such sharing is not possible between nonequality and inequality borders.

The following proposition then summarizes the situation for testing linear borders in N-dimensions.

Proposition 1

Given assumptions 1)-6), with each OFF point chosen a distance ϵ from the corresponding border, the domain testing strategy is guaranteed to detect all domain errors of magnitude greater than ϵ using no more than $P * (N + 3)$ test points per domain.

XI. An Example of Error Detection Using the Domain Strategy

The domain testing strategy has been described and validated using somewhat complicated algebraic and geometric arguments. In this section, we hope to complement those discussions by demonstrating how a 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.

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:

$$-8 \leqslant I \leqslant 8 \qquad -5 \leqslant J \leqslant 5.$$

Even though the input space is assumed to be continuous, the coordinates of each test point are specified to an accuracy of 0.2 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 consists of three simple IF constructs, the first two of which are inequalities and the last of which is an equality. The input space structure is diagrammed in Fig. 6 where the solid diagonal border across the entire space is produced by the first predicate, the dashed

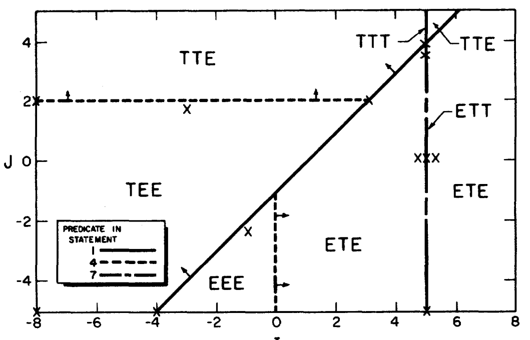


Fig. 6. Input space domain test points.

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 discuss, viz. TTE and ETT.

```
Statement Number    READ I, J;
       1            IF I ⩽ J + 1
       2               THEN K = I + J - 1;
       3               ELSE K = 2 * I + 1;
       4            IF K ⩾ I + 1
       5               THEN L = I + 1;
       6               ELSE L = J - 1;
       7            IF I = 5
       8               THEN M = 2 * L + K;
       9               ELSE M = L + 2 * K - 1;
                    WRITE M;
```

Table II illustrates two types of errors we would like to consider. The first is an error in the inequality predicate in statement 4 of the above program (K ⩾ I + 1) where it is assumed that the correct predicate should be (K ⩾ I + 2). This corresponds to an inequality border shift, and the modified domain structure is shown in Fig. 7. Three points have been selected to test this border, and it can be seen in Table II that the two ON points detect this error where M and M′ represent the output variables for the given program and for the assumed correct program, respectively. Note that as a result of this error, the vertical border at I = 0 in Fig. 6 has also shifted to I = 1 in Fig. 7 and, if tested, would also reveal this error.

Table II also shows the effect of an error in an equality predicate in statement 7 of the given program. It is assumed that the correct predicate should be (I = 5 - J) rather than the (I = 5) predicate which occurs in the given program. Fig. 8 shows the modified input space structure, and it can be seen that equality borders TTT and ETT have shifted. Table II shows the five points which test the ETT border, and note that two ON points both detect this shift.

XII. Extensions of the Domain Testing Strategy

Many assumptions were required in presenting the previous results, but to some extent, these assumptions were made to allow a simple exposition of the domain testing strategy. This section will discuss assumptions 3), 4), and 5) which deal with

TABLE II
DETECTION OF DOMAIN ERRORS FOR INEQUALITY AND EQUALITY PREDICATES

Domain in Error	Given Statement in Error	Given Predicate Interpretation	Assumed Correct Statement	Correct Predicate Interpretation	Test Points for This Border			
						Point	M	M'
TEE	4 IF $(K \geq I + 1)$ (Inequality Predicate) (See Fig. 7)	$J \geq 2$	IF$(K \geq I + 2)$	$J \geq 3$		ON $(-8, 2)$	-22	14
						OFF $(-3, 1.8)$	-4.6	-4.6
						ON $(3, 2)$	11	8
ETT	7 IF $(I = 5)$ (Equality Predicate) (See Fig. 8)	$I = 5$	IF $(I = 5 - J)$	$I = 5 - J$	two ON points	$(5, -5)$	23	27
						$(5, 3.8)$	23	27
					two OFF points	$(4.8, 0)$	26	26
						$(5.2, 0)$	28	28
					extra ON point	$(5, 0)$	23	23

