Reliability of the Path Analysis Testing Strategy

WILLIAM E. HOWDEN

Abstract—A set of test data T for a program P is *reliable* if it reveals that P contains an error whenever P is incorrect. If a set of tests T is reliable and P produces the correct output for each element of T then P is a correct program. Test data generation strategies are procedures for generating sets of test data. A testing strategy is reliable for a program P if it produces a reliable set of test data for P. It is proved that an effective testing strategy which is reliable for all programs cannot be constructed. A description of the path analysis testing strategy is presented. In the path analysis strategy data are generated which cause different paths in a program to be executed. A method for analyzing the reliability of path testing is introduced. The method is used to characterize certain classes of programs and program errors for which the path analysis strategy is reliable. Examples of published incorrect programs are included.

Index Terms—Path analysis, program correctness, program testing, symbolic evaluation.

I. Introduction

OUR INTUITION tempts us into believing that when we test a program on a set of tests T we know more about the reliability of the program than its reliability over this set. When a program works correctly for a set of "well-chosen" tests we have the feeling that unless we have forgotten to test for some possible class of mistakes that the program is correct. This paper studies this phenomenon for a particular class of testing strategies.

II. Proving Correctness by Testing

Suppose that P is a program which is meant to compute a function F with domain D. Then the correctness of P can be determined by testing it on each element of D. If D is effectively infinite this approach is infeasible. In order to determine correctness by testing it is necessary to be able to generate a finite set $T \subset D$ which has the following two properties:

1) for each $x \in T$ there is a computable procedure for determining whether or not P terminates for x; and

2) $P(x) = F(x)$ for all $x \in T \Rightarrow P(x) = F(x)$ for all $x \in D$. The first condition is necessary so that a programmer will know when to stop a program which is caught in an infinite loop.

In practice, it is often possible to predict in advance an upper bound b(x) such that if P fails to terminate for x within computation time b(x), then P will not terminate. The correctness of programs having this property can be determined with test sets satisfying property 2) above. In this and the following sections we will assume that the programs P satisfy this property. The discussion could also be carried out without the assumption but at the cost of requiring that "reliable" test data satisfy both conditions 1) and 2) above, rather than just condition 2).

It is easy to prove that a test set T satisfying condition 2) exists for all programs P and functions F.

Theorem 1: Suppose that P is a program for computing a function F with domain D. There exists a finite subset T of D which can be used to determine the correctness of P, i.e., there exists a finite set $T \subseteq D$ such that

$$P(x) = F(x) \quad \text{for all} \quad x \in T \Rightarrow P(x) = F(x) \quad \text{for all} \quad x \in D.$$

Proof: Either P is correct or it is not. If P is correct choose $T = \{x\}$ for any $x \in D$. If P is not correct there exists some x such that $P(x) \neq F(x)$. Choose $T = \{x\}$. The set T satisfies the required conditions.

The proof of the above theorem is disappointing. If we knew whether or not P was correct it would not be necessary to test it. What is really required is a computable procedure, or test strategy, which can be used to generate a test set T for any program P. It can be proved that no such procedure exists.

Theorem 2: There exists no computable procedure H which, given an arbitrary program P and function F with domain D, can be used to generate a nonempty finite set $T \subset D$ such that:

$$P(x) = F(x) \quad \text{for all} \quad x \in T \Rightarrow P(x) = F(x) \quad \text{for all} \quad x \in D.$$

Proof: If such a procedure existed, it would be possible to use it to determine the equivalence of arbitrary primitive recursive functions. It is known that no such procedure exists [3]. The argument runs as follows. Let P_1 and P_2 be any two programs in a language for constructing programs which compute primitive recursive functions. Let F be the function computed by P_2. Since P_1 and P_2 are primitive recursive they terminate and H can be used for determining the correctness of P_1 for calculating F. This is equivalent to determining the equivalence of P_1 and P_2.

Theorem 2 tells us that the best that can be hoped for are test strategies which work for particular classes of programs $\mathcal{P}$. In the following sections we will examine the systematic testing strategy which is currently receiving the most attention. Classes of programs $\mathcal{P}$ are characterized for which the strategy is reliable. First the notion of reliability is considered.

III. Reliable Test Data

Any subset of a program's input domain can be considered a set of test data. A testing strategy is a procedure for choosing a set of test data.

Definition: Suppose P is a program for computing a function F whose domain is the set D. Let $T \subset D$. T is a *reliable test set* for P if:

Manuscript received December 30, 1975; revised March 1, 1976.

The author is with the Department of Applied Physics and Information Science, University of California, San Diego, La Jolla, CA 92037.

$P(x) = F(x)$ for all $x \in T \Rightarrow P(x) = F(x)$ for all $x \in D$.

Another way of stating this is that T is reliable for P if T reveals that P is incorrect whenever P contains an error (i.e., $P(x) \neq F(x)$ for some $x \in T$).

The testing strategies we will consider are actually procedures for choosing a sequence $\{T_i\}_{i=1}^{n}$ of subsets of the input domain of a program. Each of the sets T_i is chosen to test some aspect of the program. In path analysis approaches to testing, each set T_i consists of the subset of the input domain which causes a path P_i in the program to be executed. We assume that the programmer constructs a test set T from the sequence $\{T_i\}_{i=1}^{n}$ by choosing one element from each T_i. This approach to testing strategies makes it possible to consider different levels of reliability.

