The Application of Error-Sensitive Testing Strategies to Debugging

Lori A. Clarke
Debra J. Richardson

COINS Technical Report 82-36
October 1982

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

This research was funded in part by the National Science Foundation under grant NSFMCS 81-04202.

Abstract

Error-sensitive test data selection strategies assist in the selection of test data that focus on the detection of particular types of errors. Traditionally, these strategies have been rather ad hoc. Recently formal testing methods have been developed which more rigorously apply the ideas underlying error-sensitive test data selection to the functional representation of a program provided by symbolic evaluation. This paper describes two such error-sensitive test data selection strategies, computation testing and domain testing. An approach for assisting in the debugging process, based on information about errors detected through the use of these strategies, is discussed.

Introduction

Program errors can be considered from two perspectives -- cause and effect. The goal of program testing is to detect errors by discovering their effects, while the goal of debugging is to search for the associated cause. In the last decade, there has been considerable work on tools and techniques to support the testing process. We are exploring ways in which the results of testing research can be used to assist in the debugging process.

Recently error-sensitive test data selection strategies have been developed, that attempt to assure the detection of certain classes of errors or provide quantifiable error bounds. This paper describes two of the more promising of these strategies, computation testing and domain testing, and proposes a comprehensive and rigorous set of guidelines for applying them. These strategies are based upon the functional representation of a program provided by symbolic evaluation. By selecting test data that focus on specific types of errors, these strategies may also be useful in revealing the cause of detected errors, thus assisting in the debugging process as well. This paper outlines ways in which symbolic evaluation and these test data selection strategies can be used as debugging aids.

The next section of this paper provides a brief overview of symbolic evaluation and an example is presented to demonstrate the technique. The third section describes the two test data selection strategies and, using the results from the symbolic evaluation process, applies each strategy to the example. The final section discusses how these strategies might be applied to the debugging process and outlines directions of future research.

Symbolic Evaluation

Symbolic evaluation provides a functional representation of the paths in a program or module. To create this representation, symbolic evaluation assigns symbolic names for the input values and evaluates a path by interpreting the statements on the path in terms of these symbolic names. During symbolic evaluation, the values of all variables are maintained as algebraic expressions in terms of the symbolic names. Similarly, the branch predicates for the conditional statements on a path are represented by constraints in terms of the symbolic names. After symbolically evaluating a path, its functional representation consists of the path computation, which is a vector of algebraic expressions for the output values (including the values returned by parameters) and the path domain, which is defined by the conjunction of the path's branch predicate constraints. For path P_J the path computation and path domain are denoted by $C[P_J]$ and $D[P_J]$, respectively.

Using symbolic evaluation, the path computation and path domain can be developed incrementally by interpreting each statement on a path. After symbolically evaluating a sequence of statements on a path, the symbolic representation of the path to that point can be shown. This representation consists of the current symbolic representation for each variable and the conjunction of the branch predicate constraints that have been created so far. This conjunction of constraints is called the path condition and is used to determine the feasibility of the path being examined. If, at any point during the symbolic evaluation, it can be determined that the path condition is infeasible -- that is, there are no data for which the sequence of statements could be executed -- then symbolic evaluation of that path can be terminated. Nonexecutable paths are a common phenomena in programs (especially unstructured programs). Of course, no method can determine the feasibility of any arbitrary path condition [DAVI73]. When path feasibility or infeasibility can not be determined, symbolic evaluation can still continue, but the selection of test data must be manually decided.

The procedure TRIANGLE, shown in Figure 1, is used to illustrate symbolic evaluation. Note that the left hand side of the listing is annotated with node numbers so that statements or parts of statements can easily be referenced. A path description is the ordered list of nodes encountered on the path.

Symbolic interpretation of the statements on a path P_J provides a symbolic representation of the path computation and path domain. The path computation $C[P_J]$ consists of the symbolic representation of the output values. The symbolic representation of the path domain $D[P_J]$ is provided by the path condition. Note that only the input values that satisfy the path condition could cause execution of the path. Figure 2 shows the symbolic representations of the path domains and path computations resulting from symbolic evaluation of all paths in TRIANGLE.

