Testing Techniques Based on Symbolic Evaluation

Debra J. Richardson
Lori A. Clarke

Technical Report 84-12

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

Testing Techniques Based on Symbolic Evaluation[1]

Debra J. Richardson
Lori A. Clarke

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

Symbolic evaluation is a program analysis method that represents a program's computations and domains by symbolic expressions. This method has been the foundation for much of the current research on software testing. In particular, most test path selection and test data selection techniques, which are two of the primary concerns of program testing, require the information provided by symbolic evaluation. Moreover, symbolic evaluation techniques have recently been extended to be applicable to specifications. This provides the basis for both specification-guided program testing, whereby the specification is used to assist in the program testing process, and specification testing, whereby the specification itself is tested.

In this paper, symbolic evaluation is explained. Several path selection and test data selection techniques that utilize the information provided by symbolic evaluation are then described. In addition, several specification-guided program testing and specification testing methods that use symbolic evaluation are introduced.

1. INTRODUCTION

The ever increasing demand for larger and more complex programs has created a need for automated support environments to assist in the software development process. One of the primary components of such an environment will be validation tools to detect errors, determine consistency, and generally increase confidence in the software under development. Several of the validation tools being developed employ a method, called symbolic evaluation, which creates a symbolic representation of the program. This paper describes symbolic evaluation and surveys some of the testing applications of this method.

Symbolic evaluation monitors the manipulations performed on the input data. Computations and their applicable domain are represented algebraically over the input data, thereby describing the relationship between the input data and the resulting values. Normal execution computes numeric values but loses information about the way in which these numeric values were derived, whereas symbolic evaluation preserves this information. When further analyzed, this

---

[1] This report appears in the Proceedings of the Center for Software Reliability Workshop: Software Requirements, Specification, and Testing.

information provides the basis for several testing techniques.

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 program coverage, data flow testing, and perturbation testing have been proposed. For the test data selection problem, a number of informal guidelines have been put forth. Recently there has been considerable work on developing more systematic test data selection techniques that can either eliminate certain classes of errors or provide a quantifiable error bound. Many of the current path selection and test data selection techniques base their analysis on the information provided by symbolic evaluation.

The above testing techniques are referred to as structural techniques since they base their analysis solely on the information provided by a given implementation. There are two drawbacks to such an approach. First, it ignores the information that may be available from a specification. Second, it delays testing until the implementation is complete, thereby not detecting errors in the most timely and cost-effective manner. Research efforts that use symbolic evaluation to assist in solving both these problems are currently underway. Specification-guided program testing techniques use the information provided by symbolic evaluation of a specification to guide in the testing of its implementation, while specification testing techniques employ symbolic evaluation to actually test a specification.

The next section of this paper provides a brief overview of symbolic evaluation, with an example to demonstrate the method. The third section describes a number of ways in which symbolic evaluation of a program aids the path selection and test data selection aspects of testing. The fourth section discusses the use of symbolic evaluation for specification-guided program testing and specification testing.

2. SYMBOLIC EVALUATION

Symbolic evaluation provides a functional representation of the paths in a module or group of modules. 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. Thus, 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 two parts, path computation and path domain. The path computation is a vector of algebraic expressions for the output values, which include written output values as well as output parameters and exported global values. The path domain 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.

A symbolic representation of all executable paths through a program is usually unreasonable to create due to the presence of loops. If the number of iterations of a loop is dependent on unbounded input values, there is an effectively 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 in which the members differ only by their number of loop iterations.

The loop analysis technique attempts to represent each loop by a loop expression describing the effects of that loop. For each analyzed loop, a conditional expression is created representing the final iteration count for any arbitrary execution of the loop. The final iteration count is expressed in terms of the symbolic values of the variables at entry to the loop. In addition to the final iteration count, the loop expression describes each variable modified within the loop by its symbolic value at exit from the loop. These symbolic values are expressed in terms of the final iteration count and the symbolic values of the variables at entry to the loop. The loop expression is often composed of several subcases -- one for the fall through case and one for each typical case with a unique exit condition.

Once the loop expression is formed, the nodes in the loop can be replaced by a single node, annotated by this loop expression. Later, when such a loop is encountered during symbolic evaluation, each subcase in the loop expression must be considered in the symbolic evaluation process. Thus, the resulting symbolic expression may also be composed of a number of subcases.