Definition: Suppose P is a program and H is a testing strategy. Let $\{T_i\}_{i=1}^{n}$ be the subsets of the input domain of P generated by H. Suppose that any set T which can be constructed by choosing one element from each T_i is a reliable test set. Then H is a *reliable test strategy* for P.

Theorem 3: Suppose that P is a program for computing a function F and H is a testing strategy for P. Let $\{T_i\}_{i=1}^{n}$ be the sequence of input domain subsets generated by H. Then H is a reliable test strategy for $P \Longleftrightarrow$ P is correct or there exists a set T_i such that $P(x) \neq F(x)$ for all $x \in T_i$.

In order to simplify our discussion we will assume in the following sections that input domain sets can be infinite and that variables are represented to arbitrarily precise accuracy. Some of the definitions, theorems, and examples are correct independently of this assumption. The others require occasionally awkward modifications.

In some cases a testing strategy H is "almost reliable" for a program P. Consider the program P in Example 1. The program is correct except for the missing initialization assignment SUM = X which should occur before the DO-loop. One possible strategy for testing P is to choose data which test all paths which cause less than n iterations of the loop in P. Suppose that the strategy H involves the generation of a sequence $\{T_i\}_{i=1}^{n}$ consisting of the subsets of the input domain which cause the loop in P to be executed i times, $1 \leqslant i \leqslant n$, and the selection of a set T containing one element chosen at random from each nonempty T_i. P gives incorrect answers for each element (X, E) of each T_i except for the pairs (Y, E) where Y is the default setting of the uninitialized variable SUM. Decompose each T_i into two sets T_i' and T_i'' where T_i' contains all (X, E) where $X \neq Y$ and T_i'' contains all (X, E) where X = Y. T_i, T_i', and T_i'' are all two-dimensional subsets of Euclidean 2-space. Since the two-dimensional area of T_i is nonzero and the two-dimensional area of T_i'' is zero, the chance of choosing an element at random from T_i which lies in T_i'' is negligible. It is almost certain that all of the elements of T will belong to the subsets T_i', and that the error in P will be revealed when P is tested on T. H is therefore "almost reliable."

Note that the phrase "almost reliable" is only meaningful for the strategy H in which elements of T_i are chosen at random. In practice, elements of T_i may not be chosen at random and it will not be meaningful to describe the associated strategy as being almost reliable.

Example 1: The following program P is supposed to compute sine(x), using the Maclaurin series, for any real number x. It is missing an initialization assignment.

```
      DOUBLE PRECISION FUNCTION SIN (X, E)
C     THIS DECLARATION COMPUTES SIN(X) TO ACCURACY E
      DOUBLE PRECISION E, TERM, SUM
      REAL X
      TERM = X
      DO 20 I = 3, 100, 2
      TERM = TERM * X**2/(I*(I-1))
      SUM = SUM + ((-1)**(I/2)) * TERM
      IF ( DABS (TERM) .LT. E) GO TO 30
20    CONTINUE
30    SIN = SUM
      RETURN
      END
```

Definition: Suppose that P is a program for computing a function F whose domain D is a subset of Euclidean n-space and H is a testing strategy for P. Let $\{T_i\}_{i=1}^{n}$ be the sequence of input domain subsets generated by H. Suppose the T_i have subsets T_i' such that any set T' containing one element from each T_i' is a reliable test set for P. If $T_i' = T_i$ whenever the volume of T_i is zero, and T_i' contains all of T_i except a subset of zero volume whenever the volume of T_i is nonzero, then H is an *almost reliable test strategy* for P.

Theorem 4: Suppose that P is a program for computing a function F whose domain D is a subset of Euclidean n-space and that H is a testing strategy for P. Let $\{T_i\}_{i=1}^{n}$ be the sequence of input domain subsets generated by H. Then H is an almost reliable test strategy for $P \Longleftrightarrow$ P is correct or there exists a set T_i such that:

1) the n-dimensional volume of T_i is zero and $P(x) \neq F(x)$ for all $x \in T_i$ or

2) the n-dimensional volume of T_i is nonzero and $P(x) \neq F(x)$ for all but a zero volume subset of T_i.

Theorems 3 and 4 indicate that we should attempt to choose test sets T_i so that if $P(x) \neq F(x)$ for some $x \in T_i$ then $P(x) \neq F(x)$ for (almost) all $x \in T_i$.

IV. Classes of Programs and Program Errors

The classes of programs $\mathcal{P}$ for which we will characterize the reliability of the path analysis testing strategy are associated with different kinds of errors in programs. We will be concerned with programs which are either correct or can be considered deviations from a hypothetical correct program P*. The "differences" between P and P* define the errors in P. Each class of programs $\mathcal{P}$ will consist of correct programs P* together with incorrect programs P which differ from P* by some type of error.

A *path* through a program corresponds to some possible flow of control. A path may be *infeasible* in the sense that there is no input data which will cause the path to be executed. Flows of control involving different numbers of iterations of loops

are considered to be different paths. In general, a program containing loops will have an infinite number of paths. The errors in a program can be categorized in terms of their effects on the paths through the program.

Associated with each path through a program is the subset of the input domain which causes the path to be followed and a sequence of computations which is carried out by the path.

