A RIGOROUS APPROACH TO ERROR-SENSITIVE TESTING*

Lori A. Clarke
Debra J. Richardson

COINS Technical Report 82-28
November 1982

*This paper appears in the Proceedings of the 16th Hawaii International Conference on System Sciences.

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

A RIGOROUS APPROACH TO ERROR-SENSITIVE TESTING

Lori A. Clarke
Debra J. Richardson
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Error-sensitive testing 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 strategies have been developed that more rigorously apply the ideas underlying error-sensitive testing by using the functional representation of a program provided by symbolic evaluation. This paper describes two such error-sensitive strategies, computation testing and domain testing. An approach to integrating these strategies, and the possibility of automating this approach are discussed.

1. INTRODUCTION

As was evident at the workshop on the Effectiveness of Testing and Proving Methods (16), there are a number of exciting research projects addressing the problems of software testing. For the most part, recent research has moved from developing tools that gather information about programs to developing techniques that actually apply this information. Moreover, there is a deeper awareness and understanding of the theoretical limitations of the techniques that are being developed. While many of these techniques are quite complex, the wide availability of increased computing power makes the actual realization and everyday utilization of such techniques an imminent possibility. A testing method, composed of an integrated and well understood set of testing techniques, could be developed that provides considerable guidance in the testing process.

For the most part, current testing research is directed at either the problem of determining the paths, the particular sequences of statements that must be tested, or the problem of selecting revealing test data for the selected paths. For the path selection problem, techniques such as data flow testing (11,13,14), perturbation testing (8,19), and mutation testing (6) have been proposed. For the test data selection problem, a number of informal guidelines (6,7,10) have been put forth. Recently there has been considerable work in developing more rigorous test data selection strategies that can either eliminate certain classes of errors or provide quantifiable error bounds. In this paper, we assume a reasonable method of path selection is available and concentrate on the test data selection aspects of testing. Two of the more rigorous error-sensitive testing strategies, computation testing (5,9) and domain testing (3,18), are described. By expanding on these strategies, a comprehensive and rigorous set of guidelines, based upon the functional representation of a program provided by symbolic evaluation, are proposed. Moreover, these guidelines consider both arithmetic and data manipulation errors.

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. Incorporating either one of these strategies into a testing method would inprove its error detection capabilities. An integrated approach to test data selection that combines both strategies would be even better. Thus, the final section discusses integrating these strategies and the possibilities of automating such an approach.

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

2. 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 PJ the path computation and path domain are denoted by C[PJ] and D[PJ], respectively.

The forward expansion method is the most straightforward and efficient way to do symbolic evaluation (2) and thus is the method described here. Using forward expansion the path computation and path domain are 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 (4). 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 RECTANGLE, 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. Paths are designated by the ordered list of nodes encountered on the path. Figure 2 shows the symbolic evaluation of the path (s,1,3,4,5,6,7,8,9,6,10,f).

Before interpretation of a path, the path condition is initialized to the value true and the values of all variables are set to their initial values: the input parameters are assigned

```
procedure RECTANGLE (A,B: in digits 8;
          H: in digits 3 range -1.0..1.0;
          F: in array [0..2] of digits 8;
          AREA: out digits 8;
          ERROR: out boolean) is
  -- RECTANGLE approximates the area under the
  -- quadratic equation F[0] + F[1]*X + F[2]*X**2
  -- from X=A to X=B in increments of H.

     X,Y: digits 8;
 s   begin
 1      if H > B - A then
 2         ERROR := true;
        else
 3         ERROR := false;
 4         X := A;
 5         AREA := F[0] + F[1]*X + F[2]*X*2;
 6         while X + H <= B loop
 7            X := X + H;
 8            Y := F[0] + F[1]*X + F[2]*X**2;
 9            AREA := AREA + Y;
           end loop;
10         AREA := AREA*H;
        endif;
 f  end RECTANGLE;
```

Figure 1.
Procedure RECTANGLE

symbolic names, variables that are initialized before execution are assigned their corresponding constant value, and all other variables are assigned the undefined value "?". Thus, before symbolically evaluating a path in RECTANGLE, the variables would be set to the initial values specified for node s in Figure 2, where variable names are written in upper case and symbolic names in lower case.