Unlike the TRIANGLE example, most programs contain loops. A symbolic representation of all executable paths through such a program is usually unreasonable since there may be a large, or even infinite, number of executable paths. One approach to this problem is to replace each loop with a closed form expression that captures the effect of that loop [CHEA79,CLAR81]. Using this technique, a path may then represent a class of paths that differ only by their number of loop iterations. While this is a powerful technique, it is not always successful. Other methods that guide in the selection of a subset of paths such as data flow testing [LASK79,NATF81,RAPP81], mutation analysis [DEMI79], and blindness testing [ZEIL81] are currently being explored but are not discussed further here. In the next section, it is assumed that a reasonably powerful method of path selection is being applied and the test data selection strategies are thus described for a selected set of paths.

```
   procedure TRIANGLE( A, B, C: in natural;
         CLASS: out integer; AREA: out real) is
   ASQRD, BSQRD, CSQRD: integer;
   S: real;
s  begin
1  if (A < B) or (B < C) then
      -- illegal input
2     CLASS := -1;
3     AREA := 0.0;
   else -- A >= B >= C
      -- legal input
4     if (A /= B) and (B /= C) then
         -- triangle is scalene
5        ASQRD := A*A;
6        BSQRD := B*B;
7        CSQRD := C*C;
8        if (ASQRD = BSQRD + CSQRD) then
            -- right triangle
9           CLASS := 3;
10          AREA := B * C / 2.0;
         else -- ASQRD /= BSQRD + CSQRD
            -- not right triangle
11          S := (A + B + C) / 2.0;
12          AREA := sqrt( S*(S-A)*(S-B)*(S-C) );
13          if (ASQRD < BSQRD + CSQRD) then
               -- acute triangle
14             CLASS := 4;
            else -- ASQRD > BSQRD + CSQRD
               -- obtuse triangle
15             CLASS := 5;
            endif;
         endif;
16    elsif (A = B) and (B = C) then
         -- equilateral triangle
17       CLASS := 1;
18       AREA := A*A*sqrt(3.0)/4.0;
      else -- (A /= B) or (B /= C)
         -- isosceles triangle
19       CLASS := 2;
20       if (A = B) then
21          AREA := C * sqrt(4*A*B-C*C) / 4.0;
         else -- B = C
22          AREA := A * sqrt(4*B**C-A*A) / 4.0;
         endif;
      endif;
   endif;
f  end TRIANGLE;
```

Figure 1. Procedure TRIANGLE

```
P1:       s,1,4,16,19,20,22,f
D[P1]:    (a - b > 0) and (b - c = 0)
C[P1]:    CLASS = 2
          AREA = a * sqrt(4.0*b**c - a**2) / 4.0

P2:       s,1,4,16,19,20,21,f
D[P2]:    (a - b = 0) and (b - c > 0)
C[P2]:    CLASS = 2
          AREA = c * sqrt(4.0*a*b - c**2) / 4.0

P3:       s,1,4,16,17,18,f
D[P3]:    (a - b = 0) and (b - c = 0)
C[P3]:    CLASS = 1
          AREA = a**2*sqrt(3.0)/4.0

P4:       s,1,4,5,6,7,8,11,12,13,15,f
D[P4]:    (a - b > 0) and (b - c > 0) and
                    (a**2 - b**2 - c**2 > 0)
C[P4]:    CLASS = 5
          AREA = sqrt((-a**4 + 2*a**2*b**2 +
                       2*a**2*c**2 - b**4 +
                       2*b**2*c**2 - c**4) / 16.0)

P5:       s,1,4,5,6,7,8,11,12,13,14,f
D[P5]:    (a - b > 0) and (b - c > 0) and
                    (a**2 - b**2 - c**2 < 0)
C[P5]:    CLASS = 5
          AREA = sqrt((-a**4 + 2*a**2*b**2 +
                       2*a**2*c**2 - b**4 +
                       2*b**2*c**2 - c**4) / 16.0)

P6:       s,1,4,5,6,7,8,9,10,f
D[P6]:    (a - b > 0) and (b - c > 0) and
                    (a**2 - b**2 - c**2 = 0)
C[P6]:    CLASS = 3
          AREA = b * c / 2.0

P7:       s,1,2,3,f
D[P7]:    ((a - b < 0) or (b - c < 0))
C[P7]:    CLASS = -1
          AREA = 0.0
```

Figure 2. Paths of TRIANGLE

Test Data Selection Strategies

A test data selection strategy should provide guidance in the selection of test data for a program. Ideally, executing the program on the selected data reveals errors in the program or provides confidence in the program's correctness. As noted, program testing detects an error by discovering the effect of that error. It is possible, however, that an error on an executed path may not produce erroneous results for some selected test case; this is referred to as coincidental correctness. For example, suppose that a computation z=a*2 is incorrect and should be z=a**2; if no test data other than a=0 or a=2 are selected, the error will not be detected. Although this appears to be a contrived example, coincidental correctness is a very real phenomenon. Test data selection strategies must address this problem.