Definition: Suppose P_i is a path through a program P. Then the *path domain* $D_i = D(P_i)$ for P_i is the subset of the input domain which causes P_i to be executed. The *path computation* $C_i = C(P_i)$ for P_i is the function which is computed by the sequence of computations in P_i.

The domain of the functions C_i is considered to be the domain D of P. During execution of the program P, each computation $C(P_i)$ is only carried out over the path domain $D(P_i)$. In general C_i may not be defined over all of D or, since P may contain errors, even over all of D_i. In comparing two computations C_i and C_j, we say that C_i and C_j are equivalent ($C_i = C_j$) if C_i and C_j are defined for the same subset D' of D and $C_i(x) = C_j(x)$ for all $x \in D'$.

Symbolic evaluation of a path can be used to construct a system of predicates which describes the path domain of any finite path in terms of the path's input variables [2], [12], [14]. Symbolic evaluation can also be used to construct a set of expressions describing the path computation for any finite path in terms of input variables. In the symbolic evaluation process, symbols are used to stand for symbolic input values and variables in expressions are bound by substitution of the symbolic expressions representing their current symbolic values. Example 2 contains the path domains and path computations for two of the paths in the program in Example 1.

Example 2:

a) Path which exits from loop during first iteration. *Path domain:* All (x, E) such that $|x^3/(3*2)| < E$. *Path computation:* SUM - $x^3/(3*2)$.

b) Path which exits from loop during second iteration. *Path domain:* All (x, E) such that $|x^3/(3*2)| \geq E$ and $|x^5/(5*4*3*2)| < E$. *Path computation:* SUM - $x^3/(3*2) + x^5/(5*4*3*2)$.

The effects of program errors on the paths through a program can be described in terms of their effects on the path domains and path computations of the paths. Three simple classes of errors will be studied. If there is an isomorphism (one-to-one correspondence) between the paths P_i of P and the paths P_i^* of the correct version P* of P such that $D(P_i) = D(P_i^*)$ and $C(P_i) = C(P_i^*)$ for all paths, then P = P* and P is correct. If P is not correct, no isomorphism having those properties can be constructed. Either the domains or the computations, or both, of P and P* will be different.

Definition: Suppose P is an incorrect program for computing a function F and P* is a correct program. Suppose there is an isomorphism between the paths P_i of P and the paths P_i^* of P* such that for all pairs of paths (P_i, P_i^*), $D(P_i) = D(P_i^*)$ but that for some pair (P_k, P_k^*), $C(P_k) \neq C(P_k^*)$. Then P contains a *path computation* or *computation error.*

Definition: Suppose P is an incorrect program for computing a function F and P* is a correct program. Suppose there is an isomorphism between the paths P_i of P and the paths P_i^* of P* such that for all pairs of paths (P_i, P_i^*), $C(P_i) = C(P_i^*)$, but that for some pairs (P_k, P_k^*), $D(P_k) \neq D(P_k^*)$. Then P contains a *path domain* or *domain error.*

Definition: Suppose P is an incorrect program for computing a function F and P* is a correct program. Suppose there is an isomorphism between the paths P_i of P and a subset of the paths P_i^* of P* such that $C(P_i^*) = C(P_i)$ and $D(P_i^*) \subset D(P_i)$ for all paths P_i in P. Then P contains a *subcase error.*

Definition: $\mathcal{C}(P)$ is the set of all path computations for all paths in the program P. $\mathcal{D}(P)$ is the set of all path domains for all paths in P.

When a program contains a computation error we assume that the paths in P and P* have been indexed so that $D(P_i) = D(P_i^*)$ for all paths. When it contains a domain or a subcase error we assume they have been indexed so that $C(P_i) = C(P_i^*)$ for all paths P_i in P.

Different relationships can be proved between classes of statement type errors and errors which are defined in terms of the domains and computations for a program.

Theorem 5: Suppose that P is an incorrect program and that the only difference between P and P* is in some statement which does not affect the flow of control in P. Then P has a computation error.

Theorem 6: Suppose that P is an incorrect program and that the only difference between P and a correct program P* is in some statement which affects the flow of control in P. Then P may have a computation, domain, or subcase error.

V. Path Analysis Testing Strategy

In the path analysis approach to testing a program, P is tested by generating test data which cause selected paths in P to be executed. Much of the current work in test data generation involves systems which automate parts of the path analysis testing strategy. In some of the systems the user selects program paths and the computer generates descriptions of the data which cause the paths to be followed. In other systems the program is automatically decomposed into classes of paths and one path is selected from each class. All of the systems result in the generation of a sequence of sets $\{T_i\}_{i=1}^n$ which correspond to path domains or to unions of path domains.

In practice, a program P may have an infinite number of paths. Any practical path analysis strategy will have to involve a procedure for selecting a subset of the total set of paths. In the analysis carried out in this section the potential reliability of path analysis strategies is examined by considering the degree of reliability that could be obtained if it were possible to test every path in a program. In this idealized situation the path analysis testing strategy results in the generation of a sequence of sets $\{T_i\}_{i=1}^n$ which corresponds to the complete set of path domains for a program. A (possibly infinite) set of test data T is constructed by choosing one element at random from each nonempty set T_i. This testing strategy will be referred to as *P-testing.*

Definition: Suppose P_i is a path in a program P. Then $P_i(x)$ is computed by carrying out the sequence of nontransfer statement computations in P_i, i.e., $P_i(x) = C_i(x)$ where $C_i = C(P_i)$.