After initiallizing the variables and path condition, each statement is interpreted, as it is encountered on the path, by substituting the curent symbolic value of a variable wherever that variable is referenced. Thus, for the assignment statement at node 5 in RECTANGLE, the current symbolic values of X and F after interpretation of statements (s,1,3,4) are substituted into the expression on the righthand side, resulting in a*f[1]+2.0*a*f[2]+f[0] being assigned to AREA. If AREA is subsequently referenced on the path, then this new value would be substituted for AREA. For a conditional statement, the branch predicate corresponding to the selected path is interpreted. Thus when evaluating node 1, the branch predicate representing the condition to go from node 1 to node 3 is the complement of the condition at node 1. This evaluated branch predicate is first simplified and then conjoined to the previously generated path condition, resulting in the path condition

true and not(h > b-a) = (a-b+h >= 0.0)

Symbolic interpretation of the statements on a path PJ provides a symbolic representation of the path computation and path domain. The path computation C[PJ] consists of the symbolic representation of the output values. The symbolic

```
s  A=a
   B=b
   H=h
   F=f
   AREA=?
   ERROR=?
   X=?
   Y=?
   PC=true

1  PC=true and not (h>b-a)
     ==(a-b+h<=0.0)

3  ERROR=false

4  X=a

5  AREA=f[0] + f[1]*a + f[2]*a*2
       ==a*f[1] + 2.0*a*f[2] + f[0]

6  PC=(a-b+h<=0.0) and (a+h<=b)
     ==(a-b+h<=0.0)

7  X=a + h

8  Y=f[0] + f[1]*(a+h)+f[2]*(a+h)**2
    ==a*f[1] + a**2*f[2] + 2.0*a*f[2]*h
      + f[0] + f[1]*h + f[2]*h**2

9  AREA = a*f[1] + 2.0*a*f[2] + f[0]
         + a*f[1] + a**2*f[2] + 2.0*a*f[2]*h
         + f[0] + f[1]*h + f[2]*h**2
       == 2.0*a*f[1] + 2.0*a*f[2]
          + a**2*f[2] + 2.0*a*f[2]*h
          + 2.0*f[0] + f[1]*h + f[2]*h**2

6  PC = (a-b+h<=0.0) and not (a+h+h<=b)
     ==(a-b+h<=0.0) and (a-b+2.0*h > 0.0)

10 AREA = (2.0*a*f[1] + 2.0*a*f[2]
          + a**2*f[2] + 2.0*a*f[2]*h
          + 2.0*f[0] + f[1]*h + f[2]*h**2) * h
       == 2.0*a*f[1]*h + 2.0*a*f[2]*h
          + a**2*f[2]*h + 2.0*a*f[2]*h**2
          + 2.0*f[0]*h + f[1]*h**2 + f[2]*h**3


D: (a-b+h <= 0.0) and (a-b+2.0*h > 0.0)

C: ERROR = false
   AREA = 2.0*a*f[1]*h + 2.0*a*f[2]*h
        + a**2*f[2]*h + 2.0*a*f[2]*h**2
        + 2.0*f[0]*h + f[1]*h**2 + f[2]*h**3
```

Figure 2.
Symbolic Evaluation of RECTANGLE

representation of the path domain D[PJ] 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 domain and path computation resulting from symbolic evaluation of the path (s,1,3,4,5,6,7,8,9,6,10,f) in RECTANGLE.