The testing literature has classified errors into two types according to their effect on the path domains and path computations. If an incorrect path computation exists, a computation error is said to have occured. Such an error may be caused by an inappropriate or missing assignment statement that affects the function computed by the path. If a path domain is incorrect, a domain error is said to have occured. Domain errors can be further divided into path selection errors and missing path errors. A path selection error occurs when a program incorrectly determines the conditions under which a path is executed. This may be due to an incorrect conditional statement or an incorrect assignment statement that affects a conditional statement. A missing path error occurs when a special case requires a unique sequence of actions, but the program does not contain a corresponding path. This type of error is caused by missing conditional statements.

Error-sensitive test data selection strategies assist in the selection of test data that focus on the detection of particular types of errors. Moreover, such strategies minimize the acceptance of coincidentally correct results by astutely selecting test data aimed at exposing, not masking, errors. Error-sensitive test data selection has traditionally been rather ad hoc. Howden's functional testing [HOWD80], Myer's error guessing [MYER79], and Weyuker's error-based testing [WEYU81] are intuitive guidelines for selecting test data likely to expose commonly occuring errors. Each approach is based on an examination of the statements in a program or inspection of an informal description of the intent of the program. More rigorous application of the ideas underlying error-sensitive test data selection, which analyze the representations of the path domains and path computations provided by symbolic evaluations, have been developed. Computation testing strategies analyze the path computations and select test data aimed at revealing computation errors. Domain testing strategies concentrate on the detection of domain errors by analyzing the path domains and selecting test data near the

boundaries of those domains.

Computation Testing

Computation testing strategies focus on the detection of computation errors. Test data for which the path is sensitive to computation errors are selected by analyzing the symbolic representation of the path computation. In general, a path computation may contain arithmetic manipulations or data manipulations, which are inherently sensitive to different classes of computation errors.

Path computations containing predominately arithmetic manipulations are sensitive to errors relating to the use of numeric values and operators in arithmetic expressions. The following list provides guidelines for selecting test data for such computations, along with the class of computation errors the data is geared toward detecting:

1) all symbolic names in $C[P_J]$ take on distinct numeric values (erroneous reference to an input value);

2) each symbolic name corresponding to a multiplier, a divisor, and an exponent in $C[P_J]$ takes on
   a) the values zero, one, and negative one (erroneous processing of special input values),
   b) nonextremal positive and negative values (erroneous processing of typical values),
   c) extremal* positive and negative values (erroneous processing of atypical values or occurrence of overflow or underflow);

3) each term in $C[P_J]$ takes on
   a) the only zero value (a term masking an error in another term),
   b) the only non-zero value (enables independent evaluation of errors in a term),
   c) nonextremal positive and negative values (erroneous processing of typical values),
   d) extremal positive and negative values (erroneous processing of atypical values or occurrence of overflow or underflow);

4) each repetition count in a closed form expression in $C[P_J]$ takes on
   a) the value zero (erroneous processing of loop fall through),
   b) the value one (erroneous processing of single loop iteration),
   c) a nonextremal positive value (erroneous processing of typical loop traversal),
   d) an extremal positive value (erroneous processing of atypical loop traversal);

*A value of large magnitude often serves the purpose of an extremal value for unbounded values.

5) $C[P_J]$ takes on
   a) the value zero (erroneous production of special output values),
   b) nonextremal positive and negative values (erroneous production of typical output values),
   c) extremal positive and negative values (erroneous production of atypical output values or occurrence of overflow or underflow).

Figure 3 shows the test data selected for path P_1 of TRIANGLE using these computation testing guidelines. Notice first that for the test datum (3,2,2) coincidental correctness occurs. The comprehensive set of guidelines ensures that test data are also selected for which the computation error is revealed; the path computation should be CLASS=2 and AREA=a*sqrt(4.0*b*c-a**2)/4.0. In addition, a missing path error is detected with the test datum (2,1,1). The procedure fails to check whether the input values satisfy the triangle inequality (a<b+c); if not, they do not represent the sides of a triangle and the path computation should be CLASS=0 and AREA=0.0.