The procedure RECTANGLE, shown in Figure 1, is used to illustrate symbolic evaluation. Note that the lefthand side of the listing is annotated with node numbers so that statements or parts of statements can easily be referenced. Paths (or classes of paths) are designated by the ordered list of nodes encountered on the path. Figure 2 provides the loop expression for RECTANGLE. Figure 3 shows the path domains and computations for RECTANGLE, where path P_3 represents the class of paths with one or more iterations of the loop. A detailed description of the method of symbolic evaluation employed here may be obtained from [CLAR83].

```
procedure RECTANGLE (A,B: in real; H: in real range -1.0..1.0;
          F: in array [0..2] of real;
          AREA: out real; 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: real;
s   begin
       — check for valid input
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.

case

—fall through

$(-b + h + X_0 > 0.0)$:

$AREA = AREA_0$

$X = X_0$

$Y = Y_0$

—exit after first or subsequent iteration

$(-b + h + X_0 \leq 0.0)$ and $(k_c = \min< k \mid (k \geq 1)$ and $(-b + h + h*k + X_0 > 0.0) >)$

$= (-b + h + X_0 \leq 0.0)$ and $(k_c = int(b/h - X_0/h))$:

$AREA = AREA_0 + f[0]*k_c - f[1]*h*k_c/2.0 + f[1]*k_c*X_0 + f[1]*h*k_c**2/2.0$
$+ f[2]*h**2*k_c/6.0 - f[2]*h*k_c*X_0 + f[2]*k_c*X_0**2$
$+ f[2]*h**2*k_c**2/2.0 + f[2]*h*k_c**2*X_0 + f[2]*h**2*k_c**3/3.0$

$X = h*k_c + X_0$

$Y = f[0] + f[1]*X_0 + f[1]*h*k_c + f[2]*X_0**2 + 2.0*f[2]*h*k_c*X_0 + f[2]*h**2*k_c**2$

endcase

Figure 2: Loop Expression for RECTANGLE.

P_1 : (s,1,2,f)

$D[P_1]$: $(a - b + h > 0.0)$

$C[P_1]$: $AREA = ?$

$ERROR = true$

P_2 : (s,1,3,4,5,6,10,f)

$D[P_2]$: $(a - b + h \leq 0.0)$ and $(a - b + h > 0.0)$

$=$ false *** infeasible path ***

P_3 : $(s,1,3,4,5,6,(7,8,9,6)^+,10,f)$

$D[P_3]$: $(a - b + h \leq 0.0)$ and $(k_c = int(-a/h + b/h)$

$C[P_3]$: $AREA = (f[0] + a*f[1] + 2.0*a*f[2] + f[0]*k_c - f[1]*h*k_c/2.0 + a*f[1]*k_c$
$+ f[1]*h*k_c**2/2.0 + f[2]*h**2*k_c/6.0 - a*f[2]*h*k_c + a**2*f[2]*k_c$
$+ f[2]*h**2*k_c**2/2.0 + a*f[2]*h*k_c**2 + f[2]*h**2*k_c**3/3.0) * h$
$= f[0]*h + a*f[1]*h + 2.0*a*f[2]*h + f[0]*h*k_c + a*f[1]*h*k_c - f[1]*h**2*k_c/2.0$
$- a*f[2]*h**2*k_c + a**2*f[2]*h*k_c + f[1]*h**2*k_c**2/2.0 + f[2]*h**3*k_c/6.0$
$+ a*f[2]*h**2*k_c**2 + f[2]*h**3*k_c**2/2.0 + f[2]*h**3*k_c**3/3.0$

$ERROR = false$

Figure 3: Path Domains and Computations for RECTANGLE.

---

3. PROGRAM TESTING APPLICATIONS

Testing research has evolved from primarily gathering information about a program to analyzing that information so as to detect errors or provide a guarantee that certain classes of errors cannot occur. The division of the testing process into path selection and test data selection components is based on the recognition that, in general, it is impractical, if not impossible, either to test all paths through the program or to test all inputs to a path. Thus criteria for selecting a subset of paths and criteria for selecting a subset of the input data for those paths are needed. The basic goal is to select paths and test data that will detect errors or guarantee their absence over the whole program. This section describes several path selection and test data selection techniques and emphasizes how these techniques utilize symbolic evaluation.

3.1. Path Selection

Three criteria for selecting paths that have typically been used for program testing are statement, branch, and path coverage. Statement coverage requires that each statement in the program occurs at least once on one of the selected paths. Likewise, branch coverage requires that each branch predicate occurs at least once on one of the selected paths, and path coverage requires that all paths be selected. These three measures provide an ascending scale of confidence in testing; branch coverage implies statement coverage, while path coverage implies branch coverage. Given a reliable method of test data selection, path testing would constitute a proof of correctness. Since path coverage implies the selection of all feasible paths through the routine, attaining path coverage is usually impractical, if not impossible.

It is generally agreed that branch coverage should be a minimum criteria for path selection. Achieving even this level of coverage is not always straightforward. Statically generating a list of paths that satisfy this criterion usually results in a number of infeasible paths being selected. Data flow techniques that attempt to generate only feasible paths by excluding inconsistent pairs of branch predicates have been shown to be NP complete [GABO76]. Thus, symbolic evaluation is a useful technique for aiding in the selection of executable paths. The ATTEST system [CLAR76], for example, uses a dynamic, goal-oriented approach for automated path selection whereby each statement on a path is selected, based on its potential for a selected coverage criterion. When an infeasible path is encountered, ATTEST chooses one of the alternative statements. When there is more than one consistent alternative, the choice is based on the selected coverage criterion [WOOD80].