A symbolic representation of all executable paths through RECTANGLE is unreasonable since there is an effectively infinite number of executable paths. This problem exists for any program in which the number of iterations of a loop is dependent on unbounded input values. One approach to this problem is to replace each loop with a closed form expression that captures the effect of that loop (1,2). 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. It can, however, be successfully applied to RECTANGLE. Figure 3 shows the symbolic representations of the domains and computations of the classes of paths in RECTANGLE. Note the P3 represents all paths that traverse the loop; P2 represents the fall through case of the loop, which is infeasible; P1 represents the case in which the input data is erroneous.

```
  P1 : (s,1,2,f)
D[P1] : (a-b+h > 0.0)
C[P1] : AREA=?
        ERROR=true

  P2  : (s,1,3,4,5,6,10,f)
D[P2] : (a-b+h <= 0.0) and (a-b+h > 0.0)
        == false ***infeasible path ***

  P3  : (s,1,3,4,5,(6,7,8,9),10,11,f)
D[P3] : (a-b+h <= 0.0)
C[P3] : AREA = a*f[1]*h + 2.0*a*f[2]*h + f[0]*h
             + sum<i:=1..int(-a/h+b/h) |
               (a*f[1]*h + a**2*f[2]*h
               + 2.0*a*f[2]*h**2*i + f[0]*h
               + f[1]*h**2*i + f[2]*h**3*i**2)>
        ERROR = false
```

Figure 3.
Path Domains and Computations for RECTANGLE

3. 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. In general, 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 data; 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 that test data selection strategies must address.

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 occurred. 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 occurred. 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 testing 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 testing has traditionally been rather ad hoc. Foster's test case analysis (7), Howden's functional testing (10), Myer's error guessing (12), Weyuker's error-based testing (17), and the test data selection aspects of mutation testing (6) provide intuitive guidelines for selecting test data likely to expose commonly occurring 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 testing, 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.

In RECTANGLE there are four errors, one computation error and three missing path errors. The first error is caused by an erroneous computation at statement 5; statement 5 should be AREA := F[0]+F[1]*X+F[2]*X**2. The second error occurs when h=0 and b>a and results in an infinite loop. The third error occurs when the sign of h is different than the sign of b-a -- that is when a>b and h>0 or when a<b and h<0. The fourth error occurs when a+int(-a/h+b/h)*h < b since the area under the quadratic is computed beyond the point specified by b. In the ensuing discussion it is shown how the application of computation and domain testing strategies to RECTANGLE detects each of these four errors.

3.1. Computation Testing

Computation testing is based on the assumption that the way an input value is used within the path computation is indicative of a class of potential computation errors. Analysis of the symbolic representation of the path computation reveals the manipulations of the input values that have been performed to compute the output values. In general, a path computation may contain arithmetic manipulations or data manipulations, which are inherently sensitive to different classes of computation errors and thus two sets of guidelines are provided.

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[PJ] 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[PJ] 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[PJ] 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[PJ] 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[PJ] 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 underflow or overflow).

Figure 4a describes the test data needed to satisfy these computation testing criteria for the class of paths P3 of RECTANGLE. The criteria are not applicable for the trivial computation of path P1. Note that epos and eneg represent extremal positive values and extremal negative values, respectively, for the symbolic name that appears as arguments. In some instances, extremal values are determined by the program -- e.g., epos(h)=1.0 and eneg(h)=-1.0. When an input is unbounded, however, the tester must specify extremal values to be used. In RECTANGLE, for instance, the extremal values we selected for a, b, and f are

---

Conditions to satisfy criterion 1
(a≠b,h,f,[0],f[1],f[2])
and (b≠h,f[0],f[1],f[2])
and (h≠f[0],f[1],f[2])
and (f[0]≠f[1],f[2])
and (f[1]≠f[2])

Conditions to satisfy criterion 2a
a=0.0,-1.0,1.0
b=0.0,-1.0,1.0
h=0.0,-1.0,1.0
f[0]=0.0,-1.0,1.0
f[1]=0.0,-1.0,1.0
f[2]=0.0,-1.0,1.0

Conditions to satisfy criterion 2b
a=tpos(a),tneg(a)
b=tpos(b),tneg(b)
h=tpos(h),tneg(h)
f[0]=tpos(f[0]),tneg(f[0])
f[1]=tpos(f[1]),tneg(f[1])
f[2]=tpos(f[2]),tneg(f[2])

Conditions to satisfy criterion 2c
a=epos(a),eneg(a)
b=epos(b),eneg(b)
h=epos(h),eneg(h)
f[0]=epos(f[0]),eneg(f[0])
f[1]=epos(f[1]),eneg(f[1])
f[2]=epos(f[2]),eneg(f[2])

Conditions to satisfy criterion 3a
f[1]=0.0 and a,h,f[0],f[2]≠0.0
f[2]=0.0 and a,h,f[0],f[1]≠0.0
f[0]=0.0 and a,h,f[1],f[2]≠0.0

Conditions to satisfy criterion 3b
a,f[1],h≠0.0 and f[0]=f[2]=0.0
a,f[2],h≠0.0 and f[0]=f[1]=0.0
f[0],h≠0.0 and f[1]=f[2]=0.0
f[1],h≠0.0 and a=f[0]=f[2]=0.0
f[2],h≠0.0 and a=f[0]=f[1]=0.0

Conditions to satisfy criterion 3c
a=tpos(a) and f[1]=tpos(f[1]) and h=tpos(h)
and f[0]=f[2]=0.0
a=tneg(a) and f[1]=tneg(f[1]) and h=tneg(h)
and f[0]=f[2]=0.0
a=tneg(a) and f[2]=tpos(f[2]) and h=tpos(h)
and f[0]=f[1]=0.0
a=tneg(a) and f[2]=tneg(f[2]) and h=tneg(h)
and f[0]=f[1]=0.0
f[0]=tpos(f[0]) and h=tpos(h) and f[1]=f[2]=0.0
f[0]=tneg(f[0]) and h=tneg(h) and f[1]=f[2]=0.0

Conditions to satisfy criterion 3d
a=epos(a) and f[1]=epos(f[1]) and h=epos(h)
and f[0]=f[2]=0.0
a=eneg(a) and f[1]=eneg(f[1]) and h=eneg(h)
and f[0]=f[2]=0.0
a=eneg(a) and f[2]=epos(f[2]) and h=epos(h)
and f[0]=f[1]=0.0
a=eneg(a) and f[2]=eneg(f[2]) and h=eneg(h)
and f[0]=f[1]=0.0
f[0]=epos(f[0]) and h=epos(h) and f[1]=f[2]=0.0
f[0]=eneg(f[0]) and h=eneg(h) and f[1]=f[2]=0.0

Conditions to satisfy criterion 4a
infeasible

Conditions to satisfy criterion 4b
b - a = h

Conditions to satisfy criterion 4c
b - a = 10*h

Conditions to satisfy criterion 4d
b - a = 100*h

Conditions to satisfy criterion 5a
h = 0.0

Conditions to satisfy criterion 5b
a=tpos(a) and b=tpos(b) and h=tpos(h)
and f[0]=tpos(f[0]) and f[1]=tpos(f[1])
and f[2]=tpos(f[2])
a=tneg(a) and b=tneg(b) and h=tneg(h)
and f[0]=tneg(f[0]) and f[1]=tneg(f[1])
and f[2]=tneg(f[2])

Conditions to satisfy criterion 5c
a=eneg(a) and b=epos(b) and f[0]=epos(f[0])
and f[1]=epos(f[1]) and f[2]=epos(f[2])
and h>0
a=eneg(a) and b=epos(b) and f[0]=eneg(f[0])
and f[1]=eneg(f[1]) and f[2]=eneg(f[2])
and h>0

NOTE: epos and eneg indicate extremal positive and negative values, respectively, for the symbolic name that appears as an argument. Likewise, tpos and tneg indicate typical (nonextremal) positive and negative values, respectively, for the symbolic name that appears as an argument.

Figure 4a.
Conditions for Satisfying
Computation Testing Guidelines for
Arithmetic Manipulations in RECTANGLE

epos(a) = epos(b) = 100.0, eneg(a) = eneg(b) = -100.0, epos(f[0]) = epos(f[1]) = epos(f[2]) = 10.0, eneg(f[0]) = eneg(f[1]) = eneg(f[2]) = -10.0.