Path computations containing data manipulation typically maintain compound data structures and as a result are sensitive to errors that involve data movement operations rather than arithmetic operations. The following list provides guidelines for selecting test data for such computations, along with the class of computation errors the data is geared toward detecting:

1) all component selectors in $C[P_J]$ take on
   a) distinct values (erroneous interaction between different components),
   b) identical values (erroneous duplicate use of a component);

2) each component selector in $C[P_J]$ takes on
   a) a nonextremal value (erroneous processing of components in the midst of the structure),
   b) an extremal value (erroneous processing of components on the edge of the structure);

3) all components of a compound structure referenced in $C[P_J]$ take on
   a) distinct values (erroneous compound selector),
   b) identical values (erroneous processing of duplicate values);

4) the size of a compound structure referenced in $C[P_J]$ takes on
   a) nonextremal values (erroneous processing of typical structures)
   a) extremal values (erroneous processing of atypical structures or insufficient storage);

Criteria Satisfied	Test Data (a,b,c)	Actual Output (class,area)	Expected Output (class,area)
Computation Testing:			
1,2b,5b,3c	(3,2,2)	(2,1.98)	(2,1.98)
2a,5a	(2,1,1)	(2,0.0)	(0,0.0)
2b,3d,5c	(17,9,9)	(2,167305.38)	(2,25,14)
2c	(100,99,99)	overflow	(2,4272.29)
3d,5c	(11,10,10)	overflow	(2,41.76)
Domain Testing:			
on (b-c=0)	(2,1,1)	(2,0.0)	(0,0.0)
off (b-c=0)	(1,1,2)	(-1,0.0)	(-1,0.0)
off (b-c=0)	(2,2,1)	(2,0.97)	(2,0.97)
on (b-c=0)	(100,1,1)	overflow	(0,0.0)
off (b-c=0)	(100,1,2)	(-1,0.0)	(-1,0.0)
off (b-c=0)	(100,2,1)	overflow	(0,0.0)
on (b-c=0)	(100,99,99)	overflow	(2,4272.29)
off (b-c=0)	(99,99,100)	(-1,0.0)	(-1,0.0)
off (b-c=0)	(100,100,99)	(2,4301.02)	(2,4301.02)

Figure 3. Test data for TRIANGLE

5) a compound structure referenced in $C[P_J]$ takes on
   a) an empty value (erroneous initialization or processing of underflow)
   b) a full value (erroneous processing of overflow)

These guidelines are not applicable to the path computations in TRIANGLE, since they do not contain data manipulations.

A path computation may contain both arithmetic and data manipulations, in which case all applicable guidelines should be considered. It is important to note that the guidelines may not all be satisfiable due to the condition defining $D[P_J]$ or the representation of $C[P_J]$. In selecting test data for path P_1 of TRIANGLE, for example, several guidelines could not be satisfied due to the constraints that a, b, and c be positive and that b=c. These computation testing guidelines subsume those proposed by Howden [HOWD80] for special values testing and extremal output values testing, as well as the error-sensitive test case analysis proposed by Foster [FOST80].

When the path computations fall into specialized categories, the general computation testing guidelines can be tuned to guide in the selection of an even more comprehensive set of test data. For example, if a path computation involves trigonometric functions, then guidelines dependent upon their properties should be exploited. Polynomial functions are another category for which the guidelines can be refined. Under certain assumptions, it is possible to demonstrate the correctness of a polynomial path computation by means of testing. This is called polynomial testing and is based on algebraic results, which are applicable only when an upper bound on the algebraic complexity of the "correct" path computation is known. If the path computation $C[P_J]$ should be a univariate polynomial of maximal degree T-1, the selection of T linearly independent test points is sufficient to determine whether $C[P_J]$ is correct. If the path computaton $C[P_J]$ should be a multivariate polynomial in K input values of maximal degree T-1, $C[P_J]$ must be tested for T^K linearly independent test points in order to determine that it is correct [HOWD78b]. The practicality of polynomial testing is limited to polynomials in few variables and of low degree, since the number of test points required to determine correctness increases rapidly with the number of variables and the degree. Probabilistic arguments have been proposed for selecting fewer test points without sacrificing much accuracy [DEMI78].