Unfortunately, branch coverage is easily shown to be inadequate; no matter what test data is selected for these paths, many simple, common errors will go undetected. Several stronger criteria have been proposed for selecting paths that fall between the two levels of reliability and expense associated with branch testing and path testing. Some alternative criteria use a modified path coverage criterion that simply limits loop iterations. For example, the EFFIGY system [KING76] generates all paths with a bound specified on the number of loop iterations and the ATTEST system, in addition to statement or branch coverage, attempts to select paths that traverse each loop a minimum and maximum number of times.

Howden has proposed the boundary-interior method for classifying paths [HOWD75]. With this method, two paths that differ in ways other than in loop traversals are in different classes. In addition, two paths that differ only in the way they traverse loops are in different classes if

1. one is a boundary and the other an interior test of a loop;

2. they enter or leave a loop along different loop entrance or loop exit branches;
3. they are boundary tests of a loop and follow different paths through the loop;
4. they are interior tests of a loop and follow different paths through the loop on their first iteration of the loop.

A boundary test is one which enters the loop but leaves it before carrying out a complete traversal and an interior test carries out at least one complete traversal of the loop. A set of test data is considered to cover all classes if at least one path from each class is exercised by the test data. Again, symbolic evaluation is useful for determining a set of feasible paths that satisfy the loop criterion. Moreover, when loop analysis is successful in creating a closed form representation of the loop, then this representation is useful in determining the paths that satisfy the selected loop criterion.

An alternative to the use of control flow as the determining factor in path selection is the use of data flow. Data flow techniques [LASK79, NTAF81, RAPP82] require the selection of subpath(s) based on particular sequences of definitions and references to the variables in the program. Rapps and Weyuker [RAPP82] have described a partial ordering on a family of data flow techniques for path selection. Figure 4 shows this partial ordering as well as its relation to statement, branch, and path coverage. As an example of the application of these techniques, consider the flow chart in Figure 5. Def coverage requires the selection of a subpath containing each definition of a variable; the following paths satisfy def coverage: (1,2,3,5,6,8) and (1,2,3,5,7,8). Note that this set of paths does not satisfy either statement or branch coverage since statement 4 is not executed. Use coverage requires the selection of some subpath from each definition of a variable to each use of that variable; the following paths satisfy use coverage: (1,2,3,5,6,8) and (1,2,4,5,7,8). Du-path coverage, on the other hand, requires the selection of all minimum loop subpaths from each definition of a variable to any use of that variable. In addition to the two paths for use coverage, the path (1,2,3,5,7,8) must be selected because it includes a subpath from the definition of Y at node 3 to its use at node 8. Note that there is one more path, (1,2,4,5,6,7), that would need to be selected to satisfy path coverage but no additional flows of data are to be gained by testing that path. Although the data flow path selection techniques can be applied independently of symbolic evaluation, a number of infeasible paths will be generated unless data flow analysis and symbolic evaluation techniques are paired together.

In addition to using control and data flow information, path selection techniques have been developed that relate directly to the elimination of potential errors in program statements. Perturbation testing [HALE82, ZEIL83] attempts to compute the set of potential errors in arithmetic expressions that cannot possibly be detected by testing only the current set of selected test paths, regardless of the test data selection techniques employed for those paths. Perturbation testing derives a set of characteristic expressions that describe the undetectable perturbations (errors). This information can be used to select additional paths that must be

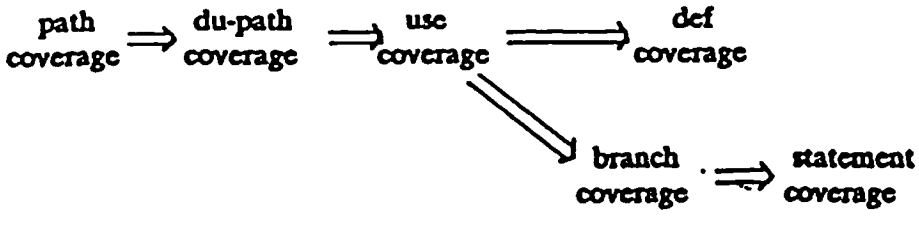

Figure 4: Data Flow Testing Criteria.

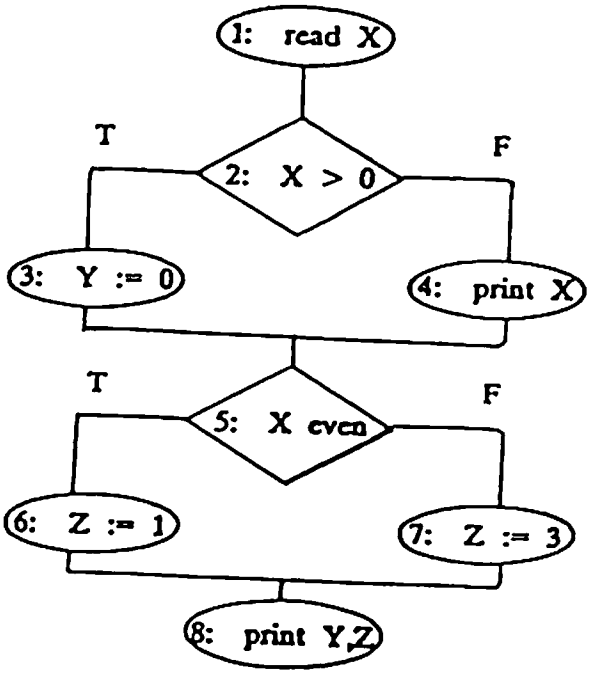

Figure 5: Data Flow Testing Example.