The test data described in Figure 4a would reveal error one in RECTANGLE. Notice that statement 5 may produce coincidentally correct results for some of the selected test data but the error is guaranteed to be detected by the data satisfying criterion 2a. The second error is also guaranteed to be detected by this criterion. While there is a reasonable chance that data selected to satisfy the computation guidelines will detect errors three and four, these guidelines do not guarantee that these errors will be revealed.

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[PJ] 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[PJ] takes on
   a) a nonextremal value (erroneous processing of components in the midst of the structure),
   b) extremal value (erroneous processing of components on the edge of the structure);

3) all components of a compound structure referenced in C[PJ] 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[PJ] takes on
   a) nonextremal values (erroneous processing of typical structures)
   a) extremal values (erroneous processing of atypical structures or insufficient storage);

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

Only criterion 3 is applicable to the path computations in RECTANGLE, since each component selector for the array F is a constant. Figure 4b describes the test data to satisfy this criterion. Since none of the errors in RECTANGLE involve F, none of the errors are detected by this criterion.

Conditions to satisfy criterion 3a
f[0]≠f[1],f[2]
f[1]≠f[2]

Conditions to satisfy criterion 3b
f[0]=f[1]=f[2]

Figure 4b.
Conditions for Satisfying
Computation Testing Guidelines for
Data Manipulations in RECTANGLE

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[PJ] or the representation of C[PJ]. In selecting test data for RECTANGLE, for example, guidelines 3a and 3b could not be completely satisfied due to the relationships between the terms in the computation. These computation testing guidelines subsume those proposed by Howden (10) for special values testing and extremal output values testing, as well as the error-sensitive test case analysis proposed by Foster (7).

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. In RECTANGLE an example for which an extended set of guidelines are required is the INT function that appears in the computation of AREA. Data should be selected so that the dropped remainder that results from applying the INT function is both zero and nonzero. Data satisfying this extension would reveal the fourth error.

Polynomial functions are a category for which the guidelines have been refined (9). 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, applicable only when an upper bound on the algebraic complexity of the "correct" path computation is known. If the path computation C[PJ] should be a univariate polynomial of maximal degree T-1, the selection of T linearly independent test points is sufficient to determine whether C[PJ] is correct. If the path computaton C[PJ] should be a multivariate polynomial in K input values of maximal degree T-1, C[PJ] must be tested for TK linearly independent test points in order to determine that it is correct (11). The practicaltiy 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.

3.2. Domain Testing

Domain testing is concerned with detecting both path selection errors and missing path errors. Proposed domain testing strategies emphasize selecting test points near the domain boundaries. A path selection error is manifested by a shift in some section of a path domain boundary and hence are likely to be caught by such strategies. Missing path errors, however, are particularly difficult to detect 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 (15). Missing path errors often correspond to a missing path domain that is near a boundary of an existing path domain, and thus these errors may be caught by existing domain testing strategies.

One proposed domain testing strategy (3,18) 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 5 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 (3). Figure 6 shows

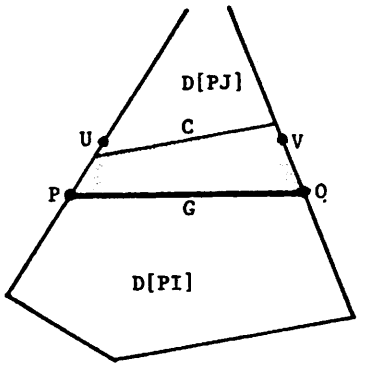

Figure 5.
Border Shift Detected by
Domain Testing Strategy

Conditions for on points for (a-b+h<=0.0)
a=100.0 and b=99.0 and h=-1.0
a=99.0 and b=100.0 and h=1.0
a=100.0 and b=100.0 and h=1.0
a=-100.0 and b=-99.0 and h=1.0
a=-100.0 and b=-100.0 and h=0.0
a=-99.0 and b=-100.0 and h=-1.0

Conditions for off points for (a-b+h<=0.0
a=100.01 and b=98.99 and h=-0.99
a=99.01 and b=99.99 and h=1.0
a=100.01 and b=99.99 and h=0.01
a=-99.99 and b=-99.01 and h=1.0
a=-99.99 and b=-100.01 and h=0.01
a=-98.99 and b=-100.01 and h=-0.99

Figure 6.
Conditions for Satisfying
Domain Testing Strategy
for RECTANGLE

the test data selected for the paths in RECTANGLE to satisfy the domain testing strategy. The only closed border of the path domain is (a-b+h >= 0.0). If extremal values of 100.0 and -100.0 are assumed for the inputs A and B, this border has six vertices. The figure indicates whether each datum is an on point or an off point (on or above the border). The first three errors in RECTANGLE are revealed by domain testing. Error one is detected by execution of any of the on points. Error two is detected by test data satisfying either of the two conditions (a = 100.0 and b = 100.0 and h = 0.0) or (a = -100.0 and b = -100.0 and h = 0.0). Error three is detected by test data satisfying either of the two conditions (a = 100.01 and b = 99.99 and h = 0.01) or (a = -99.99 and b = -100.01 and h = 0.01).

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 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. When only such heuristics are applicable, however, a bound on the error cannot be quantified.

The domain testing stategy subsumes both the boundary value testing and condition coverage guidelines proposed by Myers (12) and the extremal input values testing proposed by Howden (10). 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 interpreted branch predicates 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 (3,18). 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 interpreted branch predicates of low degree.

4. AN INTEGRATED APPROACH

Combining the computation and domain testing strategies results in the selection of data that more rigorously test a path than other strategies proposed to date. In fact, it could be argued that combining these two strategies will result in a test data selection method that is likely to detect all errors on the paths to which it is applied. There are two major reasons for this optimism. First, both strategies astutely select test data with the intent of detecting errors. Second, these strategies select a large number of test points, which are scattered throughout a path domain, so that even missing path errors are likely to be caught.

There are two major drawbacks, however, associated with combining these strategies. First, both strategies often produce a prohibitively excessive number of test points. Second, selecting data to satisfy these strategies is often a complex, ill-defined process. This section discusses the need to develop an effective procedure for integrating these strategies in which the overall number of test points is substantially reduced without a corresponding loss in reliability. Also, the possibility of providing automated support for the test data selection process is explored.

When considering the number of test points associated with either strategy, it is important to note that many of the test data that satisfy one selection criterion also satisfy others. This overlap occurs within a strategy as well as between the two strategies. When applying computation testing to the RECTANGLE example, each of the test data selected to satisfy criterion 3c satisfy several of the conditions for criterion 2b, and likewise for 3d and 2c. In addition, each of the on points chosen by the domain testing strategy also satisfy the computation testing criteria 2c and 3d.

Although the guidelines outlined for both computation and domain testing are fairly well-defined, a systematic procedure for actually selecting the data needs to be derived. Such a procedure could be designed so as to maximize the number of criteria satisfied by each selected data point, thereby minimizing the total number of selected test points. It is improbable, however, that the cost of finding a minimal set of test data would be cost-effective. A heuristic approach, which exploits the overlap among the criteria in an attempt to reduce the number of selected points, should certainly be developed. In defining an algorithm for combining computation and domain testing, another factor that should be considered is the flexibility of the selection criteria. For example, the domain testing criteria are generally very explicit about the test data, whereas some of the computation criteria can be satisfied by a number of different data points. Thus, domain testing criteria should probably be satisfied before the computation testing criteria are considered.

Another consideration for reducing the number of test points is to provide various levels of testing. Only the highest level of testing would require that all the test data selection criteria be satisfied. Life critical software would utilize this testing level but less critical software could utilize lower levels. In developing these testing levels, some of the more costly criteria would only be associated with the higher testing levels. Moreover, both computation and domain testing can be applied more or less rigorously, with the expected effect on cost and reliability. For example, probabilistic arguments have been made for greatly reducing the number of test points that must be selected for polynomial testing without sacrificing too much confidence (5). Similarly, a weaker version of domain testing, requiring considerably fewer test points, has also been evaluated (3). The development of testing levels must consider the cost and cost benefits associated with each test data selection criteria. Moreover, these testing levels should also be associated with appropriate path selection strategies. It is not reasonable to pair a weak path selection criterion, such as statement coverage, with the highest level of test data selection. Finally, care must be taken not to create a proliferation of levels. Testers do need a few well differentiated choices however.

Even if well defined procedures are available for satisfying a testing level, automated support for the testing process is required. Evidence supports this need, since even a weak testing criterion such as statement coverage is difficult to achieve without a tool to monitor program coverage. Symbolic evaluation tools provide a symbolic representation of the program, but automatic support for path selection and test data selection are needed as well. While these tasks can not always be completely automated, the need for human interaction can be minimized. Moreover, bookkeeping support tools are needed to keep track of all the information, such as stubs, drivers, input/output pairs, and test results, associated with a large testing endeavor. In some instances, specifications describing the expected output can be utilized so that the results from testing a program can be automatically verified. As is evident by the guidelines provided for computation and domain testing, reliable testing is a complex process. It is unrealistic to expect to achieve a reliable level of testing without providing programmers with appropriate evaluation and bookkeeping tools to support this process.

In addition to investigating the integration and automation of test data selection, some theoretical and experimental evaluations should be undertaken. While some of the test data selection criteria have been carefully investigated, others are only heuristics. Some criteria may subsume others and the interaction among some criteria is not well understood. As with polynomial testing, more reliable criteria can be developed for other well defined classes of computations, such as boolean expressions. Experimental studies evaluating the actual effectiveness of these strategies for detecting errors are also needed. It would be beneficial to experimentally evaluate the different levels of testing so that reliability measures can be associated with each level. For example, the expected meantime between failures or expected ratio of remaining errors to statements would be useful statistics that would help managers choose the appropriate testing level for a program.

In sum, this paper provides a description of test data selection strategies aimed at detection of computation and domain errors. Combined they provide a strong basis for a reliable test data selection method. There still remain several unanswered questions on how to refine, integrate, automate, and evaluate the test data selection process.

6. REFERENCES

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

2. 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.

3. 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.

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

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

6. 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.

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

8. A. Haley and S. Zweben, "Development and Application of a White Box Approach to Integration Testing," Workshop on Effectiveness of Testing and Proving Methods, Avalon, California, May 1982.

9. W.E. Howden, "Algebraic Program Testing," ACTA Informatica, 10, 1978.

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

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

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

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

14. S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection," Sixth International Conference on Software Engineering, October 1982.

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

16. Workshop on Effectiveness of Testing and Proving Methods, Avalon, California, May 1982.

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

18. 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.

19. 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.

7. Biographical Sketches

Lori A. Clarke received the B.A. degree in mathematics from the University of Rochester in 1969 and the Ph.D. degree in computer science from the University of Colorado in 1976.

Since 1975 she has been on the faculty in the Department of Computer and Information Science at the University of Massachusetts where she is currently an associate professor. Her interests include software development, programming languages, and compiler design.

Dr. Clarke is an IEEE distinguished visitor and an ACM National Lecturer.

Debra J. Richardson received the B.A. degree in Mathematics from Revelle College of the University of California at San Diego in 1976 and the M.S. and Ph.D. degrees in Computer and Information Science from the University of Massachusetts at Amherst in 1978 and 1981.

Dr. Richardson is currently a Visiting Assistant Professor in the Department of Computer and Information Science at the University of Massachusetts at Amherst. Her interests include program testing and verification, specification languages, and software development environments.

Dr. Richardson is a member of the Association for Computing Machinery, SIGPLAN, SIGSOFT, and the IEEE Computer Society.