Perturbation Testing for Computation Errors

Steven J. Zeil

COINS Technical Report 83-23
July 1983

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the National Science Foundation, grant no. MCS-8210084

Abstract

The use of algebraic techniques in defining a neighborhood of functions is particularly suited to testing for computation errors. Two possible approaches are Howden's algebraic testing method and perturbation testing, which in this paper is generalized to permit analysis of individual test points rather than entire paths. These approaches are shown to be mathematically equivalent when applied to a program's black-box output. Perturbation testing, however, offers more flexibility in the choice of potential errors to be investigated. A significant alternative offered by perturbation testing is the ability to work in the static domain, choosing test data to eliminate possible error terms in specific assignment and output statements.

I. Introduction

A common approach in testing research has been the design of testing strategies to detect all members of some class of errors. Even though the type of errors in a given program are probably not known prior to testing, various complementary strategies can then be employed to cover a variety of error types.

A classification of errors which has proven useful is the division into domain and computation errors. A domain error occurs when incorrect output is generated due to executing a wrong path through the program [10]. A strategy for detecting domain errors has been presented by Cohen and White [15], with improvements suggested by Clarke, Hassell, and Richardson [3], but will not be considered further here.

This paper is concerned with methods for detecting computation errors, which occur when the correct path through the program is taken, but incorrect output is generated. Howden has proposed an interesting approach to detecting computation errors called "algebraic testing" [11,12]. An alternative approach is perturbation testing [16,17], which centers on the idea of deriving the set of functions which would yield the same test results as the program, and choosing new test data to minimize the size of that set. In this paper, the techniques of perturbation testing will be expanded to cover computation errors, and the strong relationship between the algebraic and perturbation testing methods will be explored.

II. Algebraic Testing

The key idea behind algebraic testing is the use of standard analysis techniques for distinguishing among all functions in a given class, such as the class of linear functions or polynomials of fixed degree [11,12]. For example, it is well known that any n+1 distinct points will distinguish all

degree n polynomials in one variable. If one knew, therefore, that a given program computed a degree n polynomial, and that the intended function of that program was also a polynomial of degree n, then any n+1 distinct test points would suffice to completely test that program.

This technique can be extended to more elaborate programs. Howden cites lesser known results which show that the class of multinomial functions in k variables with exponents less than t can be tested using t^k points arranged in a configuration called a "cascade set" [11,12]. In fact, it will be shown later that algebraic testing is closely related to classical interpolation problems. By implication, it can be extended to any vector space of functions, since interpolation problems are solvable on vector spaces. The class of functions must, however, have been previously analyzed to derive a set of rules for constructing distinguishing sets of test data.

It can be argued that the number of points required for algebraic testing on multinomials is unnecessarily inflated by treating each input uniformly. It may be extremely unlikely that some inputs (e.g. flags) would ever appear in the output raised to a power higher than one, or that certain combinations of variables would ever be multiplied together in a term of the output multinomial. The number of points in a cascade set is required only if every variable in the multinomial may be raised to the same maximum power and may be arbitrarily multiplied by the other variables. Other arguments for reducing the number of points required have been made by DeMillo and Lipton [6] and by Rowland and Davies [14], but a full discussion of these is beyond the scope of this paper.

The requirement that the total output function of a program be a multinomial function is also highly restrictive, especially since the effect of conditional statements in a program is to partition the program's input domain, with different functions computed on each subdomain. It is possible to show that certain classes of programs may still be handled with this strategy [12], but even then the number of test points grows exponentially with the number of inputs.

Nevertheless, the overall approach has many attractive features. When partitioning prevents the description of the total program function as a multinomial, these techniques could be applied to the output functions of the individual partitions. The major drawback to this approach is the multiplication of the number of points required for testing by the number of subdomains being tested. This drawback is at least partially offset by the expectation that the individual partial functions will be considerably simpler than the aggregate program function, and so can be tested with fewer data points each. In addition, a major portion of the research into program testing has dealt with choosing paths or input partitions from which to select points for testing [9,10,13]. Applying algebraic testing to partial functions would permit its combination with these other techniques.

Even when the form of the correct output is not known, algebraic testing can provide significant confidence. Suppose that a set of test points chosen to distinguish some class of polynomial or multinomial functions executes correctly. The unique function from the chosen class which would yield the same answers as the program forms an interpolating polynomial/multinomial for the program function. Hence the degree of confidence the tester gains in the correctness of the program is proportional to the closeness with which the program function can be approximated by the interpolating multinomial of the degree chosen. While interpolating polynomials in general need not converge to an arbitrary function (although convergence can be guaranteed if a finite bound exists on the higher order derivatives of the function being approximated), they often do form good approximations, especially at points well inside the extreme values of the chosen test points [4].