Domain Testing

Domain testing is based on the observation that points satisfying boundary conditions are most sensitive to domain errors. A path selection error is manifested by a shift in some section of a path domain boundary. A missing path error typically corresponds to a missing path domain along some section of the boundary of an existing path domain. Missing path errors are particularly insidious, however,

since it is possible that only one point in a path domain should be in the missing path domain. In this case the error will not be detected unless that point happens to be selected for testing. Missing path errors cannot be found systematically unless a specification is employed by the test data selection strategy, as is done by the partition analysis method [RICH81].

The domain testing strategy [CLAR82 ,WHIT80] selects test data on and near the boundaries of each path domain. The boundary of a path domain is composed of borders with adjacent path domains. For each closed border, the strategy selects "on" test points, which lie on the border and thus in the path domain being tested, and "off" test points, which lie on the open side of the border and thus in an adjacent path domain. In such a way, domain testing attempts to detect border shifts, which occur when the border being tested is incorrect -- that is, it differs from the correct border. If the correct results are produced for each of the on and off test points, the border must be "close" to the correct border. An undetected border shift can only occur if the on test points and the off test points lie on opposite sides of the correct border. The undetectable border shifts are kept "small" by choosing the off test points as close to the border being tested as possible. In fact, with the proper selection of on and off test points, a quantified error bound measuring the set of elements placed in the wrong domain by an undetected border shift can be provided. Figure 4 illustrates a border shift, where G is the border being tested, C is the correct border, and the set of elements placed in the wrong domain is shaded. This border shift is revealed by testing the on points P and Q and the off points U and V, since the off point V is in the wrong domain. For a path domain border resulting from an inequality predicate in two-dimensions (two input values), the selection of four test data points (two on points and two off points) is most effective for detecting border shifts. For an inequality border in higher dimensions, 2*V (where V is the number of vertices of the border) test data points (V on points and V off points) must be selected for best results. For an equality border, twice as many off points, divided between the two sides of the border, must be selected. A thorough description of the domain testing strategy and its effectiveness is provided in [CLAR82]. Figure 3 shows the test data selected for path P_1 to satisfy the domain testing strategy. The only closed border of the path domain is (b-c=0), which has three vertices. The figure indicates whether each datum is an onpoint or an offpoint (above or below the equality border). Notice that several of the points selected reveal the missing path error, which is also detected by computation testing.

The basic domain testing strategy described is useful for testing path domain borders that involve both arithemetic manipulations and data manipulations in which the values of component selectors are known. Complications