Theorem 7: Suppose that $\mathcal{P}$ is a set of programs containing a correct program P* for computing some function F. Then P-testing is reliable for testing the programs P in $\mathcal{P} \Longleftrightarrow$ each

$P \in \mathcal{P}$ is either correct or has a feasible path P_i such that $P_i(x) \neq P^*(x)$ for all $x \in D(P_i)$.

In the following subsections the reliability of P-testing is characterized for different classes of programs $\mathcal{P}$.

A. Computation Errors

In this subsection we will assume that $\mathcal{P}$ is a set of programs for computing a function F and that the programs in $\mathcal{P}$ are either correct or contain computation errors. Recall that the type of error an incorrect program contains is defined with respect to a particular correct program. Each of the incorrect programs P in $\mathcal{P}$ are assumed to have computation errors relative to a particular correct program P* which is also in $\mathcal{P}$. The paths in the incorrect programs P and the correct program P* are indexed so that $D(P_i) = D(P_i^*)$ for all paths.

Theorem 8: P-testing is reliable for testing the programs in $\mathcal{P} \Longleftrightarrow$ every program P in $\mathcal{P}$ is either correct or has a feasible path P_i such that $P_i(x) \neq P_i^*(x)$ for all $x \in D(P_i)$.

All of the examples in this section are taken from the "Common Blunders" section in *The Elements of Programming Style* [13]. Incorrect statements in the programs are italicized. Corrections are also italicized and are enclosed in angle brackets. The original incorrect programs are the programs without the statements in angle brackets. The corrected programs are the programs which contain the italicized statements in angle brackets but not the other italicized statements.

Example 3: The following program P is supposed to compute the number of class marks which fall within certain ranges. It contains an incorrect assignment statement which causes a computation error.

```
      DO 40 I=1, N
C     TEST IF DATA IS IN RANGE
      IF (MARKS(I) .LT. 1 .OR. MARKS(I) .GT. 100) GO TO 30
C     TRANSFORMATION TO DIRECTLY DETERMINE CLASS
C     INTERVAL MEMBERSHIP
      J = MARKS(I) - 1/10 + 1
      < J = (MARKS(I) - 1)/10 + 1 >
      NCLASS(J) = NCLASS(J) + 1
      GO TO 40
30    WRITE (3, 102) MARKS(I)
102   FORMAT (' ***MKS001 - DATA OUT OF RANGE ', 112)
40    CONTINUE
```

P-testing is reliable for testing P. (Let P_i be the path in P which causes the DO-loop to be traversed exactly once and which causes the incorrect statement to be executed during that iteration.)

Twelve of the eighteen errors in Kernighan and Plauger [13] are computation errors. P-testing is reliable or almost reliable for discovering nine of these errors. For one error it was not immediately obvious whether or not P-testing is reliable. In general, the P-testing reliability question is undecidable.

B. Domain Errors

In this subsection we will assume that $\mathcal{P}$ is a set of programs which are either correct or contain domain errors. The paths in the incorrect programs P and in the corresponding correct programs P* are indexed so that $C(P_i) = C(P_i^*)$ for all paths in P.

The conditions for the reliability of P-testing for domain errors are simplified when a program's paths are "distinct."

Definition: Suppose P_1 and P_2 are two paths in a program P with domain D. P_1 and P_2 are *distinct* if $P_1(x) \neq P_2(x)$ for all $x \in D$. The paths in a program are distinct if any pair of paths in the program is distinct.

Theorem 9: Suppose that the paths in each correct program P* in $\mathcal{P}$ are distinct. Then P-testing is reliable for testing the programs in $\mathcal{P} \Longleftrightarrow$ every program P in $\mathcal{P}$ is either correct or has a feasible path P_i such that $D(P_i) \cap D(P_i^*) = \phi$.

Proof:

1) Suppose P-testing is reliable for each $P \in \mathcal{P}$. Let $P \in \mathcal{P}$ and suppose P is not correct. Theorem 7 implies there exists a feasible path P_i such that for all $x \in D(P_i)$, $P_i(x) \neq P^*(x)$. Suppose $D(P_i) \cap D(P_i^*) \neq \phi$. Choose $x \in D(P_i) \cap D(P_i^*)$. $C(P_i) = C(P_i^*) \Rightarrow P_i(x) = P_i^*(x)$. $x \in D(P_i^*) \Rightarrow P^*(x) = P_i^*(x) \Rightarrow P_i(x) = P^*(x)$ which is contradictory.

2) Suppose $P \in \mathcal{P}$ has a feasible path P_i such that $D(P_i) \cap D(P_i^*) = \phi$. Let $x \in D(P_i)$. $D(P_i) \cap D(P_i^*) = \phi \Rightarrow x \in D(P_j^*)$ for some $j \neq i$. $C(P_i) = C(P_i^*)$ and paths distinct $\Rightarrow P_i(x) \neq P_j^*(x)$. $P^*(x) = P_j^*(x) \Rightarrow P_i(x) \neq P^*(x)$. $\Rightarrow$ P-testing is reliable for P.