tested in order to detect these possible perturbations. As an example, consider the flow chart in Figure 6. Along path (...,1,3,...) the value of Z is the same as the value of 2•X at node 3. Any error in the predicate at node 3 that can be represented by k • (Z – 2•X), where k is a constant, could not be detected along path (...,1,3,...). For instance, if the branch predicate at node 3 should have been Z – X > Y, the error would not be detected. Along path (...,2,3,...), however, this equality does not hold and thus this error could be detected. In general, another proposed path will be a useful test if, and only if, it eliminates one or more expressions describing undetectable perturbations. In effect, perturbation testing systematically captures the interesting error detection capabilities of mutation testing [BUDD81], a method that sequentially introduces a large number of small errors (mutants) into a program and then determines which of these errors were not detected by the selected test data. The perturbations of a statement can be represented by using symbolic evaluation techniques. Perturbation testing is currently being implemented as an extension to the ATTEST symbolic evaluation system.

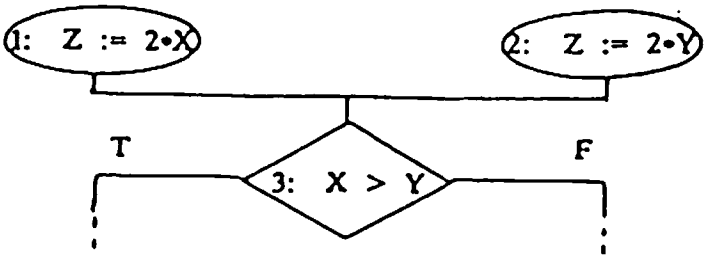

Figure 6: Perturbation Testing Example.

3.2. Test Data Selection

Symbolic evaluation, like most other methods of program analysis, does not actually execute a routine in its natural environment. Evaluation of the path computation for particular input values returns numeric results, but because the environment has been changed, these results may not always agree with those from normal execution. Errors in the hardware, operating system, compiler, or symbolic evaluation system itself may cause an erroneous result. It is thus important to test the routine on actual data. In addition, testing a routine demonstrates its run-time performance characteristics.

To assist with test data selection, several error-sensitive heuristics have been proposed. Myer's error guessing [MYER79], Foster's error-sensitive test case analysis [FOST80], Weyuker's error-based testing [WEYU81], and Redwine's engineering approach [REDW83] provide guidelines for selecting test data to detect likely errors. Each approach is based on examining the statements in a program or an informal description of the intent of the program.

The symbolic representation of a path can be used as the basis on which to select test data for a path. The most straightforward technique simply examines the PC to determine a solution – that is, one arbitrary test datum to execute the path. SELECT [BOYE75] and ATTEST are two symbolic execution systems that generate such test data by using an algebraic technique for evaluating the constraints comprising the path domain.

More rigorous techniques have been proposed that appear to capture the ideas underlying the error-sensitive heuristics by characterizing potential errors in terms of their effects on a path. For these techniques, errors are classified into two types, computation errors and domain errors, according to whether the effect is an incorrect path computation or an incorrect path domain. A domain error may be either a missing path error, which occurs when a special case requires a unique sequence of actions but the program does not contain a corresponding path, or a path selection error, which occurs when a program recognizes the need for a path but incorrectly determines the conditions under which that path is executed. A number of test data selection techniques focus on the detection of either domain or computation errors. These techniques analyze the symbolic representations created by symbolic execution and select data for which the path computation and path domain appear sensitive to errors. A difficult problem, which must be addressed by these techniques, is the possibility that an error on an executed path may not produce erroneous results for particular test data; this is referred to as coincidental correctness. For an example, note that the second multiplication operator in statement 5 of RECTANGLE should be an exponentiation operator. If this statement is only executed when A=0.0 or 1.0, then the actual resulting value and the intended value agree. Although this is a contrived example, coincidental correctness is a common phenomenon of testing. A goal, therefore, is to minimize the occurrence of coincidentally correct results by astutely selecting test data aimed at exposing, not masking, errors.

In RECTANGLE there are five errors: one computation error, three missing path errors, and a path selection error. As noted above, 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 and third errors are caused by an erroneous check for a valid input value for h when a > b (the input check is only correct if a < b). If a > b, then h must be negative (error two) and its absolute value must be less than a – b (error three). Both errors two and three are missing path errors. Moreover, h cannot be zero, regardless of the relationship between a and b or an infinite loop results; this fourth error is also a missing path error. A correct check for invalid

input follows:

```
if (A > B and H ≥ 0.0) or (A < B and H ≤ 0.0) then
ERROR := true;
else if (abs (H) > abs (B – A)) then
ERROR := true;
```

Another situation, which might be considered a fifth 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. A more accurate algorithm would add in the area of a smaller rectangle on the last iteration of the loop (or subtract the excess upon exit). In the ensuing discussion it is shown how four of these five errors are detected by test data selection techniques.

Computation testing techniques select test data aimed at revealing computation errors. One such approach analyzes the symbolic representations of the path computation. This approach 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. Guidelines have been proposed for selecting test data aimed at revealing computation errors that are considered likely to occur for both types of path computations [CLAR83]. One of these guidelines states that each symbolic name corresponding to a multiplier in the path computation should take on the special values zero, one, and negative one, as well as nonextremal and extremal values. Note that such a selection of values for A in RECTANGLE would reveal the first error.

There have been some theoretical results showing that more rigorous computation testing techniques can guarantee the absence of certain types of computation errors when the path computations fall into well-behaved functional classes. For example, there are a few techniques that can be applied if the symbolic value for an output parameter should be a polynomial. For a univariate polynomial with integer coefficients whose magnitudes do not exceed a known bound, a single test point can be found to demonstrate the correctness of that polynomial [ROWL81]. Alternately, for a univariate polynomial of degree N, N+1 test points are sufficient [HOWD78]. Probabilistic arguments have been made for reducing this number without sacrificing much confidence [DEMI78], and similar results have been provided for multivariate polynomials.

When the path computations fall into specialized categories, the 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 takes on the value zero and both positive and negative values. Data satisfying this extension would alert the tester to the poor termination condition (the fifth error).

Domain testing techniques [CLAR82, WHIT80] concentrate on the detection of domain errors by analyzing the path domains and selecting test data "on" and slightly "off" the closed borders of each path domain. 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 7 illustrates a border shift, where G is the given border, C is the correct border, and the set of elements in the wrong domain is shaded. The border shift is revealed by testing the on points P and Q and the off points U and V, since V is in the wrong domain. For a border in higher dimensions, 2*v (where v is the number of vertices of the border) test data points must be selected for best results. A thorough description of the domain testing technique and its effectiveness is provided in [CLAR82]. Figure 9 shows the test data selected for the paths in RECTANGLE to satisfy the domain testing technique. The only closed border is ($a - b + h \leq 0.0$). 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). Four of the five 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 either of the two off points (a = 100.0 and b = 99.99 and h = 0.01) or (a = -99.99 and b = -100.0 and h = 0.01). Error four is detected by either of the two on points (a = 100.0 and b = 100.0 and h = 0.0) or (a = -100.0 and b = -100.0 and h = 0.0). The inaccurate termination condition (error five) is revealed by testing either of the off points (a = 100.0 and b = 98.99 and h = -1.0) or (a = -98.99 and b = -100.0 and h = -1.0). The third error is a missing path error that will not be detected by domain testing. This error occurs when ($a > b$) and ($h < 0.0$) and ($abs(h) > a - b$), which implies that $a - b + h < 0.0$; this describes points in the domain of P_3 but not on the closed border and thus will not be selected by domain testing.

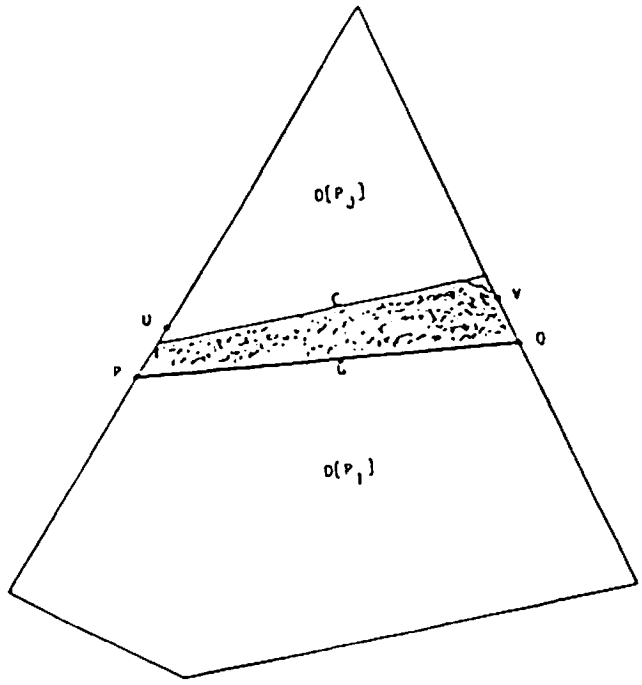

Figure 7: Domain Testing Strategy.

Conditions for on points for $(a - b + h \leq 0.0)$

$a = 100.0$ 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 $h = 0.0$
$a = -100.0$ 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 $(a - b + h \leq 0.0)$

$a = 100.0$ and $b = 98.99$ and $h = -1.0$
$a = 99.01$ and $b = 100.0$ and $h = 1.0$
$a = 100.0$ and $b = 99.99$ and $h = 0.01$
$a = -100.0$ and $b = -99.01$ and $h = 1.0$
$a = -99.99$ and $b = -100.0$ and $h = 0.01$
$a = -98.99$ and $b = -100.0$ and $h = -1.0$

Figure 8: Conditions for Satisfying Domain Testing Strategy for RECTANGLE.

Existing domain testing techniques are aimed at the detection of path selection errors. As illustrated in the example, missing path errors may not be detected by such techniques. A missing path error is particularly difficult to detect since it is possible that only one point in a path domain should be in the missing path domain; the error will not be detected unless that point happens to be selected for testing. When a missing path error corresponds to a missing path domain that is near a boundary of an existing path domain, then the error may be caught by domain testing techniques, as occurred in RECTANGLE for errors two and four. Missing path errors cannot be found systematically, however, unless a specification is employed by the test data selection method, as is done by specification-guided program testing.

4. SPECIFICATION TESTING APPLICATIONS