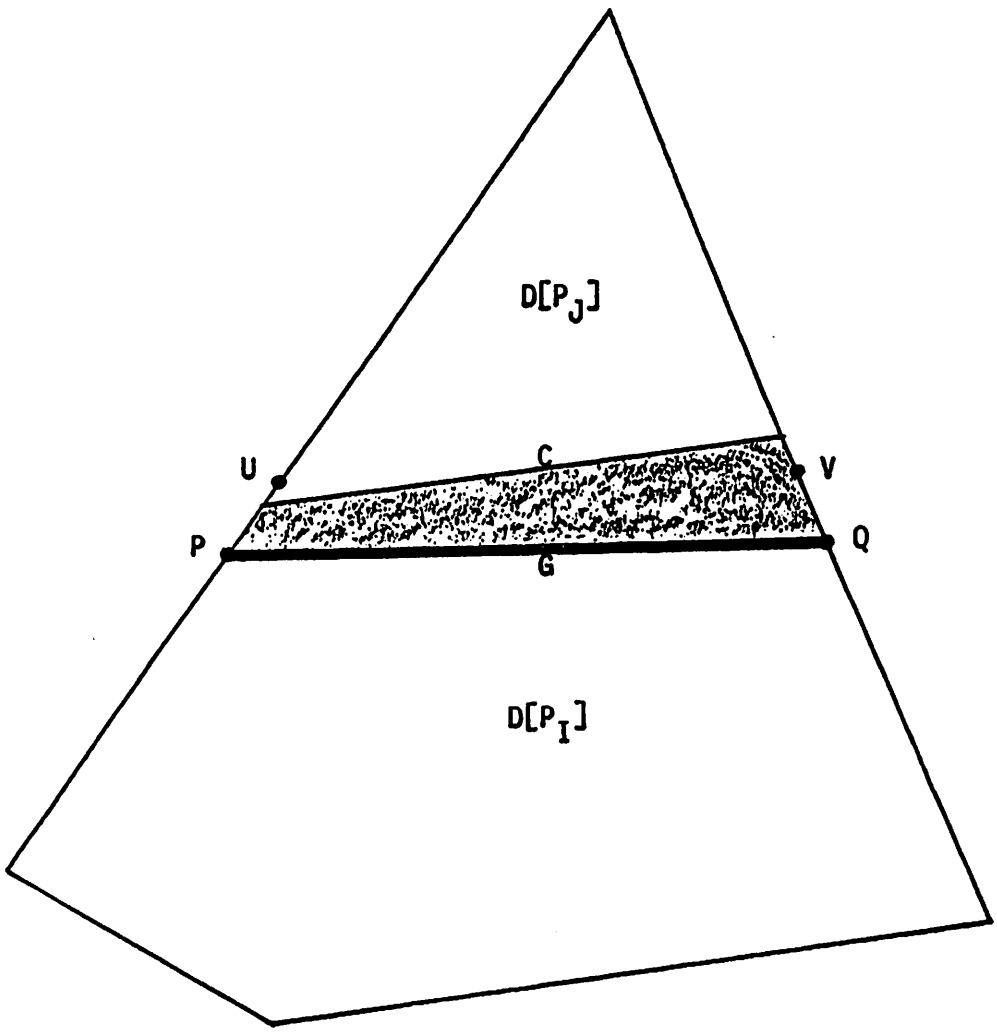

Figure 4. Border Shift Detected by Domain Testing

in applying the strategy arise when the values of component selectors depend on input values. Due to the dependencies among components of a compound structure and the component selectors, it may not be possible to find good on and off test points for a particular border. The intuitive concepts underlying domain testing can be used as heuristics to test the borders of a path domain. For instance, if a path domain border references a component of a compound structure with a selector of unknown value, it is important to test values both inside and just outside the domain for both the selector and the component. With the application of such heuristics, however, a bound on the error cannot be quantified.

The domain testing strategy subsumes both the boundary value testing and condition coverage guidelines proposed by Myers [MYER79] as well as the extremal input values testing proposed by Howden [HOWD80]. Domain testing is a relatively new test data selection strategy for which much further research is needed. The strategy has been well defined for domains that are continuous, linear convex polyhedra. This assumes that the input space is continous and that none of the branch predicate constraints contain a disjunction and all relational expressions are linear. Adequate modifications have been proposed for both nonconvex and discrete domains, although several problems remain to be addressed [CLAR82,WHIT80]. As yet, however, the strategy has only been sufficiently defined for linear borders. Modifications have been proposed that require the selection of on and off test points near each of the local minima and maxima of a nonlinear border. Unfortunately, the practical applicability of domain testing is limited to paths with non-linear constraints of low degree.

Applying Testing Information to the Debugging Process

It is obvious that much of the information obtained in the testing process would be valuable during debugging. Eventual software development environments should include testing and debugging tools that work cooperatively. This section explores how the information created by symbolic evaluation, computation testing, and domain testing can be applied to the debugging process. While some of the applications are fairly straightforward, others are the subject of current research.

The functional representation provided by symbolic evaluation can quite naturally be used during debugging. Symbolic testing [HOWD78a] refers to the method of manually examining the path computation and path condition to ascertain the correctness of the path. It must be noted that for some paths the complexity of these expressions is too great for this to be a reasonable exercise. On the other hand, there are many programs for which symbolic testing is effective [HOWD78a]. In a similar way, symbolic debugging can be defined as the process of manually

examining the path computation and path domain in order to obtain information about the cause of a known error, regardless of whether that error was detected by symbolic testing or another testing strategy.

It is anticipated that symbolic debugging will not be as susceptible to programmer oversight as is symbolic testing. The major weakness of symbolic testing is that programmers often overlook the errors. For example, the error in the algebraic expression for AREA on path P_1 of TRIANGLE could easily be overlooked. Once an error is known to exist, however, the programmer knows which algebraic expression is in error and can focus on that expression. The erroneous expression is then likely to provide clues to the actual cause of the error.

Another benefit of symbolic evaluation is that it provides information about the source of an error -- that is, the actual erroneous statements. It is often the case that programmers are not sure of the actual path that was taken when erroneous results are discovered. Symbolic evaluation provides a precise path description in the form of a list of statements on the path. Thus after symbolic debugging helps determine the cause of an error, the statements on the path can be examined to determine the actual source of that error.

To further focus on the possible source, symbolic evaluation systems could be modified to provide more explicit information concerning the statements that might have caused the error. For each algebraic expression in the path computation and constraint in the path condition, a list of the statements on the path that had any effect on that expression or constraint could be provided, rather than merely the path description. We refer to this as the expression's or constraint's definition list. When an erroneous expression or constraint is detected, only statements from the corresponding definition list need be examined. For the error in computing AREA on path P_1 in TRIANGLE, for example, only statement 22 would be on AREA's definition list; thus, the programmer is immediately directed to the source of the error. While definition lists will often be longer than one entry, they will usually be much shorter than the path description. Thus the number of statements that must be considered when tracking down the source of an error can be considerably reduced.

This proposed modification is relatively simple to implement. There are two techniques that could be used. The first technique would create the definition lists while a path is being interpreted. Using this technique, a definition list is associated with every variable. When a variable is assigned a value, the new definition list for the variable is the union of the definition lists of all variables referenced in the assignment statement plus the current statement. The definition list for a constraint is formed similarly. The second technique for deriving the definition list employs data flow methods to determine all the assignment statements that can affect a statement along

any path in a program. For a branch predicate or a statement that produces an output value, the intersection of the list of affecting statements with the statements in the path description provides the corresponding constraint's or expression's definition list. The advantage of this second technique is that the data flow analysis can be done for all statements very efficiently at compile time.

We know of two debugging systems that have used symbolic evaluation techniques to help determine the cause and find the source of program errors [BALZ69,FAIR75]. Neither of these systems provided the definition lists discussed above but both provided information about a path. At the time these systems were developed, they were too expensive to be used extensively. With the reduced cost and increased speed of hardware, such systems may now be more practical. Moreover, for programs in which reliablility is of utmost importance, the overhead of sophisticated testing strategies based on symbolic evaluation has already been warranted. For such programs, it seems only reasonable to employ the symbolic representations used for testing in the debugging process as well.

In addition to symbolic evaluation, the testing strategies also provide useful information to assist with the debugging process. Each test case generated by the computation testing or domain testing strategy could clearly state its goal -- that is, the potential error it is designed to uncover. If a test case results in erroneous output, then the stated goal often provides clues as to the actual cause of the error. For example, if input data is selected so that a term in a computation is the only non-zero term (criteria 3b in computation testing) and the resulting computation is in error, then this term should be examined carefully. Of course, this usually implies that each statement on the definition list for the erroneous algebraic expression be examined.

While including the test goal with each test case would be beneficial, care must be taken when evaluating this information. Programmers must keep in mind that computation and domain testing analyze the path computations and domains, which when erroneous represent the effect of an error and not necessarily its cause. Also, a test case may have several goals and only one may be indicative of the error. Several example situations that give an idea of some of the problems are given below.

Test data that uncovers a domain error does not necessarily imply that a branch predicate is in error. The constraint resulting from interpretation of the predicate was erroneous for this test case, but the actual source of the error could have been an assignment that affected this interpretation. The definition list associated with the erroneous constraint would, of course, be of value during debugging.

Just like domain errors can be caused by faulty assignment statements, computation errors can be caused by faulty branch predicates. For example, if a loop index is off by one, the branch predicate for loop termination is wrong. The effect of this erroneous predicate, however, may be an erroneous computation. There are several other situations in which computation errors are particularly elusive. For example, a test case may produce an erroneous computation, but examination of the algebraic expression reveals that this computation is correct for most of the associated path domain but not for this particular test case. In testing terminology, this could be either type of domain error, a path selection error or a missing path error. For a path selection error one or more constraints are in error, and the test point in question executed the wrong path. For a missing path error, a special case was completely forgotten by the programmer, and the testing process was astute enough to select a test case for the missing path.

The problems in applying testing information to debugging that are discussed above are further complicated when errors interact. More than one error may be present and in some cases their interaction may complicate tracking down the source of the errors. In fact, the domain testing strategy is not guaranteed to work when the computations are incorrect for the selected test cases. There is some recent work on path selection that addresses the problems of interacting errors [ZEIL81] that should also be considered with regard to debugging.

In sum, it is clear that more work needs to be done to develop debugging methods that employ information derived during the testing process. This paper has focused on only some testing techniques, namely symbolic evaluation, computation testing, and domain testing; other testing strategies should also be considered. Incorporating information derived from pre-implementation descriptions, such as designs or formal specifications, is currently being investigated as an aid to testing [RICH81] and may also be helpful for debugging. Moreover, several test path selection strategies [ZEIL81,RAPP81,NTAF81,LASK79] may be very valuable in assisting in locating the source of errors. Thus, despite the problems noted above, there appears to be considerable potential, without much additional overhead, for incorporating testing information into the debugging process.

References

[BALZ69] R.M. Balzer, "EXDAMS -- Extendable Debugging and Monitoring System," 1969 Spring Joint Computer Conference, AFIPS Conference Proceedings, 34, AFIPS Press, Montvale, New Jersey, 576-580.

[BOYE75] R.S. Boyer, B. Elspas, and K.N. Levitt, "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution," Proceedings of the International Conference on Reliable Software, April 1975, 234-244.

[CHEA79] T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs," IEEE Transactions on Software Engineering, SE-5,4, July 1979, 402-417.

[CLAR78] L.A. Clarke, "Automatic Test Data Selection Techniques," Infotech State of the Art Report on Software Testing, 2, September 1978, 43-64.

[CLAR81] L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods -- Implementations and Applications," Computer Program Testing, North-Holland Publishing Co., B.Chandrasekaran and S.Radicchi (eds.), 1981, 65-102.

[CLAR82] L.A. Clarke, J. Hassell, and D.J. Richardson, "A Close Look at Domain Testing," IEEE Transactions on Software Engineering, SE-8, 4, July 1982, 380-390.

[DAVI73] M. Davis, "Hilbert's Tenth Problem is Unsolvable," American Mathematics Monthly, 80, March 1973, 233-269.

[DEMI78] R.A. DeMillo and R.J. Lipton, "A Probablistic Remark on Algebraic Program Testing," Information Processing Letters, 7, June 1978.

[DEMI79] R.A. DeMillo, 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.

[FAIR75] R.E. Fairley, "An Experimental Program-Testing Facility," IEEE Transactions on Softlware Engineering, SE-1,4 December 1975, 350,357.

[FOST80] K.A. Foster, "Error Sensitive Test Case Analysis (ESTCA)," IEEE Transactions on Software Engineering, SE-6, 3, May 1980, 258-264.

[HOWD75] W.E. Howden, "Methodology for the Generation of Program Test Data," IEEE Transactions on Computer, C-24,5, May 1975, 554-559.

[HOWD78a] W.E.Howden, "An Evaluation of the Effectiveness of Symbolic Testing," Software: Practice and Experience, 10, July-August 1978, 381-397.

[HOWD78b] W.E. Howden, "Algebraic Program Testing," ACTA Informatica, 10, 1978.

[HOWD80] W.E. Howden, "Functional Program Testing," IEEE Transactions on Software Engineering, SE-6,2, March 1980, 162-169.

[HUAN75] J.C. Huang, "An Approach to Program Testing," ACM Computing Surveys, 7,3, September 1975, 113-128.

[LASK79] J.W. Laski, "A Hierarchical Approach to Program Testing," Department of Systems Design, University of Waterloo, Waterloo, Ontario, Canada, Technical Report No.55CFW130779.

[MYER79] G.J. Myers, The Art of Software Testing, John Wiley & Sons, New York, New York, 1979.

[NTAF81] S.C. Ntafos, "On Testing With Required Elements," Proceedings of COMPSAC '81, November 1981, 132-139.

[RAPP81] S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection," Computer Science Department, New York University, New York, New York, Technical Report No.023, December 1981.

[RICH81] D.J. Richardson and L.A. Clarke, "A Partition Analysis Method to Increase Program Reliability," Fifth International Conferene on Software Engineering, March 1981, 244-253.

[WEYU81] E.J. Weyuker, "An Error-Based Testing Strategy," Computer Science Department, New York University, New York, New York, Technical Report No.027, January 1981.

[WHIT80] L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Transactions on Software Engineering, SE-6, May 1980, 247-257.

[ZEIL81] S.J. Zeil and L.J. White, "Sufficient Test Sets for Path Analysis Testing Strategies," Proceedings of the Fifth International Conference on Software Engineering, 1981, 184-191.