The necessary conditions for reliability in the above theorem do not depend on path distinctness. The following theorem is a special case of Theorem 9.

Theorem 10: If P-testing is reliable for testing the programs in $\mathcal{P}$ then every program $P \in \mathcal{P}$ is either correct or it has a feasible path P_i such that $D(P_i) \cap D(P_i^*) = \phi$.

Theorem 10 can be used to prove that P-testing is not reliable for the program P in the following example. For all paths P_i in P, $D(P_i) \cap D(P_i^*) \neq \phi$.

Example 4: The following program P is supposed to compute sine(x) using the Maclaurin series. It contains an incorrect transfer statement which causes a domain error.

```
      DOUBLE PRECISION FUNCTION SIN (X, E)
C     THIS DECLARATION COMPUTES SIN(X) TO ACCURACY E
      DOUBLE PRECISION E, TERM, SUM
      REAL X
      TERM = X
      SUM = X
      DO 20 I = 3, 100, 2
      TERM = TERM * X**2/(I*(I-1))
      SUM = SUM + ((-1)**(I/2))* TERM
      IF (TERM .LT. E) GO TO 30
      < IF (DABS (TERM) .LT. E) GO TO 30 >
20    CONTINUE
30    SIN = SUM
      RETURN
      END
```

The paths in the above program are not distinct. They are "almost distinct" in the sense that for any pair of paths P_i^* and P_j^* in P*, $P_i^*(x) = P_j^*(x)$ for at most a zero area subset of the two-dimensional domain of P*. A theorem similar to Theorem 9 can be proved which characterizes the conditions under which P-testing is almost reliable for programs containing almost distinct paths. P-testing is neither reliable nor almost reliable for this example.

Three of the eighteen errors in Kernighan and Plauger [13] are domain errors. P-testing is reliable or almost reliable for discovering only one of these three errors.

C. Subcase Errors

In this subsection we will assume that $\mathcal{P}$ is a set of programs which are either correct or contain subcase errors. The paths in the programs P and in the corresponding programs P* are indexed so that $C(P_i) = C(P_i^*)$ for all paths in P.

The domain error theorems in Section V-B depend only on the equivalence of path computations in P and P*. The same theorems can also be proved for subcase errors.

Theorem 11: Suppose that the paths in each correct program P* in $\mathcal{P}$ are distinct. Then P-testing is reliable for testing the programs in $\mathcal{P}$ $\Longleftrightarrow$ every program $P \in \mathcal{P}$ is either correct or has a feasible path P_i such that $D(P_i) \cap D(P_i^*) = \phi$.

Theorem 12: If P-testing is reliable for testing the programs in $\mathcal{P}$ then every program $P \in \mathcal{P}$ is either correct or it has a feasible path P_i such that $D(P_i) \cap D(P_i^*) = \phi$.

The paths in the correct version of the following program are distinct. Theorem 11 or Theorem 12 can be used to prove that P-testing is not reliable for P. For all paths P_i in P, $D(P_i) \cap D(P_i^*) \neq \phi$.

Example 5: The following program P is supposed to compute and print out a table of monthly mortgage payments. The program is missing a transfer statement. The omission causes a subcase error.

```
        DECLARE (A, R, M, B, C, P) FIXED DECIMAL (13, 4);
L10:    GET LIST (A, R, M);
        PUT SKIP EDIT ('THE AMOUNT IS', A) (A(13), F(10, 2))
                      ('THE INTEREST RATE IS', R) (A(23), F(6, 2))
                      ('THE MONTHLY PAYMENT IS', M) (A(25), F(8, 2));
        IF M < = A*R/1200 THEN GO TO L30;
        PUT SKIP(3) EDIT
        (' MONTH    BALANCE    CHARGE    PAID ON PRINCIPLE') (A);
        PUT SKIP;
        B = A;
        DO I = 1 TO 60;
           C = B*R/1200;
           IF B+C < M THEN GO TO L20;
           P = M - C; B = B-P;
           PUT SKIP EDIT (I, B, C, P) (F(13), 3 F(13, 2));
        END;
        C = B*R/1200;
<L20:   IF B+C <.005 THEN GO TO L10 >
 L20:   PUT SKIP(2) EDIT ('THERE WILL BE A LAST PAYMENT
                          OF:' , B+C) (A(35), F(8, 2));
        GO TO L10;
L30:    PUT SKIP (2) EDIT ('UNACCEPTABLE MONTHLY PAYMENT') (A);
        GO TO L10;
```

Only one of the eighteen errors in Kernighan and Plauger [13] is a subcase error.

D. Combinations of Errors

Each of the programs in the above examples contains a single error. In each case the error is caused by a single incorrect statement. In some cases it is intuitively meaningful to describe a program having several incorrect statements as containing a single error while in other cases the program should be described as involving several errors. We will informally define an error E in a program P to be a set of "differences" between P and a correct program P*.

Two of the programs in the above examples are partially corrected versions of the programs in Kernighan and Plauger [13]. The original versions of these programs contained more than one error. In general, a program may contain several errors and it is important to consider the combinatorial effects of the errors on the reliability of the testing strategy. In this section we will characterize a set of conditions under which the reliability of P-testing for single errors is preserved when errors occur in combination.