The major drawbacks to the program testing techniques described above are their failure to consider specifications or to be applicable earlier in the software lifecycle. Specification-guided program testing techniques attempt to incorporate information from the specification into the analysis process. It is only through such techniques that missing path errors can be detected in a systematic way. Of course there is no guarantee that a specification will adequately capture the functionality of a problem. Moreover, many of the proposed specification languages are difficult to comprehend. Thus, it is imperative that specifications themselves be tested in order to provide some assurance about their validity. In this section, techniques based on symbolic evaluation for specification-guided program testing and specification testing techniques are described.

4.1. Specification-Guided Program Testing

A specification provides an independent description of the external behavior of a procedure and thus provides an alternative, and usually more abstract, representation to which an implementation of the procedure can be compared. Often the specification provides valuable functional information that should be utilized for testing.

The partition analysis method [RICH78, RICH81a] incorporates information derived from such a specification with information derived from the corresponding implementation to assist in determining program reliability. This is done by symbolically evaluating a specification and a corresponding implementation, and then evaluating the differences and similiarities between their symbolic representations. With appropriate modifications, symbolic evaluation techniques can be applied to specifications written in a number of different kinds of specification languages. The resulting symbolic representation provides a specification-based partition, which consists of descriptions of specification subdomains and their corresponding computations.

To compare the specification and implementation, the partition analysis method forms a procedure partition. This partition is constructed by finding the non-empty intersections between the implementation-based partition, which is defined by the program path domains, and the specification-based partition, which is defined by the specification subdomains. The procedure partition thus divides the set of input data for the procedure into procedure subdomains so that the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. The procedure partition also embodies representations of the specification subdomain computations and path computations associated with each procedure subdomain. By forming the procedure partition, the procedure's domain is decomposed into more manageable units, as is the task of evaluating program reliability. The partition analysis method then applies both verification and testing techniques in the context of each procedure subdomain in the procedure partition.

Partition analysis verification uses information related to each procedure subdomain to verify consistency between the specification and the implementation for the subdomain. Partition analysis verification is a variation on symbolic testing [HOWD77]. Symbolic testing involves examining the symbolic representations of the path domains and computations. Partition analysis verification, however, compares these representations with those derived from the specification. Partition analysis verification uses standard proof techniques to determine the equality of the specification subdomain's computations to their corresponding path computations, where both are restricted over the procedure subdomain. Most verification methods [DEUT73, FLOY67, KING69, LOND75] prove that the implementation is consistent to assertions, which serve as the specification of the procedure's intended behavior. These assertions, however, are seldom developed independently of the implementation; rather they are associated with the structure of the implementation (as in loop invariant assertions). Partition analysis verification, on the other hand, is designed to use an independent specification that most likely would have been written in one of the pre-implementation phases of the software development process [BAUE79, CAIN75, SILV79, WARN74, WIRT73, YOUR75]. Recent experimental results [RICH82] show that partition analysis verification is capable of detecting fairly subtle inconsistencies between two descriptions of a procedure. Of course, partition analysis verification suffers some of the same drawbacks as other verification approaches since, in general, the proof of computation equality is undecidable. When proof techniques do fail, testing can provide some assurance of the equality of the computations or find examples of their inequality.

Thus, partition analysis complements the verification process by the astute selection of test data on which the implementation should be executed. Partition analysis testing constructs a test data set by selecting data from each subdomain of the procedure partition. The symbolic representations of a procedure subdomain and the associated computations are employed to direct the selection of this test data. Partition analysis testing thereby draws on information describing both the intended and actual function of the procedure. To increase the likelihood of detecting errors, partition analysis testing employs some of the computation and domain

testing techniques previously described. Partition analysis testing has been shown to be a powerful testing method [RICH82]. The reasons for this are three-fold. First, it integrates several complementary testing techniques. Second, the selected test data appropriately characterize the procedure based on both the implementation and the specification. As such, it is one of the few testing methods to address missing path errors. Third, the testing and verification processes are integrated within partition analysis so that they might complement and enhance one another; the testing of some elements in the procedure subdomain may assist in verification, while the verification process may direct the selection of test data.