In the sections which follow, an alternative to algebraic testing will be presented which retains its chief advantages while providing more flexibility in the choice of functional classes and in dealing with partitioned functions.

III. Blindness Expressions for Test Points

The general problem of selecting test data in order to determine whether an arbitrary program computes the same function as an arbitrary specification is unsolvable [10]. A reasonable compromise is to design testing strategies which are capable of distinguishing between the given program and any members of some class of related programs [2,8,17]. Intuitively, as the size of this class increases, so does the confidence gained from the testing method. This intuition has been justified formally by Gourlay [8]. The tester is then required to make some assertion about the relationship between the given program and the correct program it approximates, defining a neighborhood of programs in which a correct version is expected to lie.

Previous research by the author has demonstrated a method for deriving the set of potential errors in arithmetic expressions which are missed by a set of previously selected test paths, no matter what points are chosen within the path domains [16,17]. An error in an arithmetic expression in a given statement can be represented by the addition of a perturbing function to the "correct" form of the expression. This view of incorrect expressions is more flexible than might be initially apparent, since an error term can involve subtracting out the correct expression and adding a completely different one.

The selection of a class of possible perturbing functions defines the neighborhood of programs to be examined. Little can be done if the class of functions from which the potential error terms are drawn is unrestricted. If, however, the set of possible error terms is believed to be a finite-dimensioned vector space (e.g. a set of polynomials or multinomials of fixed degree), then the set of error terms which cannot possibly be detected by a given test path can be readily computed [16,17].

This analysis can be easily extended to individual data points. Begin by describing the state of the program at any point in its execution in terms of the current environment, $\bar{v}$, where

$$\bar{v} = (x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n).$$

The x_i are the inputs to the program, and the y_i are the current values assigned to the program's variables. A path within a program is an ordered sequence of statements representing a possible flow of control. Each assignment statement in a path transforms the environment, generating a new value for one of the program variables. For any path P_A we can designate a function C_A which represents a transformation equivalent to that of the program computations along that path:

$$\bar{v}_A = C_A(\bar{v})$$

Initially, only the input values, x_i, will be considered to be defined, with the program variables being defined later via these computations. Input statements will therefore be treated as assignments of input values to the appropriate variables.

When a statement containing an arithmetic expression is executed, the value of that expression is determined by substituting the appropriate values from the current environment for each of the variables in the expression. Hence if an arithmetic expression T is evaluated at the end of path P_A, the resulting value is $T(\bar{v}_A)$, where $\bar{v}_A$ is the environment resulting from the execution of path P_A. This can be expressed as a function of the program inputs by utilizing the path computation:

$$T(\bar{v}_A) = T \circ C_A(\bar{v}_0)$$

where $\bar{v}_0$ is the initial environment (with only inputs and constants defined), and "$\circ$" denotes functional composition.

Consider now the possibility that the expression T has been replaced by an erroneous expression T'. Then (T'-T) represents an error term which was added to T to produce the incorrect expression T'. While in general this error term could be any function, in practice the original expression, T', is directly observable in the program, and the tester may have some notion of the

correct functional class of T. If so, this intuition constrains the set of possible error terms. An assertion that the error terms fall within some finite-dimensioned vector space (such as a set of polynomials or multinomials of some fixed degree) will permit useful analysis without being overly restrictive. In all of the discussion which follows, if later inspection should reveal that the tester's initial choice of error class was too limited, a more general functional class can be substituted with no penalty for the initial mistake as long as the new class of possible errors entirely contains the old class. This would be the case, for example, if testing was simply moved from one degree of multinomial to a higher degree.

Since the error terms will be taken from vector spaces, an appropriate representation is defined by

$$\alpha\tilde{e} = T' - T$$

where α is any real number (and is non-zero whenever $T' \neq T$), and $\tilde{e}$ is a function drawn from the chosen vector space which has been normalized according to some appropriate norm. $\tilde{e}$ may be considered the "direction" of the error and α the "magnitude".

The conditions under which $\alpha\tilde{e}$ may be detected will vary somewhat depending on the type of statement and error for which testing is being done [16]. In this paper we shall be concerned with computation errors. In most languages, computation errors are caused by faults in assignment statements or in the expressions which form the arguments of output statements. The case of faults in assignment statements turns out to be the more general of these two and so will be considered first.