Definition: Suppose $E = \{E_1, E_2, \cdots, E_k\}$ is a set of errors in a program P. Let P_{E_i} be the program which contains only the error E_i and not the other errors (i.e., E_i is the difference between P and a given correct program P*). Let $P_E = P$. An error E_i in E is *independent* in E if for all x in the domain of P,

$$P_{E_i}(x) \neq P^*(x) \Rightarrow P_E(x) \neq P^*(x).$$

An error E_i in a program is independent relative to a set of errors E if the introduction of the other errors into the program does not "correct" any of the incorrect output caused by the error E_i.

Theorem 13: Let $E = \{E_1, E_2, \cdots, E_n\}$ be a set of computation errors in a program P (i.e., P_{E_i} has a computation error for each E_i). Suppose some error E_i is independent in E. Then if P-testing is reliable for P_{E_i} it is also reliable for P_E.

Proof: If P-testing is reliable for P_{E_i} then there exists a nonempty path domain D_j in $\mathcal{D}(P_{E_i})$ such that $P_{E_i}(x) \neq P^*(x)$ for all $x \in D_j$. Since E contains computation errors, $\mathcal{D}(P_E) = \mathcal{D}(P_{E_i})$ which implies that $D_j \in \mathcal{D}(P_E)$.

The conditions for the preservation of P-testing reliability

are more complicated when other than computation errors are involved.

Theorem 14: Let $E = \{E_1, E_2, \cdots, E_n\}$ be a set of errors in a program P. Suppose E contains an independent error E_i and that P-testing is reliable for P_{E_i}. The reliability of P-testing for P_{E_i} implies the existence of a domain D_j in $\mathfrak{D}(P_{E_i})$ such that $P_{E_i}(x) \neq P^*(x)$ for all $x \in D_j$. If $D_j \in \mathfrak{D}(P_E)$ or $D_j \supset D$ for some $D \in \mathfrak{D}(P_E)$ then P-testing is reliable for P_E.

In practice, we will want each error E_i in a set E to satisfy the conditions of the above theorem relative to each subset $\overline{E}$ of E containing E_i. This will ensure that P-testing reliability is preserved as the errors in E are discovered and removed from the program.

The only simple example of error independence and testing reliability in Kernighan and Plauger [13] involves errors which are "almost independent" and for which P-testing is almost reliable. Suppose E_i is an error in a set of errors E and that P is a program whose domain is a subset of Euclidean n-space. Let X be the subset of the domain D of P for which $P_{E_i}(x) \neq F(x)$. Then E_i is almost independent in E if $P_E(x) \neq F(x)$ for all $x \in X$ if X has zero n-dimensional volume or for all but a zero volume subset of the elements of X if X has nonzero volume. Both Theorems 13 and 14 can be rewritten as theorems involving errors which are almost independent and strategies which are almost reliable.

Example 6: The program P in this example is another version of the sine program in Example 4. In its original form in Kernighan and Plauger [13] the program contains several errors, including the three in this example.

```
      DOUBLE PRECISION FUNCTION SIN(X, E)
C     THIS DECLARATION COMPUTES SIN(X) TO ACCURACY E
      DOUBLE PRECISION E, TERM, SUM
      REAL X
      TERM = X
      < SUM = X >
      DO 20 I = E, 100, 2
      TERM = TERM *X**2/(I*(I-1))
      SUM = SUM + (-1**(I/2))* TERM
      < SUM = SUM + ((-1)**(I/2))*TERM >
      IF (TERM .LT. E) GO TO 30
      < IF (DABS(TERM) .LT. E) GO TO 30 >
20    CONTINUE
30    SIN = SUM
      RETURN
      END
```

P_{E_1} is the program containing the missing assignment SUM = X. P_{E_2} contains the transfer statement with the missing DABS function and P_{E_3} the assignment with the missing parentheses. P_{E_2} is the program P in Example 4.

P-testing is almost reliable for P_{E_1} and P_{E_3} but not for P_{E_2}. E_1 and E_2 are almost independent relative to any subset of $E = \{E_1, E_2, E_3\}$. $\mathfrak{D}(P_{E_1})$ and $\mathfrak{D}(P_{E_2})$ both contain domains D_j which satisfy the special conditions of Theorem 14. This implies that P-testing is almost reliable for $P_{\overline{E}}$ where $\overline{E}$ is any subset of E containing either E_1 or E_3. Once E_1 and E_3 have been discovered and removed from P_E, P-testing will no longer be reliable for the remaining program P_{E_2}.

VI. Related Work

A number of different program testing tools have been developed which can be used to help automate the testing of programs. Earlier work in the area concentrated on the testing of program statements and branches rather than on program paths. The Algol W compiler [18] has an option which will cause a users program to be instrumented so that when the program is executed a table of statement execution counts is generated. The Fortran preprocessor PET [19] can be used to generate similar types of information for the statements and program branches in Fortran programs.