Partition analysis has been applied to several different kinds of specification languages, including state transitions and both high- and low-level procedural languages [RICH81b]. Thus, the basic ideas of applying symbolic evaluation to pre-implementation descriptions and comparing two representations at different levels of detail seems to be generally applicable to a wide range of languages; it can be applied to compare software specifications to designs, high-level to low-level designs, VLSI specs to VLSI designs, and so on.

Weyuker and Ostrand [WEYU80] have proposed a specification-guided program testing method based on a partition of a problem into revealing subdomains, which are developed using specification related information as well as the implementation. A subdomain is revealing if the existence of one element of the subdomain that is processed incorrectly by the implementation implies that all elements of the subdomain are processed incorrectly. Revealing subdomains are constructed by overlaying a specification-based partition and an implementation-based partition. They are, therefore, similar to procedure subdomains. For this technique the implementation-based partition can be derived through symbolic evaluation, but it is not clear how to derive a specification-based partition with the appropriate error revealing characteristics.

Gourlay has developed a specification-guided program testing approach that is also similar to partition analysis testing [GOUR81]. This approach uses a predicate calculus description of the problem to form a specification-based partition. Using Gourlay's technique, a predicate calculus formula is interpreted as specifying a number of distinct computations, each of which is applicable over some subdomain. Test data is then selected from each subdomain. Gourlay proposes that this test data set be augmented by path testing; this provides the implementation-based partition and makes the final decomposition quite similar to the procedure subdomains used with partition analysis.

4.2. Specification Testing

The use of the specification to assist in the testing of an implementation solves many of the problems inherent in program testing. It is desirable, however, to initiate the testing process before the implementation is complete; if an error introduced in a pre-implementation phase is revealed soon after its inception, the cost of its detection and correction is greatly reduced. It is, therefore, imperative that testing be performed throughout the software development process.

As noted above, the partition analysis method can be employed as an approach to solving this problem. Extensions of symbolic evaluation to specification languages have been designed so as to be applicable to a wide range of notations and levels of abstraction. The partition analysis method can thereby be applied to compare a specification at one level of abstraction to one at a higher level of abstraction. Kemmerer [KEMM84] suggests a similar approach for INAJO, a language for the state machine approach to specifying the functionality of a program. Each

level of specification is proven to be consistent with the level above, and the implementation is proven to be consistent with the lowest level specification. By induction, the implementation has been shown to be consistent with the highest level specification.

The problem with these approaches is that the actual functionality of the specifications has not been tested; instead consistency between two levels of specification has been shown in a postulated environment. Incorrect functionality may not be discovered until several levels of refinement have been completed. It may be costly to backup the lifecycle and rewrite specifications. Thus, specifications should be tested to determine if they achieve the desired functionality. Kemmerer proposes two techniques for testing specifications. The first is to convert a non-procedural specification into a procedural form that serves as a rapid prototype to use for testing. The second approach is to perform a symbolic evaluation of the sequence of operations and check the resultant symbolic values to see if they define the desired resultant states.

This latter approach to pre-implementation testing has been experimented with using a symbolic evaluation system for the GIST specification language [COHE82]. In this project, symbolic evaluation of a formal specification is envisioned as an alternative to early prototype development. Symbolic evaluation is used in an attempt to characterize the behaviors that satisfy a given specification. Logic errors in the specification that are uncovered are pointed out as unintended or missing behaviors. As was similarly noted for programs, symbolic evaluation of a specification tests a range of possible inputs as opposed to concrete execution of a prototype, which for each test case only tests a single path for a unique set of inputs.

5. SUMMARY

In this paper several testing techniques that use symbolic evaluation are described.

For the path selection aspects of testing, symbolic evaluation is useful in determining path feasibility for the control flow and data flow criteria. It is also being used in the analysis employed by perturbation testing. It is interesting to note that path selection and symbolic evaluation have a symbiotic relationship. Symbolic evaluation is used to guide the selection of paths, which are then symbolically evaluated. Thus, adaptive systems, where path selection and symbolic evaluation dynamically interact, must be considered.

Several test data selection techniques are being developed that select data based on an examination of the symbolic representations created by symbolic evaluation. Both computation and domain testing techniques have been developed using this approach. While the initial work in this area is quite promising, it is clear that a complementary set of techniques must be developed.

Several testing approaches that utilize a specification have also been developed using symbolic evaluation techniques. Specification-guided program testing assists in the selection of test data for the implementation by using information derived from the specification. Specification testing provides the capability to test the system under development before implementation is underway.

For the most part, current research is addressing the issues of path selection, test data selection, and specification testing as independent topics. It is clear, however, that these topics are closely related and eventually should be integrated into a software development

environment.

REFERENCES

BAUE79 F.L. Bauer, M. Broy, R. Gratz, W. Hesse, B. Krieg-Bruckner, H. Partsch, P. Pepper, and H. Wossner, "Towards a Wide-Spectrum Language to Support Program Specification and Program Development," Program Construction, Lecture Notes in Computer Science, Springer-Verlag, 1979.

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.

BUDD81 T.A. Budd, "Mutation Analysis: Ideas, Examples, Problems and Prospects," Computer Program Testing, North-Holland Publishing Company, B. Chandrasekaran and S. Radicchi (eds.), 1981, 129-148.

CAIN75 S.H. Cain and E.K. Gorden, "PDL -- A Tool for Software Design," Proceedings of the National Conference on Computers 75, 1975, 271-276.

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.

CLAR76 L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, SE-2, 3, September 1976, 215-222.

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.

CLAR83 L.A. Clarke and D.J. Richardson, "A Rigorous Approach to Error-Sensitive Testing," Sixteenth Hawaii International Conference on System Sciences, January 1983.

COHE82 D. Cohen, W. Swartout, and R. Balzer, "Using Symbolic Execution to Characterize Behavior," ACM SIGSOFT Rapid Prototyping Workshop, Software Engineering Notes, 7,5, December 1982, 25-32.

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

DEUT73 L.P. Deutsch, "An Interactive Program Verifier," Ph.D. Dissertation, University of California, Berkeley, May 1973.

FLOY67 R.W. Floyd, "Assigning Meaning to Programs," Proceedings of a Symposium in Applied Mathematics, 19, American Mathematical Society, 1967, 19-32. Communications of the ACM, 14, 1, January 1971, 39-45.

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

GABO76 H.N. Gabow, S.N. Maheshwari, and L.J. Osterweil, "On Two Problems in the Generation of Program Test Paths," IEEE Transactions on Software Engineering, SE-2, 3, September 1976, 227-231.

GOUR81 J.S. Gourlay, "Theory of Testing Computer Programs," Ph.D. Thesis, University of Michigan, 1981.

HALE82 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.

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

HOWD77 W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, SE-3, 4, July 1977, 266-278.

HOWD78 W.E. Howden, "Algebraic Program Testing," ACTA Informatica, 10, 1978.

KEMM84 R.A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," University of California at Santa Barbara, Department of Computer Science, Technical Report 84-06, March 1984, to appear IEEE Transactions on Software Engineering.

KING69 J.C. King, "A Program Verifier," Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, September, 1969.

KING76 J.C. King, "Symbolic Execution and Program Testing," CACM, 19, 7, July 1976, 385-394.

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

LOND75 R.L. London, "A View of Program Verification," Proceedings International Conference on Reliable Software, April 1975, 534-545.

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.

RAPP82 S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection," Sixth International Conference on Software Engineering, October 1982.

REDW83 S.T. Redwine, "An Engineering Approach to Test Data Design," IEEE Transactions on Software Engineering, SE-9, 2, March 1983, 191-200.

RICH81a D.J. Richardson, L.A. Clarke, "A Partition Analysis Method to Increase Program Reliability," Fifth International Conference on Software Engineering, March 1981, 244-253.

RICH81b D.J. Richardson, "Specifications for Partition Analysis," Department of Computer and Information Science, University of Massachusetts, Technical Report 81-34, August 1981.

RICH82 D.J. Richardson and L.A. Clarke, "On the Effectiveness of the Partition Analysis Method," Proceedings of the IEEE Sixth International Computer Software and Applications Conference, November 1982, 529-538.

ROWL81 J.H. Rowland and P.J. Davis, "On the Use of Transcendentals for Program Testing," Journal of the Association for Computing Machinery, 28,1, January 1981, 181-190.

SILV79 B.A. Silverburg, L. Robinson, and K.N. Levitt, "The Languages and Tools of HDM," Stanford Research Institute Project 4828, June 1979.

WARN74 J.D. Warnier, "Logical Construction of Programs," Van Nostrand Reinhold Co., New York, 1974.

WEYU80 E.J. Weyuker and T.J Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Transactions on Software Engineering, SE-6, 3, May 1980, 236-246.

WEYU81 E.J. Weyuker, "An Error-Based Testing Strategy," Computer Science Department, New York University, New York, New York, Technical Report 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, 3, May 1980, 247-257.

WIRT73 N. Wirth, "Systematic Programming," Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

WOOD80 J.L. Woods, "Path Selection for Symbolic Execution Systems," Ph.D. Dissertation, University of Massachusetts, May 1980.

YOUR75 E. Yourdon and L.L. Constantine, "Structured Design," Yourdon Press, New York, 1975.

ZEIL83 S.J. Zeil, "Testing for Perturbations of Program Statements," IEEE Transactions on Software Engineering, SE-9, 3, May 1983, 335-346.