An incorrect assignment statement may be represented as

$$v_j = T'(\bar{v})$$

where v_j is any program variable. If $\bar{v}_A$ is the environment when this statement is reached, then the fault in this statement is detectable exactly

when $T'(\bar{v}_A) \neq T(\bar{v}_A)$ and a subsequent output statement uses v_j directly or indirectly. By "using" v_j, we mean that a small change in the value of v_j would cause some change in the output. To capture this idea formally, we will say that a function $f(\bar{v})$ is <u>partially dependent</u> on v_j if, for some k, the k<u>th</u> partial derivative of f with respect to v_j is nonzero when evaluated at $\bar{v}$. For many functional classes such as multinomials, partial dependency is easily determined.

Theorem 1.

Let C_A be the computation performed along a path P_A from the start of the program up to but not including an assignment statement

$$v_j = T'(\bar{v})$$

and C_B be the computation for a path P_B leading from but not including that assignment statement to an output statement which prints the value of an expression M. Let $\bar{v}_0$ be an input causing P_A, the assignment statement, and P_B to be executed. If $\alpha\tilde{e}$, the error term in T', is in some vector space E, then the error in T' is detectable with input $\bar{v}_0$ exactly when

$$M \circ C_B \text{ is partially dependent on } v_j$$
$$\text{and}$$
$$\tilde{e} \notin \{ e: e \circ C_A(\bar{v}_0) = 0 \quad \text{and} \quad e \in E \}.$$

The proof of this theorem is essentially identical to the theorem proven in [16] for determining the set of undetectable errors for an entire path. In fact, this problem can be reduced to the one analyzed there by replacing each input statement by a set of assignment statements which set the appropriate variables equal to constants whose values are identical to those found in the input stream. Alternatively, this theorem can be proven by noting that the final line describes the set of all functions in E which interpolate to the constant zero function on the environment $\bar{v}$ and which therefore clearly could

not have been detected, if indeed they had been added in as error terms.

It is worth noting that the magnitude of the error term plays no part in the detectability of the error. If $\tilde{e}$ cannot be detected, than neither can $\alpha\tilde{e}$. In such cases, the test input $\bar{v}_0$ is said to be <u>blind</u> to $\tilde{e}$, and the final line of the theorem defines the space of errors to which the test data $\bar{v}_0$ is blind. This "blindness space" is itself a vector space, and therefore can be described using a finite set of characteristic elements. When the partial dependence requirement is not satisfied, any error term will go undetected. In such a case the test input $\bar{v}_0$ will be considered blind to the entire space of potential errors, E.

As an example of the application of this theorem, consider the subprogram RECIP shown in figure 1. Suppose that Theorem 1 is applied to the assignment statement in line 6. As a first choice of error class, it seems reasonable to use functions of about the same level of complexity as those appearing in the program itself. Consequently we will use

$$\begin{aligned} E(\bar{\alpha}) = {} & \alpha_0 + \alpha_1 A + \alpha_2 X0 + \alpha_3 ABSERR + \alpha_4 X + \alpha_5 XLAST \\ & + \alpha_6 DIFF + \alpha_7 OLDDIF + \alpha_8 A*X0 + \alpha_9 A*X + \alpha_{10} A*XLAST \\ & + \alpha_{11} X0*X + \alpha_{12} X0*XLAST + \alpha_{13} X*XLAST + \alpha_{14} A*X0**2 \\ & + \alpha_{15} A*X**2 + \alpha_{16} A*XLAST**2 \end{aligned} \qquad (1)$$

```
 1:            SUBROUTINE RECIP(A,X0,X,ABSERR)
               C
               C**  This program uses the Newton-Raphson algortihm to estimate
               C** the reciprocal of A without employing division, given an initial
               C**  estimate X0.  Iteration is halted when the product of A and the
               C**  newest estimate of 1/A is within ABSERR of 1.0, or when two
               C**  successive such products show that the estimate is moving away
               C**  from the true reciprocal.
               C
 2:                  X = X0
 3:                  DIFF = 1.E20
 4:              1   OLDDIF = DIFF
 5:                  XLAST = X
 6:                  X = 2. * X - A * X**2
 7:                  DIFF = ABS(1. - X * A)
 8:                  IF ((DIFF .LT. ABSERR) .AND. (X .NE. 0.)) GO TO 99
 9:                  IF (DIFF .LT. OLDDIF) GO TO 1
10:              2   WRITE (6,3) A,X0
11:              3   FORMAT(' ALGORITHM DOES NOT CONVERGE TO 1/',G13.6,
                    * ' WITH STARTING POINT ',G13.6)
12:             99   RETURN
13:                  END
```

Figure 1. RECIP routine

for all real α_i as our space of potential errors. Suppose that RECIP is tested with inputs (A=3, X0=0.1, ABSERR=0.001). When statement 6 is reached, the environment will be $\bar{v}_A$= (A=3, X0=0.1, ABSERR=0.001, X=0.1, XLAST=0.1, DIFF=1.0E20, OLDDIF=1.0E20). It happens that the partial dependence requirement for this statement is satisfied for all non-zero X. By theorem 1 then, the set of possible error terms which could be added to the right-hand side expression but not be detected with this execution are those for which $E(\alpha) \circ \bar{v}_A = 0$. Substituting the values from the environment into $E(\alpha)$ gives a linear equation for α which is solvable by a relatively trivial manipulation. The solution set, the set of undetected potential error terms, is the set of all linear combinations of the expressions in figure 2.

This solution set indicates, for example, that instead of

```
X = 2*X - A*X**2
```

we could have written

```
X = 2*X - 3*X**2,
X = 2*0.1 - A*0.1**2,
```

or, less obviously,

```
X = 2*X0 - A*X**2,
X = X0 + X - A*XLAST**2,
```

or any of a literally infinite number of alternate expressions which would give the same result as the original expression and hence any of which may in fact be the correct form for this program.

(A - 3)	(X0 - .1)	(ABSERR - .001)
(X - .1)	(XLAST - .1)	(DIFF - 1.E20)
(OLDDIF - 1.E20)	(A*X0 - .3)	(A*X - .3)
(A*XLAST - .3)	(X0*X - .01)	(X0*XLAST - .01)
(X*XLAST - .01)	(A*X0**2 - .03)	(A*X**2 - .03)
(A*XLAST**2 - .03)		

Figure 2. Blindness Space for First Execution of Statement 6

Continuing the execution, the program would return to statement 6 a number of times, each time with a different environment. On the second iteration, for example, the environment would be (A=3, X0=0.1, ABSERR=0.001, X=0.17, XLAST=0.17, DIFF=0.49, OLDDIF=0.49). and the blindness space would be the span of the expressions appearing in figure 3.

Some of the expressions here are the same as before; many are different. This reflects the fact that the new environment is not completely different from the old one. The value of A, for example, is unchanged, and so the term (A - 3) remains. The value of X has changed, and consequently the undetectable expressions involving X are different.

When the expression to be tested appears in an output statement, the situation is somewhat simpler. A statement of the form

PRINT $T'(\bar{v})$

can be conceptually split into two parts by introducing a temporary variable:

TEMP := $T'(\bar{v})$

PRINT TEMP.

Theorem 1 can then be applied to examine the effects of testing T'. C_B becomes simply the identity function, and M is a selector function which extracts the value of TEMP from the environment. The partial dependence requirement is clearly satisfied, and so the final line of Theorem 1 defines the blindness space for expressions in output statements.

(A - 3)	(X0 - .1)	(ABSERR - .001)
(X - .17)	(XLAST - .17)	(DIFF - .49)
(OLDDIF - .49)	(A*X0 - .3)	(A*X - .51)
(A*XLAST - .51)	(X0*X - .017)	(X0*XLAST - .017)
(X*XLAST - .0289)	(A*X0**2 - .03)	(A*X**2 - .0867)
(A*XLAST**2 - .0867)		

Figure 3. Blindness Space for Second Execution of Statement 6

Of course, a single test run is seldom considered sufficient. As additional test runs are made without finding an error, some increase in the level of confidence associated with the program is expected. Because one test may uncover errors to which an earlier test was blind, the size of the total blindness space should be reduced by additional testing. This intuition is formalized in Theorem 2.

Theorem 2.

Let P_A and P_B be subpaths satisfying the conditions of Theorem 1, and E' be the blindness space remaining from previous tests such that $E' \subseteq E$. Then if partial dependence is satisfied on P_B, the error in T' is detectable exactly when

$$\tilde{e} \notin E' \cap \{ e: e \circ C_A(\bar{v}_0)=0 \quad \text{and} \quad e \in E \}.$$

The proof of this theorem follows directly from the observation that an error term is detected if it is detectable with the most recent test or with any of the earlier tests. Hence the total blindness space for a set of tests is formed as the intersection of the individual spaces.

A test point is useful, therefore, if it reduces the size of this intersection. Since the individual blindness spaces are vector spaces, their intersection must also be a vector space. The size of a vector space may be measured by that space's dimension. Hence a proposed test is useful only if it reduces the number of characteristic expressions in the blindness space. A simple rule of thumb in choosing test data is therefore to select input data for which any one or more of the characteristic blindness expressions is non-zero.

In the example given previously, separate blindness spaces were found for different executions of the statement being tested. The set of potential error terms remaining after both executions is given by the intersection of

the spaces described by figures 2 and 3. This intersection can be computed by standard linear algebraic techniques [7,16], which have been incorporated into a prototype system which computes blindness expressions for FORTRAN programs. The intersection of these two spaces is the span of the expressions shown in figure 4. This space has fewer terms (i.e., a smaller dimension) since the combination of both executions allows fewer errors to escape detection than would have escaped either execution alone. In fact, if the execution of the input (A=3, X0=0.1, ABSERR=0.001) is allowed to run to completion, the intersections of the individual blindness spaces for each encounter with the statement being tested is reduced to the space described in figure 5.

In choosing the next test inputs, we should attempt to find a point for which any one or more of the expressions in figure 5 is non-zero. This may be trivial (e.g., find an input for which (A - 3) is non-zero), somewhat more

```
(A - 3)                    (X0 - .1)          (ABSERR - .001)
(X - XLAST)                (DIFF - OLDDIF)    (A*X0 - .3)
(A*X - 3*X)                (A*XLAST - 3*X)    (10*X0*X - X)
(10*X0*XLAST - X)          (A*X0**2 - .03)    (A*X**2 - A*XLAST**2)
(OLDDIF + 1.43E21*X - 2.43E20)                (A*X**2 - 3*X*XLAST)
(100*A*XLAST**2 - 81*X + 5.1)
```

Figure 4. Intersection of Previous Two Spaces

```
(A -3)                     (X - XLAST)        (DIFF - OLDDIF)
(A*X - 3*X)                (A*XLAST - 3*X)    (X0*X - X0*XLAST)
(ABSERR + .00225*X0 - .001225)                (A*X0 - 3*X0)
(5*A*X0**2 - 9*X0 + .75)                      (A*X**2 - 3*X*XLAST)
(A*XLAST**2 - 3*X*XLAST)
```

Figure 5. Errors Escaping Detection With (A=3, X0=0.1, ABSERR=0.001)

difficult (e.g., find inputs for which (A*XLAST**2 - 3*X*XLAST) is not zero), or completely impossible (e.g., there is no input for which (DIFF - OLDDIF) is non-zero at statement 6).

As additional useful test points are chosen, the blindness space will eventually be reduced to the set of expressions like (DIFF - OLDDIF), the set of expressions in E which are invariantly equal to zero at that point in the program. Such expressions, being always equal to zero, cannot possibly affect the program's execution. Consequently, when this point has been reached, all terms in E which can be considered as errors have been successfully tested. Unfortunately, it is not decidable when this point has been reached. How much difficulty this will pose in practice is unclear and will, no doubt, depend strongly on how complex a space is initially chosen for E.

A "complete" test of RECIP would require the computation of blindness expressions at every assignment and output statement. This does not, however, mean that the amount of test data will grow in direct proportion to the number of lines of code. Two factors act to significantly reduce the amount of test data.

First, not all statements require as elaborate an error function as was employed with statement 6. Since, for example, FORTRAN does not permit arbitrary expressions in WRITE statements, there is no need to test for quadratic or higher degree error terms in WRITE statements. Note the distinction that, although it is possible for the value of X to be in error by a quadratic function of the inputs, it is not possible to cause such an error by changing the WRITE statement itself. Such an error must result from a fault in a previously executed assignment statement. Testing for such an error is controlled by the blindness expressions at the assignment statements. Additional arguments can be advanced for simpler error terms in assignment statements based upon the tester's understanding of those statements and their purpose in the program. In effect, we allow the tester to perform "gedanken experiments" in deciding that certain error terms are highly unlikely in selected assignment statements.

The second factor serving to limit the amount of test data for perturbation testing over an entire program is both more rigorous and probably more significant. Choose any two adjacent assignment/output statements such that the second one is executed if and only if the first has just completed execution. Assume that there is a certain subset of the space of possible error terms which is being tested by both statements. Then any test point which is useful in eliminating a part of that common error space for the second statement will also be useful for the first statement, providing that the partial dependency requirement is satisfied for the first statement somewhere along the path for that data point [16]. This implies that a natural unit on which to perform test data selection is any block of consecutive assignment/output statements containing no internal labels to which execution might branch. Such groups of consecutive statements will tend to be tested almost uniformly by a given set of test data.

IV. Perturbations and Algebraic Testing

The theorems of the preceding section can be used to examine algebraic testing. Algebraic testing takes a black box approach which can be viewed as collapsing the entire program into a single statement:

$$\text{PRINT } T'(\bar{v}_0)$$

where T' is now the function computed by the program under test. A set of possible program functions, $\{T_i\}$, is postulated for which rules are known by which test points may be selected to distinguish any two members of $\{T_i\}$.

For perturbation testing, a class of error functions E was postulated such that $\{T'+e_i: \forall e_i \in E\}$ represented the set of possible arithmetic expressions which might appear in that statement. If, as is the case for the real-valued functions on which algebraic testing has been defined, the set $\{T_i\}$ is closed under addition [11,12], then algebraic testing can be treated as the special case of perturbation testing where $T' \in E$.

When $T' \in E$, the goal of test data selection to distinguish all $\{T_i\}$ is equivalent to reducing the total blindness space to the empty set. Any set of test data which satisfies either of these criteria without revealing an error proves that T' is the only function in E which could be correct. In fact, there is no inherent reason why algebraic testing must have $T' \in E$. The view of E as a set of perturbing functions is more appropriate to the neighborhood paradigm, and allows these techniques to be meaningfully applied to a wider range of programs, but the exact same set of data points will serve either interpretation of a given E.

For example, if X, the "black box" output of the reciprocal program, were expected to match the Taylor expansion of 1/A through at least three terms, then a reasonable choice for $E=\{T_i\}$ would be a multinomial of the form

$$E(\bar{\alpha}) = \alpha_0 + \alpha_1 A + \alpha_2 X0 + \alpha_3 ABSERR + \alpha_4 A*X0 + \alpha_5 A**2 + \alpha_6 X0**2 + \alpha_7 X0*A**2 + \alpha_8 A*X0**2 \qquad (2)$$

The error space used here is expressed purely in terms of inputs to the module, in accordance with the "black box" approach of algebraic testing.

As before, we will choose (A=3, X0=0.1, ABSERR=0.001) as the first test input. The blindness space for this input is shown in figure 6. If the next test point is chosen as (A=3, X0=0.5, ABSERR=0.0001), the space of undetected potential errors is reduced to the span of the expressions in figure 7. Eventually, after testing with the points listed in figure 8, the space of undetected potential errors from E becomes empty.

(A - 3)	(X0 - .1)	(ABSERR - .001)
(A*X0 - .3)	(A**2 - 9)	(X0**2 - .01)
(X0*A**2 - .9)	(A*X0**2 - .03)	

Figure 6. Errors Escaping Black-Box Testing With (A=3, X0=0.1, ABSERR=0.001)

The number of points required here is less than the $t^k=3^3=27$ points indicated in Howden's treatment simply because the use of blindness analysis permitted more control over the choice of perturbing function. Rather than use all second degree multinomial terms, we were able to drop certain terms such as ABSERR*X0. This seems quite reasonable given our understanding of the purpose of ABSERR in this program, which makes it unlikely to appear in any nonlinear terms of a computation. The ability to do such pruning of the possible functional forms becomes more important as the multinomial degree and the number of variables increase. Opportunities for such pruning will also become more apparent as the number of variables in a program increases, because of the natural groupings of variables which might reasonably appear together in any calculation. For example, HOURLY-WAGE and DAYS-WORKED may reasonably appear together, but HOURLY-WAGE and PHONE-NUMBER form such an unlikely pair that it would be wise not to worry about terms involving their

```
(A - 3)                  (A*X0 - 3*X0)          (A**2 - 9)
(X0*A**2 - 9*X0)         (A*X0**2 - 3*X0**2)
(ABSERR + .00225*X0 - .001225)
```

Figure 7. Errors Escaping Detection With
(A=3, X0=0.1, ABSERR=0.001)
(A=3, X0=0.5, ABSERR=0.0001)

A	X0	ABSERR
3.0	0.1	0.001
3.0	0.5	0.0001
8.0	0.1	0.001
0.2	3.0	0.001
0.01	10.0	0.001
5.0	0.1	0.001
0.125	10.0	0.001
0.01	2.0	0.01
0.3333	1.0	0.001

Figure 8. Complete Data Set for Black Box Test

product.

V. Static versus Dynamic Measures

The choice of error function in the above RECIP example may have seemed inappropriate to some. Even granting the arguments presented earlier regarding the relationship between testing and interpolation, one might question the appropriateness of multinomials as approximators to that program's function. In particular, although the output for any given path through the program is indeed a multinomial, the coefficients and degree of that multinomial depend on which path is chosen. Of course, if one is willing to go to much higher degree multinomials, approximations may be found for even piece-wise continuous functions like RECIP. This hardly seems, however, to be a natural approach to the problem. The penalty to be paid for such an awkward technique is the exponential growth in the amount of test data required as the degree of multinomial increases.

It is the author's contention that a major part of this awkwardness stems from basing the testing model on the dynamic properties of the function computed by the program rather than on the static properties of the code. The terms "dynamic" and "static" must be taken here in a rather loose sense, since as one considers more and more of the (static) properties of a piece of code, one eventually must determine the (dynamic) program semantics. Hence these terms denote opposite poles of a range of possible properties.

It may seem natural to emphasize dynamic quantities because the testing/verification problem (i.e. show that a program computes the same function as its specification) is expressed in those terms. Intuitively, it appears that any testing method which aspires to the solution of this problem must work primarily at the dynamic level.

The neighborhood paradigm for testing introduced in Section III, however, permits a very different approach to obtaining reliable tests. Under this paradigm, one attempts to determine whether the given program is the best approximation to the given specification from a neighborhood of similar programs. Of course, in the limit as the neighborhood size is increased, this paradigm approaches the testing/verification problem, but the neighborhood model opens the way for methods which provide a spectrum of reliability, allowing the tester to select the degree of rigor desired and the amount of effort which is economically feasible. Testing methods for which such spectra are evident include algebraic and perturbation testing, where the degree of reliability depends on the generality of the chosen class of error terms, and mutation testing where the choice of mutation operators may be varied to accomodate the economics of a given situation [5].

The neighborhood paradigm, per se, is not restricted to either static or dynamic models. The similarity property which defines a neighborhood may be chosen based on a variety of program properties. When, however, the goal is to provide a spectrum of reliability, static properties appear more practical.

Consider, for example, the case where algebraic testing has been only partly completed, or has been completed on a relatively simple class of functions, and the testers are trying to decide whether additional tests are required. They may have some very good intuition about what types of errors are plausible in their program, but they must still determine whether all those plausible errors have been accounted for by the tests conducted so far. Does eliminating all third degree multinomial errors from the dynamic function of a program like RECIP mean that, for example, all substitutions of one variable for another have been checked? If not, what level of algebraic testing will permit such a guarantee? Such questions are extremely difficult to answer because they bridge the gap between the dynamic world in which algebraic testing is conducted and the static one in which errors are commonly described.

In many ways, dynamic complexity seems almost orthogonal to the effort required for testing. Intuitively one would believe, for example, that the more often a given statement is executed by a given set of test data, the more confidence is gained in the correctness of that statement. This common sense rule has been observed experimentally [1], and the theorems of section III may be considered as a formal justification in which the rule holds as long as each new execution is referenced and occurs on a significantly changed environment. Hence loops tend to make a program easier to test, while "IF" statements tend to have the opposite effect. This is true despite the fact that both constructs tend to increase the dynamic complexity of the program function.

To this must be added the very real fact that dynamic complexity, by almost any reasonable measure, grows exponentially with increasing static complexity. One need only examine the number of paths (and hence the number of component partial functions) in a program as a function of the number of non-nested IF statements or DO-WHILE loops to be convinced of this. Such complexity is reflected in testing methods in a number of ways. Testing strategies which attempt to test each component partial function of a program function will experience a discouragingly rapid growth in the amount of test data required. Testing methods which depend on useful properties or special classes of functions are less likely to be applicable to the program function than to its simpler component functions. Algebraic testing would require ever higher degree functions if a reasonable approximation to the correct program is to be guaranteed. Such increases often seem all out of proportion to the change in intuitive complexity of the program. RECIP, for example, has a relatively complex dynamic structure, but we perceive the code to be quite simple.

It is interesting, then, to compare the test points chosen for RECIP in the previous section under both static and dynamic criteria. The error space described in equation (2) represents our uncertainty in the dynamic behavior of the program, while the error space in (1) describes static alternatives to the expression in statement 6. If the points chosen for the black box test are employed to test the static form of statement 6, many of the executions of

that statement are rejected as unnecessary for testing under the criteria of Theorem 2. In fact, the static space described by (1) is reduced after only the first five points to the set of expressions in figure 9, all of which are obviously invariantly zero and hence are not truly errors. This means that, although the two error spaces involve terms of about the same degree, and even though the static space for this problem is initially much larger, the information obtained with those points about the static form of statement 6 is much larger than the information gained about the dynamic form of the program function. Had RECIP been dominated by IF statements rather than by a loop, the results might have been exactly opposite.

VI. Conclusions

The use of algebraic techniques in defining a neighborhood of functions is particularly suited to testing for computation errors. Two possible approaches are Howden's algebraic testing method and perturbation testing. Algebraic testing establishes rules for choosing data to differentiate among all members of a functional class, and then applies those rules to any program whose output is expected to fall within that class. Perturbation testing involves the derivation of those members of the chosen functional class which are indistinguishable from the program function using all test data chosen up to that moment. When applied to the output function of the module, these approaches are mathematically equivalent, but perturbation testing offers more flexibility in the choice of functional class.

The set of programs to which these techniques are applicable can be considerably expanded by changing the goal of the testing process, from verifying that the program function is the only member of a class with the

---

```
(XLAST - X)          (OLDDIF - DIFF)     (A*XLAST - A*X)
(X0*XLAST - X0*X)    (A*XLAST**2 - A*X**2)
```

Figure 9. Final Blindness Space for Statement 6

observed behavior on the test data, to determining whether the program function has been perturbed by the addition of an error term from a chosen functional class. This view also permits each method to offer a spectrum of testing reliability by varying the class of functions used as potential error terms.

A significant alternative offered by perturbation testing is the ability to work in the static domain, choosing test data to eliminate possible error terms in specific assignment and output statements. This approach promises savings due to the lower complexity of the code as compared to the dynamic program function. In addition, perturbation testing seems to be more easily interpretable, being expressed in terms of the actual changes in the code whose possible occurrence in the code has been checked.

Bibliography

1. T. A. Budd and W. C. Miller, Detecting Typographical Errors in Numerical Programs, Univ. of Arizona Technical Report TR82-14, 1982

2. T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing", Acta Informatica, 18, pp. 31-45, 1982

3. L. A. Clarke, J. Hassell, and D. J. Richardson, A Close Look at Domain Testing, IEEE Transactions on Software Engineering, vol. SE-8, no. 4, 380-390, July 1982

4. G. Dahlquist, and A. Bjork, Numerical Methods, 1974, Prentice-Hall

5. R. A. De Millo, F. G. Sayward, and R. J. Lipton, "Program Mutation: A New Approach to Program Testing", State of the Art Report on Program Testing, 1979, Infotech International

6. R. A. De Millo and R. J. Lipton, "A Probabalistic Remark on Algebraic Testing", Information Processing Letters, 7, June 1978

7. A. Gewirtz, H. Sitomer, and A. Tucker, Constructive Linear Algebra, 1974, Prentice-Hall Inc.

8. J. Gourlay, Theory of Testing Computer Programs, Ph.D. dissertation, 1981, Univ. of Michigan

9. W. E. Howden, "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers, vol. C-24, no. 5, 554-560, May 1975

10. W. E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, vol. SE-2, no. 3, 280-215, Sept. 1976

11. W. E. Howden, Elementary Algebraic Program Testing Techniques, UCSD Computer Science Technical Report No. 12, 1976

12. W. E. Howden, "Algebraic Program Testing", Acta Informatica, vol. 10, 53-66, 1978

13. S. Rapps and E. J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection", 6th International Conference on Software Engineering, 1982, pp. 272-278

14. J. H. Rowland and P. J. Davis, "On the Use of Transcendentals for Program Testing", Journal of the ACM, vol. 28, no. 1, Jan. 1981, pp. 181-190

15. L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, vol. SE-6, no. 3, 247-257, May 1980

16. S. J. Zeil, Selecting Sufficient Sets of Test Paths for Program Testing, Ph.D. dissertation, 1981, Ohio State University, also technical report OSU-CISRC-TR-81-10

17. S. J. Zeil, "Testing for Perturbations of Program Statements", IEEE Transactions on Software Engineering, SE-9, No. 3, May 1983, pp. 335-346