Several research groups have been involved in the design and construction of testing tools which concentrate on the testing of program paths. King [14] has built an interactive system in which the user directs the system to carry out a symbolic evaluation of selected paths. The system automatically constructs representations of the domains and computations for the paths. In the SELECT system [2] the user can either select a path or cause the system to select all paths which do not iterate a loop more than some given number of times. The system automatically constructs representations of the domains and computations for the paths. For simple cases it automatically generates test data. The basic elements of the path analysis testing strategy are described in [12]. A system implementing some of the features of the complete system plan outlined in [12] is described in [11]. The system automatically decomposes a program into a finite set of classes of paths and then generates descriptions of the computations and domains for each class. A logical notation in which groups of predicates can be asserted over the range of an expression is used to represent the domains for an infinite class of paths. The system was developed as part of the McDonnell Douglas Astronautics program in software reliability. The Program Validation Project at the General Research Corporation has constructed a commercially available system [15]. The system keeps track of untested program segments in a program and prints out descriptions of test data that will cause paths containing the modules to be executed. The TRW System [9] generates descriptions of a minimal set of paths through a program which tests all of the program's branches. Clarke [5] has constructed a path analysis testing system which interfaces with the DAVE Data Flow Analysis System [16]. The system is capable of generating temporary assertions which cause the generation of test data that test for common errors such as out of bounds array references and division by zero.

The path analysis testing strategy has features in common with and to some extent is derived from research on proofs of program correctness. Deutsch's interactive program verifier [6] uses a symbolic evaluation process to generate verification conditions. In [4] Burstall uses symbolic evaluation to construct inductive proofs of program correctness. The relationship between the use of symbolic evaluation in generating verification conditions to prove correctness and in generating a set of predicates for constructing test data is described in [10].

A number of papers have recently appeared in which the authors describe research into the classification and analysis of errors. Reference [7] contains a classification of the errors occurring in a release of an operating system and a discussion of the programming methods and constructs that would have

prevented the errors from occurring. Shooman and Bolsky [17] describe the errors which were reported during the development of a 4K real time program. In their analysis of these errors, the authors discuss the nature of the changes required to correct the errors. The results of an analysis of a large system are reported in [1]. The authors describe a system for detecting classes of errors during the design phase of a software project. Their data are derived from a study reported in [20]. The study contains a comprehensive categorization of errors into both general and more specific categories. The frequency of the occurrence of the errors in several large projects is listed.

In general, the classifications and analysis of errors which have been carried out describe errors in terms of program constructs (e.g., incorrect loop condition, structural error, incorrect indexing, bit manipulation error, etc.). These classes of errors can be related to their effects on the computations and domains in a program but are not directly useful for characterizing the reliability of P-testing.

To the author's knowledge, the only other work which attempts to provide an underlying formal basis for the study of testing is described in a recent paper by Goodenough and Gerhart [8]. The use of the term "reliable" is derived from the work described in this paper.

The work described in this paper was influenced by the paper by Goodenough and Gerhart, and draws from the work on testing tools.

VII. Conclusions and Future Work

The path domain/path computation approach results in a relatively simple classification of commonly occurring errors. Sixteen of the eighteen errors in the Common Blunders section of [13] are either computation, domain, or subcase errors. The error classification can be refined to distinguish between other types of less commonly occurring errors.

The domain/computation approach provides a framework for the analysis and characterization of the reliability of path analysis testing strategies. The reliability of P-testing was analyzed for several classes of common errors.

P-testing was found to be reliable or almost reliable for about 65 percent of the program errors in the small survey of 11 programs in Kernighan and Plauger [13]. This means that if data for testing those programs are selected using the P-testing strategy, we will be "almost certain" of detecting 65 percent of the errors. This does not mean that the other errors would not be detected, only that we could not be certain of their detection.

The research described in this paper is only an introductory analysis of testing reliability. A complete analysis of reliability would involve a more extensive classification of errors. In our analysis we studied the potential reliability of path analysis testing strategies by assuming that we could test every path in a program. In any practical strategy the paths will have to be grouped into a finite set of classes of paths and one path from each class tested. The effects of different methods for grouping paths on the reliability of P-testing needs to be characterized. The continued study of P-testing reliability will have to deal with the problem of roundoff errors and data types. The programs in Kernighan and Plauger [13] contain two action errors for which P-testing is not reliable. The first involves the use of mixed mode expressions and the second results from roundoff of type REAL numbers.

The need for the development and analysis of testing strategies which are more powerful than the path analysis strategy is noted. How can the path testing method be supplemented to get closer to 100 percent reliability? A programmer has three sources of information for constructing test data: the program to be tested, its specifications, and his knowledge of commonly occurring programming errors. Path analysis testing strategies use only one of these sources. More complex strategies will involve the integration of several sources of information.

References

[1] B. W. Boehm, R. K. McClean, and D. B. Urfrig, "Some experience with automated aids to the design of large-scale reliable software," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 125–133, Mar. 1975.

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

[3] W. S. Brainerd and L. H. Landweber, *Theory of Computation*. New York: Wiley, 1974.

[4] R. M. Burstall, "Proving correctness as hand simulation with a little induction," in *Proc. IFIPS 1974*. Amsterdam, The Netherlands: North-Holland, 1974.

[5] L. Clarke, "A system to generate test data and symbolically execute programs," Dep. Computer Sciences, Univ. of Colorado, Boulder, CO, Rep. CU-CS-060-75, Feb. 1975.

[6] L. P. Deutsch, "An interactive program verifier," Ph.D. dissertation, Univ. of California, Berkeley, May 1973.

[7] A. Endres, "An analysis of errors and their causes in system programs," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975.