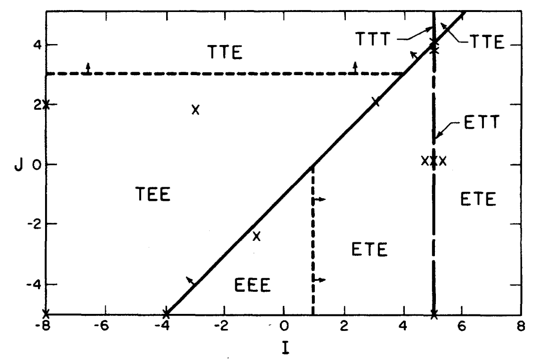


Fig. 7. Correct input space for a domain error.

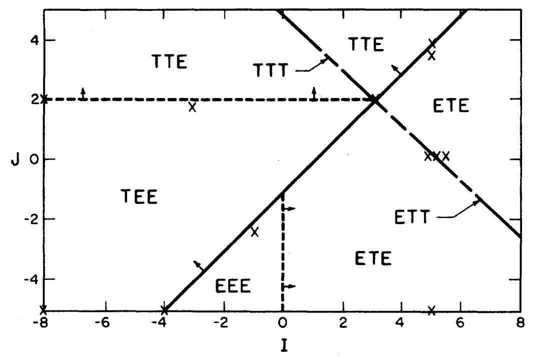


Fig. 8. Correct input space for an equality predicate error.

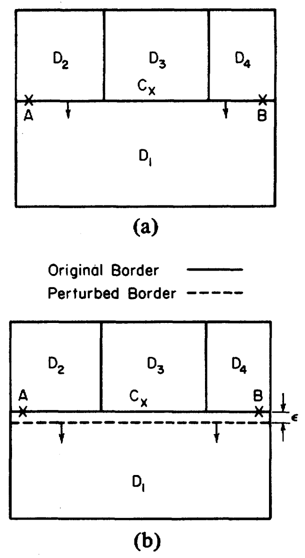


Fig. 9. The identification of adjacent domains.

compound predicates, adjacent domains which compute the same function, and nonlinear borders, respectively. The treatment of these cases will certainly require additional test points and, in some instances, will demand a considerable amount of extra processing. However, one of the main objectives of this section is to illustrate that none of the assumptions 3), 4), or 5) poses a theoretical limitation to the domain testing strategy which cannot be dealt with in some fashion.

First let us deal with the question of nonlinear predicates. 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. This question can be resolved by observing that if the nonlinear border can be limited to a restricted class specified by a finite set of parameters, then the domain strategy can be applied using a finite number of test points. This extension to nonlinear borders is examined in detail in [4] and [13]. Unfortunately, any but the simplest nonlinear predicates poses problems of extra processing which probably preclude testing except for restricted cases. For example, just finding intersection points of a set of linear and nonlinear borders can require an inordinate amount of computation.

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. We will demonstrate how domain testing can be modified to deal with this problem.

In Fig. 9(a), assuming domain D_1 were being tested, we must compare the functions calculated in domains D_1 and D_2 for test point A, D_1 and D_4 for B, and D_1 and D_3 for C. One of the major problems to be solved is the identification of these adjacent domains. We assume that when testing domain D_1, the partitioning structure of the adjacent domains and the program paths associated with these domains is not known. It would be very complicated to have to generate the domains which are adjacent to the border being tested.

Fig. 9(b) illustrates an approach to this problem. The border being tested is shifted parallel by a small distance ϵ, so that test points A and B now belong to adjacent domains D_2 and D_4, respectively. The modified program is then retested using test points A and B which will, as a byproduct, identify the paths associated with these two adjacent domains. We can then compare the output for each test point before and after the shift. If it is different, then we can definitely conclude that the adjacent domain computes a different function, and this test point can safely be used. If the output is the same for that test point, then we can conclude that either assumption 1) or 4) is violated. However, there is no way to decide this, and the only resolution is to use further test points. If we know that coincidental correctness cannot occur, then we could conclude on the basis of a single point that the adjacent domain computes the same function.

In summary, a technique of testing each point twice will assure us that assumption 4) is valid, and this redundancy might be viewed as a reasonable price to pay to eliminate this restriction. However, if an instance is found where the assumption is not valid, a basic theoretical problem exists.

Assumption 3) stated that a path contained only simple predicates, and this implied that the set of input points could be characterized quite simply as a single domain. We must consider what complications can occur for compound predicates, and how the domain strategy 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. If a compound predicate of the form [C(i) 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 as a single branch point on the path. No modifications of the domain testing strategy are required in this case.

However, compound predicates using the Boolean operator OR are more complicated. Consider a path containing the following predicates:

$$C_1, C_2, \cdots, [C_i \text{ OR } C_{i+1}], \cdots C_t.$$

The path condition in this case is the conjunction of these predicates, and in standard disjunctive normal form:

$$[C_1 \text{ AND } C_2 \text{ AND } \cdots \text{ AND } C_i \text{ AND } \cdots \text{ AND } C_t]$$
$$\text{OR } [C_1 \text{ AND } C_2 \text{ AND } \cdots \text{ AND } C_{i+1} \text{ AND } \cdots \text{ AND } C_t].$$

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 is possible.

Assuming linear predicates, each of these domains is a convex polyhedron, but the domains may overlap in arbitrary ways. 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 do not overlap, domains which partially overlap, and domains which totally overlap.

In the case where the domains do not overlap, the methodology can be applied to each domain separately. In the other two cases, more complicated procedures must be applied, and these details are discussed in [4] and [13].

In summary, compound predicates which utilize only the AND connective cause no special problems. However, when one or more OR connectives is used, a considerable amount of processing is required just to establish how adjacent domains overlap which correspond to this predicate. This additional processing probably precludes the domain strategy from constituting a viable approach for this case.

XIII. Conclusions and Future Research

The basic goal of this research is to replace the intuitive principles behind current testing procedures 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. In several cases, these limitations have proven to be theoretical problems inherent to the general path testing approach.

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 ϵ. 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, but unfortunately, nonlinear predicates pose problems of extra processing which probably preclude testing except for restricted cases.

Coincidental correctness is a theoretical limitation inherent to the path testing process, and we have argued 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 computation errors on a path. Within the class of all computable functions, there exist functions which coincide at an arbitrarily large number of points, but if there is sufficient resolution in the output space, further research is required to show that coincidental correctness is a rare occurrence for functions commonly encountered in data processing problems. The class of missing path errors, particularly those of reduced dimensionality, has proven to be another theoretical limitation to the reliability of any path testing strategy.

The domain testing strategy requires a reasonable number of test points for a single path, but the total cost may be unacceptable for a program containing a large number of paths. In particular, this may occur for programs with complicated control structures containing many iteration loops. Additional research is needed to substantially reduce the number of potential paths. One area being investigated is to obtain appropriate restrictions on control structures, especially iteration loops, so that a large number of paths can be tested as a single domain. If this approach is combined with the notion of module independence relative to domain errors, the combinatorial growth of the number of paths can be considerably reduced by domain testing small independent program modules separately. It remains to be shown that practical programs contain sufficient instances of such module independence to allow a reduction of the number of remaining paths to a practical figure.

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 domain test strategy is currently being implemented, and it will be utilized as an experimental facility for subsequent research. Experiments should indicate what sort of programming errors are most difficult to detect, and should yield extensive dynamic testing data.

Acknowledgment

The authors would like to thank B. Chandrasekaran for his assistance in preparing this paper and, in particular, for his contribution concerning the definitions of domain and computation errors.

References

[1] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT–A formal system for testing and debugging programs by symbolic execution," in *Proc. 1975 Int. Conf. Rel. Software*, Los Angeles, CA, Apr. 1975, pp. 234–245.

[2] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 215–222, Sept. 1976.

[3] E. I. Cohen and L. J. White, "A finite domain-testing strategy for computer program testing," Comput. Inform. Sci. Res. Cen., Ohio State Univ., Columbus, Tech. Rep. 77-13, Aug. 1977.

[4] E. I. Cohen, "A finite domain-testing strategy for computer program testing," Ph.D. dissertation, Ohio State Univ., Columbus, June 1978.

[5] J. L. Elshoff, "A numerical profile of commercial PL/I programs," Comput. Sci. Dep. General Motors Res. Lab., Warren, MI, Rep. GMR-1927, Sept. 1975.

[6] —, "An analysis of some commercial PL/I programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 113–120, June 1976.

[7] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156–173, June 1975.

[8] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554–560, May 1975.

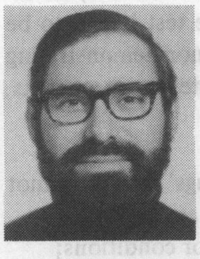
[9] —, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208–215, Sept. 1976.

[10] D. E. Knuth, "An empirical study of FORTRAN programs," *Software–Practice and Experience*, vol. 1, pp. 105–133, Apr.–June 1971.

[11] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the automated generation of program test data," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 293–300, Dec. 1976.

[12] L. J. White, F. C. Teng, H. C. Kuo, and D. W. Coleman, "An error analysis of the domain testing strategy," Comput. Inform. Sci. Res. Cen., Ohio State Univ., Columbus, Tech. Rep. 78-2, Sept. 1978.

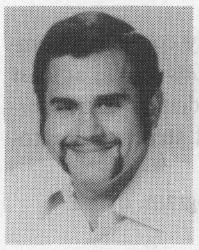
[13] L. J. White, E. I. Cohen, and B. Chandrasekaran, "A domain strategy for computer program testing," Comput. Inform. Sci. Res. Cen., Ohio State Univ., Columbus, Tech. Rep. 78-4, Aug. 1978.



Lee J. White (S'67–M'67) received the B.S.E.E. degree in electrical engineering from the University of Cincinnati, Cincinnati, OH, in 1962, and the Ph.D. degree in electrical engineering from the University of Michigan, Ann Arbor, in 1967.

He is currently a Professor and Chairman of the Department of Computer and Information Science, Ohio State University, Columbus, and holds a joint appointment in the Department of Electrical Engineering. His current research interests deal with the analysis of algorithms and software analysis and testing. He has published in the areas of pattern recognition, automatic document classification, combinatorial computing, and graph theory. He has served as a Consultant for the Monsanto Research Laboratory, Rockwell International Corporation, and a number of other industrial firms.

Dr. White is a member of the IEEE Computer Society, the Association for Computing Machinery, the Society for Industrial and Applied Mathematics, and Sigma Xi.



Edward I. Cohen was born in Boston, MA, in 1950. He received the B.S. degree in physics from Rensselaer Polytechnic Institute, Troy, NY, in 1972, and the M.S. and Ph.D. degrees, both in computer and information science, from Ohio State University, Columbus, in 1973 and 1978, respectively.

He was a Research and Teaching Associate in the Department of Computer and Information Science, Ohio State University, from 1972 to 1978. He worked as a Programmer for Neoterics, Inc., Columbus, in 1974 and as a Systems Analyst for the State Data Center, Columbus, in 1977. He is currently with the Poughkeepsie Programming Center, IBM Corporation, Poughkeepsie, NY. His current interests include program testing, software reliability, and high-performance multiprocessing system design.