[8] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975.

[9] R. H. Hoffman, "NASA/Johnson Space Center approach to automated test data generation," in *Proc. Computer Science and Statistics: 8th Annu. Symp. on the Interface*, Los Angeles, CA, Feb. 1975.

[10] W. E. Howden, "Automatic generation of program test data and proofs of program correctness," Workshop on the Attainment of Reliable Software, Univ. of Toronto, Apr. 1974.

[11] W. E. Howden and J. Laub, "Automatic case analysis of programs," in *Proc. Computer Science and Statistics: 8th Annu. Symp. on the Interface*, Los Angeles, CA, Feb. 1975.

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

[13] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*. New York: McGraw-Hill, 1974.

[14] J. C. King, "A new approach to program testing," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975.

[15] E. F. Miller, "RXVP: An automated verification system for FORTRAN," in *Proc. Computer Science and Statistics: 8th Annu. Symp. on the Interface*, Los Angeles, CA, Feb. 1975.

[16] L. J. Osterweil and L. D. Fosdick, "Data flow analysis as an aid in documentation, assertion generation, validation, and error detection," Dep. Computer Science, Univ. of Colorado, Boulder, CO, Rep. 15, Sept. 1975.

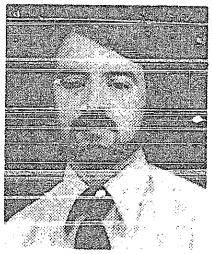
[17] M. L. Shooman and M. I. Bolsky, "Types, distribution, and test and correction times for programming errors," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975.

[18] R. L. Sites, *ALGOL W Reference Manual*, Stanford Univ., Stanford, CA, STAN-CS-71-230, 1971.

[19] L. G. Stucki, "Automatic generation of self-metric software," in

Proc. IEEE Symp. Computer Software Reliability, New York, NY, 1973.
[20] T. A. Thayer *et al.*, "Software reliability study," TRW Rep. 74-2260.1.9-29, June 1974.

William E. Howden was born in Vancouver, Canada, on December 8, 1940. He received the B.A. degree in mathematics from the University of California, Riverside, in 1963, the M.Sc. degree in mathematics from Rutgers University, New Brunswick, NJ, in 1965, the M.Sc. degree in computer science from Cambridge University, Cambridge, England, in 1970, and the Ph.D. degree in computer science from the University of California, Irvine, in 1973.



In 1965 and 1966 he was with Atomic Energy of Canada, Chalk River, Ont. From 1970 to 1974 he was a Lecturer in computer science at the University of California, Irvine. Since 1973 he has been a consultant to McDonnell Douglas, Huntington Beach, in software reliability. He is currently Assistant Professor of Information and Computer Science at the University of California, San Diego. His research interests are in software and system reliability and in interactive problem solving.

Dr. Howden is a member of the Association for Computing Machinery and the British Computing Society.

A System to Generate Test Data and Symbolically Execute Programs

LORI A. CLARKE

Abstract—**This paper describes a system that attempts to generate test data for programs written in ANSI Fortran. Given a path, the system symbolically executes the path and creates a set of constraints on the program's input variables. If the set of constraints is linear, linear programming techniques are employed to obtain a solution. A solution to the set of constraints is test data that will drive execution down the given path. If it can be determined that the set of constraints is inconsistent, then the given path is shown to be nonexecutable. To increase the chance of detecting some of the more common programming errors, artificial constraints are temporarily created that simulate error conditions and then an attempt is made to solve each augmented set of constraints. A symbolic representation of the program's output variables in terms of the program's input variables is also created. The symbolic representation is in a human readable form that facilitates error detection as well as being a possible aid in assertion generation and automatic program documentation.**

Index Terms—**Program validation, software reliability, symbolic execution, test data generation.**

I. Introduction

THERE is a growing awareness of the problems involved in testing programs and of the need for automated systems to aid in this process. This paper describes an implemented system that aids in the selection of test data and the detection of program errors.

The usual approach to program testing relies solely on the intuition of the programmer. The programmer generates data to test the program until satisfied that the program is correct. The success of this method depends on the expertise of the programmer and the complexity of the program. Experience has shown that this approach to testing programs is inadequate and costly [1]. Consequently, several alternative approaches have been proposed. These approaches can be categorized into two areas, program corrections (also called program verification or program proving) and program validation.

In the program correctness method formal mathematical proofs are used to demonstrate that a program terminates and satisfies the program's specifications. First, assertions about the program's variables are made at various points in the code and then theorem proving techniques are employed to verify the correctness of these assertions. In general, automated theorem proving techniques are used, though human assistance is still needed [2].

Program correctness has focused attention on the problems of program reliability. However, the state of the art is such that there are many drawbacks that prevent program correctness from being a practical tool, at least in the immediate future. Major difficulties are the creation of program assertions and the considerable human interaction frequently required in the theorem proving stages. Even after this rather complex process the results may be questionable. If the program cannot be proved correct this may be due to an error in the program but also may be due to a flaw in the assertions or limitation in the theorem prover, human or machine. Even if the program is proved correct, this process still may be questionable. In addition, proving programs cor-

Manuscript received October 6, 1975; revised May 14, 1976. This work was supported in part by the National Science Foundation under Grant GJ 36461.

The author is with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA.