Symbolic Evaluation -- An Aid to Testing and Verification[1]

Lori A. Clarke
Debra J. Richardson

COINS Technical Report 83-41

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

---

[1] This paper appears in Software Validation, editor Han-Ludwig Hausen, North Holland Publishing Company, 1983.

Symbolic Evaluation — An Aid to Testing and Verification[1]

Lori A. Clarke
Debra J. Richardson

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 domain by symbolic expressions. This method has been the foundation for much of the current research on software testing. Most path selection and test data selection techniques, which are two of the primary concerns of testing research, require the information provided by symbolic evaluation. Symbolic evaluation is also employed by verification techniques. In addition to formal verification, several less rigorous verification techniques utilize the symbolic expressions created by symbolic evaluation to certify program properties.

In this paper, the general symbolic evaluation method is explained. Several path selection and test data selection techniques that utilize the information provided by symbolic evaluation are then described. Some informal verification techniques, which also employ this information, are discussed. Finally, the partition analysis method, which uses symbolic evaluation to combine both testing and verification is described.

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, that creates a symbolic representation of the program. This paper describes symbolic evaluation and surveys some of the testing and verification 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, thus 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 information provides the basis for several testing and verification methods.

[1] This work was supported by the National Science Foundation under grants NSFMCS 81-04202 and 83-03320.

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.

Formal verification techniques have usually applied symbolic evaluation techniques to develop verification conditions. There are a number of less comprehensive verification techniques that have used symbolic evaluation to certify the correctness of selected program properties. Some current work is being directed at developing methods that integrate testing and formal verification, based upon symbolic evaluation.

The next section of this paper provides a brief overview of symbolic evaluation and an example is presented to demonstrate the method. The third section describes a number of ways in which symbolic evaluation aids the testing process. Both the path selection and test data selection aspects of testing are discussed. The fourth section discusses how the verification process can utilize symbolic evaluation. Some informal verification approaches are described. Finally, the partition analysis method, which uses symbolic evaluation techniques to combine testing and verification, is presented in some detail.

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 <u>path computation</u>, which is a vector of algebraic expressions for the output values (including the values returned by parameters) and the <u>path domain</u>, which is defined by the conjunction of the path's branch predicate constraints. For path P_J the path computation and path domain are denoted by $C[P_J]$ and $D[P_J]$, respectively.

The forward expansion method is the most straightforward way to do symbolic evaluation [CLAR81] 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 up 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 <u>path condition</u>, denoted <u>PC</u>, 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.

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. Path (s,1,3,4,5,6,10,f) is an example of an infeasible path.

The symbolic evaluation of the feasible path (s,1,3,4,5,6,7,8,9,6,10,f) is described below and Figure 2 shows the expressions that are generated.

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 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 current 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

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

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

$$\text{true and not } (h > b-a) = (a-b + h \leq 0.0).$$

Symbolic interpretation of the statements on a path P_J provides a symbolic representation of the path computation and path domain. The path computation $C[P_J]$ consists of the symbolic representation of the output values. The symbolic representation of the path domain $D[P_J]$ is provided by the path condition. Note that only the input values that satisfy the path condition could cause execution of the path.

```
     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.

```
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
        = f[0] + a*f[1] + 2.0*a*f[2]

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

9  AREA = f[0] + a*f[1] + 2.0*a*f[2] + f[0] + a*f[1]
            + f[1]*h + a**2*f[2] + 2.0*a*f[2]*h   f[2]*h**2
          = 2.0*f[0] + 2.0*a*f[1] + 2.0*a*f[2] + f[1]*h
            + a**2*f[2] + 2.0*a*f[2]*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*f[0] + 2.0*a*f[1] + 2.0*a*f[2] + f[1]*h
           + a**2*f[2] + 2.0*a*f[2]*h + f[2]*h**2) * h
         = 2.0*f[0]*h + 2.0*a*f[1]*h + 2.0*a*f[2]*h + f[1]*h**2
           + a**2*f[2]*h + 2.0*a*f[2]*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*f[0]*h + 2.0*a*f[1]*h + 2.0*a*f[2]*h + f[1]*h**2
          + a**2*f[2]*h + 2.0*a*f[2]*h**2 + f[2]*h**3
```

Figure 2: Symbolic Evaluation of Path 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 [CHEA79, CLAR81]. Using this technique, a path may then represent a class of paths that differ only by their number of loop iterations.

The loop analysis technique attempts to represent each loop by a loop expression, which describes 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, for each variable modified within the loop its symbolic value at exit from the loop is created in terms of the final iteration count and the symbolic values of the variables at entry to the loop.

A loop is not analyzed until all its nested loops have been replaced by their associated loop expression. At the time of analysis, therefore, each loop[1] contains only one backward branch, so each path in the loop can be symbolically evaluated. To initiate the evaluation, an iteration counter, say k, is associated with the loop. For each variable y, y_0 represents the value of the variable y on entry to the first iteration of the loop and y_k, $k \geq 1$, represents the value of the variable y after execution of the kth iteration of the loop. The body of the loop is then symbolically evaluated to get a representation of a typical iteration. This evaluation, suppose it is for the kth iteration, is identical to the normal evaluation process, except that the symbolic name initially assigned to each variable is its value after execution of iteration k–1 – that is, the initial value for y is y_{k-1}. This provides a recurrence relation for each y_k, $k \geq 1$, which is in terms of the values of the variables after iteration k–1. Next, the branch predicate controlling exit from the loop is interpreted in terms of the values of the variables after execution of the kth iteration. This provides the loop exit condition, denoted lec_k, which represents the condition under which the loop will be exited after the kth iteration. The first part of Figure 3 shows the results of this evaluation for the WHILE loop in RECTANGLE.

Next, loop analysis attempts to find solutions to the recurrence relations for each variable in terms of the values of the variables on entry to the loop. The solution to the recurrence relation for y_k is denoted by y(k) and represents the value of the variable y on exit from the kth iteration of the loop. Solutions are found first for those variables that do not reference other variables whose recurrence relations are as yet unsolved. Once a solution is found for a variable, it is substituted for all references to it in the remaining recurrence relations. This process is repeated, if possible, until all recurrence relations are solved. The loop exit condition lec_k is then solved by replacing each y_k referenced in the condition by its solution y(k) and simplifying. This provides lec(k), the condition under which the loop will be exited after execution of the kth iteration. The second part of Figure 3 provides the solutions to the recurrence relations for the loop in RECTANGLE. Although not illustrated in this example, sometimes two subcases must be considered independently: 1) the first iteration of the loop (k=1), where the recurrence relations and loop exit condition depend on the values of the variables at entry to the loop; and 2) all subsequent iterations (k>1), where the recurrence relations and loop exit condition depend on the values computed by the previous iteration.

After solutions to the recurrence relations have been determined, the loop expression can be created. The loop expression for the loop in RECTANGLE appears in the last part of Figure 3. Each subcase consists of the loop exit condition and the values of the variables at exit from the loop. The first subcase in this figure represents the fall-through condition, which must be included for any while loop or similar loop construct. For this subcase, the values at entry to the first iteration of the loop satisfy the loop exit condition and provide the values on exit from the loop. The second subcase represents one or more iterations of the loop and is derived from the solved recurrence relations and loop exit condition. Usually, for this subcase,

[1] Only single-entry, single-exit loops are considered here.

Recurrence Relations and Loop Exit Condition for RECTANGLE
Created by Symbolic Evaluation of kth Iteration of Loop

$AREA_k = AREA_{k-1} + Y_k$
$\quad = AREA_{k-1} + f[0] + f[1]*X_k + f[2]*X_k**2$

$X_k = X_{k-1} + h$
$\quad = h + X_{k-1}$

$Y_k = f[0] + f[1]*X_k + f[2]*X_k**2$

$lec_k = \text{not } (X_k + h \le b)$
$\quad = (-b + h + X_k > 0.0)$

Solved Recurrence Relations and Loop Exit Condition for RECTANGLE

$AREA(k) = AREA_0 + \text{sum} < i:=1..k \mid f[0] + f[1]*(h*i+X_0) + f[2]*(h*i+X_0)**2 >$
$\quad = AREA_0 + f[0]*k + f[1]*k*X_0 + f[2]*k*X_0**2$
$\qquad + \text{sum} < i:=1..k \mid f[1]*h*i + f[2]*h**2*i**2 + 2.0*f[2]*h*i*X_0 >$
$\quad = AREA_0 + f[0]*k + f[1]*k*X_0 + f[2]*k*X_0**2 + f[1]*h*k*(k-1)/2.0$
$\qquad + f[2]*h**2*k*(k+1)*(2*k+1)/6.0 + 2.0*f[2]*h*k*(k-1)*X_0/2.0$
$\quad = AREA_0 + f[0]*k + f[1]*k*X_0 + f[2]*k*X_0**2 - f[1]*h*k/2.0$
$\qquad + f[1]*h*k**2/2.0 + f[2]*h**2*k/6.0 + f[2]*h**2*k**2/2.0$
$\qquad + f[2]*h**2*k**3/3.0 - f[2]*h*k*X_0 + f[2]*h*k**2*X_0$
$\quad = AREA_0 + f[0]*k - f[1]*h*k/2.0 + f[1]*k*X_0 + f[1]*h*k**2/2.0$
$\qquad + f[2]*h**2*k/6.0 - f[2]*h*k*X_0 + f[2]*k*X_0**2$
$\qquad + f[2]*h**2*k**2/2.0 + f[2]*h*k**2*X_0 + f[2]*h**2*k**3/3.0$

$X(k) = h*k + X_0$

$Y(k) = f[0] + f[1]*(h*k+X_0) + f[2]*(h*k+X_0)**2$
$\quad = f[0] + f[1]*h*k + f[1]*X_0 + f[2]*h**2*k**2 + 2.0*f[2]*h*k*X_0 + f[2]*X_0**2$

$lec(k) = (-b + h + h*k + X_0 > 0.0)$

Loop Expression for RECTANGLE

case
—fall through
$(-b + h + X_0 > 0.0)$:
$\quad AREA = AREA_0$
$\quad X = X_0$
$\quad Y = Y_0$
—exit after first or subsequent iteration
$(-b + h + X_0 \le 0.0) \text{ and } (k_e = \min< k \mid (k \ge 1) \text{ and } (-b + h + h*k + X_0 > 0.0) >)$
$= (-b + h + X_0 \le 0.0) \text{ and } (k_e = \text{int}(b/h - X_0/h))$:
$\quad AREA = AREA_0 + f[0]*k_e - f[1]*h*k_e/2.0 + f[1]*k_e*X_0 + f[1]*h*k_e**2/2.0$
$\qquad + f[2]*h**2*k_e/6.0 - f[2]*h*k_e*X_0 + f[2]*k_e*X_0**2$
$\qquad + f[2]*h**2*k_e**2/2.0 + f[2]*h*k_e**2*X_0 + f[2]*h**2*k_e**3/3.0$
$\quad X = h*k_e + X_0$
$\quad Y = f[0] + f[1]*X_0 + f[1]*h*k_e + f[2]*X_0**2 + 2.0*f[2]*h*k_e*X_0 + f[2]*h**2*k_e**2$
endcase

Figure 3: Loop Analysis of RECTANGLE.

the final iteration count, call it k_e, is represented in terms of the minimum k, $k \geq 1$, such that the loop exit condition is true, and the value for each variable y at exit from the loop is represented by $y(k_e)$. In this example it is possible to precisely represent k_e by $\mathrm{int}(b/h - X_0/h)$.

The loop expression is a closed form representation capturing the effects of the loop. Thus, 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. The resulting symbolic expression may be composed of a number of subcases. Figure 4 provides the path domains and computations for RECTANGLE, where path P_3 represents the class of paths with one or more iterations of the loop.

As one might expect, there are several problems associated with loop analysis. Obtaining the solutions to the recurrence relations is not always straightforward and sometimes may not be possible. Complications arise in several situations. In particular, the interdependence between two recurrence relations may be cyclic – y may depend on x, which depends on y – in which case the recurrence relations cannot be solved. Problems also arise when conditional execution occurs within the loop body, causing conditional recurrence relations. This results in a more complicated loop expression, provided these recurrence relations can even be solved. Thus, loops often cause an explosion in the size and complexity of the global representation of a routine. Nested loops exacerbate this problem. In addition, determining consistency of a PC incorporating a loop exit condition may also be problematic. This is due to the possible representation of a final loop iteration count in terms of conditional expressions or a minimum value expression, or both. Deciding the existence of these minimum values is essentially proving routine termination. When none of these problems arise, however, the loop analysis technique provides a general evaluation of a loop that is very useful. In practice, not only can loops often be represented in a closed-form, but many loops are variants of common patterns. Recognizing these patterns [WATE79] may be easier and more efficient than invoking general axiomatic and algebraic mechanisms to solve recurrence relations.

$$
\begin{aligned}
P_1 &: (s,1,2,f) \\
D[P_1] &: (a - b + h > 0.0) \\
C[P_1] &: \mathrm{AREA} = ? \\
&\quad\ \mathrm{ERROR} = \mathrm{true}
\end{aligned}
$$

$$
\begin{aligned}
P_2 &: (s,1,3,4,5,6,10,f) \\
D[P_2] &: (a - b + h \leq 0.0) \text{ and } (a - b + h > 0.0) \\
&\quad = \text{false *** infeasible path ***}
\end{aligned}
$$

$$
\begin{aligned}
P_3 &: (s,1,3,4,5,6,(7,8,9,6)^{+},10,f) \\
D[P_3] &: (a - b + h \leq 0.0) \text{ and } (k_e = \mathrm{int}(-a/h + b/h) \\
C[P_3] &: \mathrm{AREA} = (f[0] + a*f[1] + 2.0*a*f[2] + f[0]*k_e - f[1]*h*k_e/2.0 + a*f[1]*k_e \\
&\qquad + f[1]*h*k_e**2/2.0 + f[2]*h**2*k_e/6.0 - a*f[2]*h*k_e + a**2*f[2]*k_e \\
&\qquad + f[2]*h**2*k_e**2/2.0 + a*f[2]*h*k_e**2 + f[2]*h**2*k_e**3/3.0) * h \\
&\qquad = f[0]*h + a*f[1]*h + 2.0*a*f[2]*h + f[0]*h*k_e + a*f[1]*h*k_e - f[1]*h**2*k_e/2.0 \\
&\qquad - a*f[2]*h**2*k_e + a**2*f[2]*h*k_e + f[1]*h**2*k_e**2/2.0 + f[2]*h**3*k_e/6.0 \\
&\qquad + a*f[2]*h**2*k_e**2 + f[2]*h**3*k_e**2/2.0 + f[2]*h**3*k_e**3/3.0 \\
&\quad\ \mathrm{ERROR} = \mathrm{false}
\end{aligned}
$$

Figure 4: Path Domains and Computations for RECTANGLE.

In the purest sense, the path domain and path computation are all that need be provided by symbolic evaluation. To do further analysis, however, it is desirable to simplify the symbolic representations and to determine the consistency of the PC.

Simplification can be done by converting the symbolic expressions into canonical forms. There are several available algebraic manipulation systems [BOGE75, BROW73, RICH78] that can be used to accomplish this simplification. A canonical form for the symbolic value of each output parameter might be one in which like terms are grouped together and terms are ordered first by degree and then lexically. The PC might be put into conjunctive normal form and each relational expression put into a canonical form. This canonical form might be one in which the constant term is on the right-hand-side of the relational operator and the left-hand-side has the same form as that for an output parameter. To enhance readability, we have simplified the output from symbolic evaluation to these canonical forms in all the examples given in this paper.

As noted above, only a subset of the paths in a program are executable and, therefore, it is desirable to determine whether or not the PC is consistent. One approach to this problem employs a theorem proving system. We refer to this as the axiomatic technique since it is based upon the axioms of predicate calculus. Another approach, referred to as the algebraic technique, treats the PC as a system of constraints and uses one of several algebraic methods -- such as a gradient hill-climbing or linear programming algorithms -- to solve this system of constraints. The ATTEST system [CLAR76,78], for example, uses a linear programming algorithm [LAND73] and thus employs the algebraic technique. The advantage of choosing this technique is that a solution is provided when the PC is determined to be consistent. This solution serves as test data to execute the path. Both the axiomatic and algebraic techniques work well on the simple constraints that are generally created during symbolic evaluation. No method, however, can solve all arbitrary systems of constraints [DAVI73]. In some instances, PC consistency or inconsistency can not be determined; the symbolic representations for such a path can be provided, but whether or not the path can be executed is unknown.

3. 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. For the most part, testing research has divided the testing process into path selection and test data selection components. This division 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. <u>Statement coverage</u> requires that each statement in the program occurs at least once on one of the selected paths. Likewise, <u>branch coverage</u> requires that each branch predicate occurs at least once on one of the selected paths and <u>path coverage</u> requires that all paths be selected. Branch coverage implies statement coverage, while path coverage implies branch coverage. Thus, these three measures provide an ascending scale of confidence in testing. 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 simply limit loop iterations. The EFFIGY system [KING76] generates all paths with a bound specified on the number of loop iterations. The ATTEST system strives for statement, branch, or path coverage but 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 other than in loop iterations are in different classes. In addition, two paths that differ only in loop traversals 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 provides a snapshot of 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 information concerning the flow of data through the routine. Data flow techniques [LASK79, NTAF81, RAPP82] require the selection of some subpath(s) from a definition of a variable to a use of that variable. Rapps and Weyuker [RAPP82] have described a partial ordering on a family of data flow techniques for path selection. Figure 5 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 6. Use coverage requires the selection of some subpath from each definition of a variable to each use of that variable. The following paths, therefore, satisfy use coverage: (1,2,3,5,6,8), (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 above, 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, a number of infeasible paths will be generated unless data flow analysis and symbolic evaluation techniques are paired together.

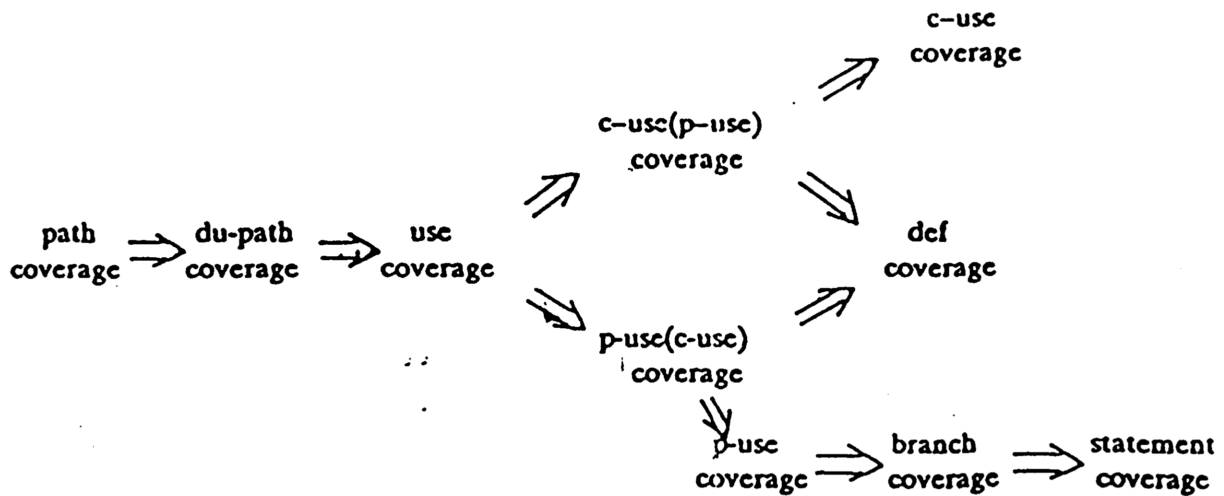

Figure 5: Data Flow Testing Criteria.

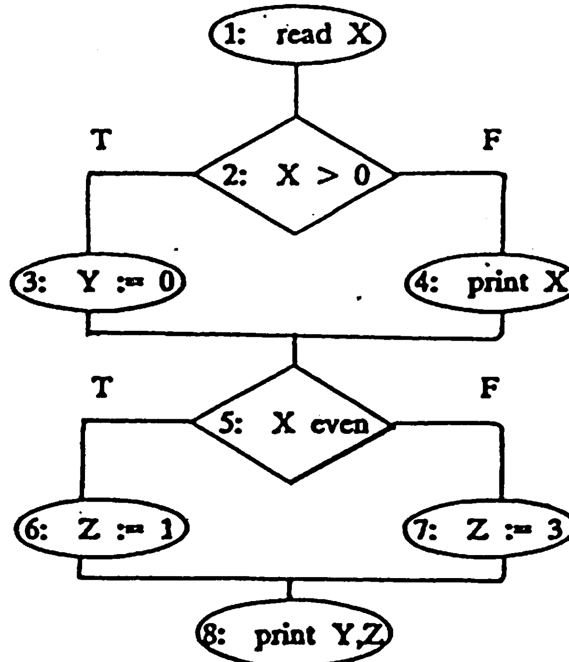

Figure 6: Data Flow Testing Example.

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 tested in order to detect possible perturbations. As an example, consider the flow chart in Figure 7. 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 the error would 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. The perturbations of a statement can be represented by using modified symbolic evaluation techniques. Perturbation testing is currently being implemented as an extension to the ATTEST symbolic evaluation system.

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.

Given a particular path through a program to be tested, a test data selection technique provides guidance in the selection of test data for that path. The symbolic representation of a path can be used as the basis on which to select such data. The most straightforward technique simply examines the PC to determine a solution – that is, test data to execute the path. As noted previously, SELECT [BOYE75] and ATTEST are two symbolic execution systems that generate such test data by using an algebraic technique for determining PC consistency.

In addition to purely random methods, 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.

More systematic 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 path selection error, which occurs when a program incorrectly determines the conditions under which a path is executed, or 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. A number of test data selection techniques focus on the detection or 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; 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

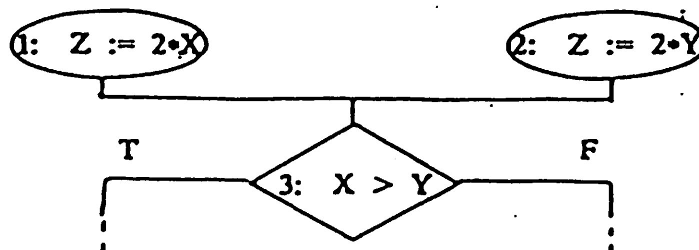

Figure 7: Peturbation Testing Example.

a common phenomenon of testing. A goal, therfore, 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 is the fourth error, which is also a missing path error. A correct check for valid 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 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 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 is 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 must confidence [DEMI78]. 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 is both zero and nonzero. 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 8 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 but not on the closed border and thus will not be selected by domain testing.

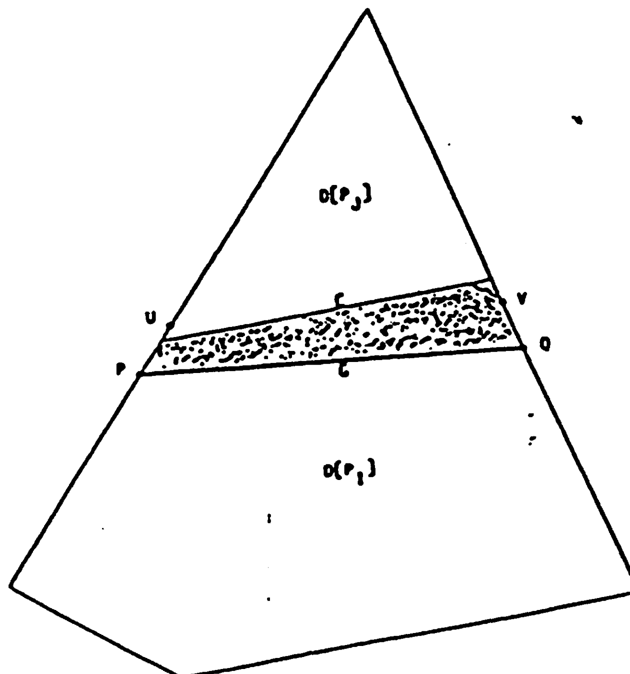

Figure 8: Domain Testing Strategy.

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 = 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 ≤ 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 9: 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 the partition analysis method [RICH81a].

In sum, the symbolic representations created by symbolic evaluation appear to be quite useful in determining what test data should be selected in order to have confidence in a path's reliability. This is a promising, yet relatively new, research area that should be explored further.

4. VERIFICATION APPLICATIONS

Formal verification methods use symbolic evaluation techniques to assist in proving the correctness of programs. Typically, input, output, and loop invariant assertions must be supplied. Verification conditions are then created by symbolically evaluating the code between two adjacent assertions. These verification conditions must then be shown to be true based on the semanitics of the programming language and any required application-dependent axioms. This process [FLOY67, HANT76, HOAR71, LOND75] and a number of related approaches to verification have been frequently described in the literature and will not be discussed here.

Instead, this section discusses some alternative verification approaches. First, some less comprehensive verification techniques that are used to detect or certify the absence of particular program properties are described. Then partition analysis, a method that integrates testing and verification is presented in some detail. Although still based on symbolic evaluation, this method uses a quite different approach to verification. One of the advantages of this method is that it can be applied to a number of different types of specification and design languages.

4.1. Certification

The symbolic representations that are generated for a path can quite naturally be used for certification. The path computation often provides a concise functional representation of the output for the entire path domain. Normal execution, on the other hand, only provides particular output values for particular input values. Examination of the path computation as well as the path condition is often useful in uncovering program errors. In RECTANGLE, for example, examination of $C[P_3]$ would most likely reveal the erroneous use of multiplication rather than exponentiation in statement 5. This method of certification is referred to as <u>symbolic testing</u> [HOWD76]. Symbolic testing is a particularly beneficial feature for scientific applications, where it is often extremely difficult to manually compute the intended result accurately due to both the complexity of the computation and the number of significant digits required for the input values.

Symbolic evaluation can also be applied in certifying the absence of specific types of program errors. At appropriate points in a routine, expressions describing error conditions can be interpreted and checked for consistency with the PC just as branch predicates are interpreted and checked. Consistency implies the existence of input values in the path domain that would cause the described error. Inconsistency implies that the error condition could not occur for any element in the path domain. While normal execution of a path may not uncover a potential run-time error, symbolic evaluation of a path can detect the presence or certify the absence of some errors for all possible inputs to the path.

The ATTEST system automatically generates expressions for predefined error conditions whenever it encounters certain program constructs. For instance, whenever a nonconstant divisor is encountered, a relational expression comparing the symbolic value of the divisor to zero is created. This expression is then temporarily conjoined to the PC. If the resulting PC is consistent, then input values exist that would cause a division by zero error and an error report is issued. If the resulting PC is inconsistent, then this potential run-time error could not occur on this path. After checking for consistency, the expression for the error condition is removed from the PC before symbolic evaluation continues.

Path verification of assertions is another method of certifying the absence of errors. Instead of predefining the error conditions, user-created assertions define conditions that should be true at designated points in the routine. An error exists if an assertion is not true for all elements of the path domain. When an assertion is encountered during symbolic evaluation, the complement of the assertion is interpreted and conjoined to the PC. Inconsistency of the resulting PC implies that the assertion is valid for the path, while consistency implies that the assertion is invalid for the routine.

Checking error conditions during symbolic evaluation provides conclusions about the occurence of that error on a specific path, and likewise for the validity of user-provided assertions. When certification is done for all (classes of) paths, conclusions can be drawn about the entire routine. Thus, if a routine is annotated with assertions that specify the intended function of the routine and these are shown to be valid for all paths, the correctness of the routine has been verified.

4.2. The Partition Analysis Method

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. The <u>partition analysis method</u> incorporates information derived from such a specification with information derived from the corresponding implementation to assist in determining program reliability. This information is embodied in the <u>procedure partition</u>, which is derived by applying symbolic evaluation techniques to both

the specification and the implementation. The procedure partition describes the similarities and differences between the two representations. The procedure partition divides the set of input data for the procedure into subdomains so that the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. By forming these subdomains, the procedure's domain is decomposed into more manageable units, as is the task of evaluating program reliability. Information related to each subdomain is used to verify consistency between the specification and the implementation for the subdomain. Traditionally, program verification approaches have been limited to specifications that are intimately tied to the implementation. Partition analysis, while borrowing from these verification techniques, accepts an independent specification and thus is applicable with most of the proposed software development methodologies [BAUE79, CAIN75, SILV79, WARN74, WIRT73, YOUR75]. In addition to verification, the subdomains are used to guide in the selection of test data. Most testing methods select data based only on the program structure and thus test the actual behavior of the implementation rather than its intended behavior. By basing test data selection on the procedure partition, however, partition analysis derives a set of test data that characterizes both the specification and the implementation, and consequently both the intended and actual behavior. Finally, the verification and testing processes of partition analysis enhance each other; the testing of some elements in the procedure subdomain may assist in verification, while the verification process may direct the selection of test data.

In Figure 10, a specification is given for a procedure that determines whether a number is prime. This specification of PRIME was developed by formalizing the simple mathematical properties of a prime number. An implementation of PRIME in Ada [WEGN80] appears in Figure 11. This implementation makes use of several facts that improve on efficiency. The procedure PRIME is used to illustrate the partition analysis method. In the remainder of this section, the formation of the procedure partition for PRIME is described and then the application of partition analysis verification and partition analysis testing are discussed. The full application of partition analysis to PRIME is provided elsewhere [RICH81b].

4.2.1. Construction of the Procedure Partition

As is evident in the PRIME example, a specification and an implementation are intended to be descriptions of the same function at different levels of abstraction. To facilitate a comparison of these two descriptions, the partition analysis method uses symbolic evaluation techniques to decompose both descriptions into functional representations, called the specification partition and the implementation partition. If the specification and implementation have the same domain and codomain and are defined for the same values, then the two descriptions of the procedure have consistent interfaces and are said to be compatible. For the following discussion, we assume that compatibility holds, although minor violations of compatibility can be handled by partition analysis with appropriate modifications [RICH81c]. The specification and

```
     procedure PRIME( N: in integer inset {2...}) return boolean =
       — PRIME returns true if N is prime or false if N is not prime
s    begin
     return case
1              N = 2 →
2                 true;
3              otherwise →
                  — if N has no factor ≤ N-1, N has no factor
4                 forall< i: integer inset {2..N-1} | (N mod i ≠ 0) >;
               endcase;
f    end PRIME;
```

Figure 10: Specification of PRIME.

```
    function PRIME( N: in integer range 2..max'int) return boolean is
      -- PRIME returns true if N is prime or false if N is not prime
      FAC: integer;
      ISPRIME: boolean;
s   begin
1     if N mod 2 = 0 or N mod 3 = 0 then
        -- if N is even and N ≠ 2 or N is divisible by 3 and N ≠ 3, N is not PRIME
2       ISPRIME := (N < 4);
      else
        -- if N is odd, any FACtor of N is odd
        -- if N is not divisible by 3, N has no FACtor in sequence 9,15,21,...
        -- if N has no FACtor ≤ sqrt(N), N has no FACtor
        -- loop checking for FACtors in the sequence 5,7,11,13,17,19,...
3       ISPRIME := true;
4       FAC := 5;
5       while FAC**2 ≤ N loop
6         if N mod FAC = 0 or N mod (FAC+2) = 0 then
7           ISPRIME := false;
            exit;
          else
8           FAC := FAC + 6;
          endif;
9       endloop;
      endif;
10    return ISPRIME;
f   end PRIME;
```

Figure 11: Implementation of PRIME.

---

implementation partitions are combined to form the procedure partition, which forms the basis for comparison of the specification and implementation within the partition analysis method.

Symbolic evaluation of the implementation P provides the implementation partition, which is the set of domains and computations of the (classes of) paths in P,

$$\{ (D[P_J], C[P_J]) \mid 1 \le J \le N \}.$$

The implementation partition of PRIME is shown in Figure 12. Symbolic evaluation and loop analysis techniques have been extended to be applicable to several specification languages [COHE82, GOUR81, RICH81c]. Using these techniques, a feasible sequence of statements through a specification, referred to as a subspec, is evaluated in terms of symbolic names assigned to the input values. A specification can then be decomposed into a finite set of (classes of) subspecs. Each subspec S_I is described by a subspec domain $D[S_I]$ and a subspec computation $C[S_I]$. The specification partition that represents a specification S is the set of domains and computations of the (classes of) subspecs in S

$$\{ D[S_I], C[S_I]) \mid 1 \le I \le M \}.$$

Figure 13 provides the specification partition of PRIME.

The specification and implementation impose two partitions on a procedure, which represent two ways in which the procedure may be divided. It is not surprising to find a subspec domain and a path domain that are equal. The testing and verification of the subspec and path computations can then be considered over this subdomain as a whole. On the other hand, there are often differences between these two partitions; a subspec domain may overlap with more than one path domain or vice versa. Such a discrepancy may be due to an error. Alternatively, this may not be indicative of an error, but rather occurs because the specification is a more abstract description of the problem than the implementation. PRIME provides an

P_1: (s,1,2,10,f)
$D[P_1]$: ((int(n/2)*2 − n = 0) or (int(n/3)*3 − n = 0))
$C[P_1]$: (n < 4)

P_2: (s,1,3,4,5,9,10,f)
$D[P_2]$: (n < 25) and (int(n/2)*2 − n ≠ 0) and (int(n/3)*3 − n ≠ 0)
$C[P_2]$: true

P_3: (s,1,3,4,(5,6,8),5,6,7,9,10,f)
$D[P_3]$: (n ≥ 25) and (int(n/2)*2 − n ≠ 0) and (int(n/3)*3 − n ≠ 0)
and exists< k_e:= 1... | ((int(n/(6*k_e−1))*(6*k_e−1) − n = 0)
or (int(n/(6*k_e+1))*(6*k_e+1) − n = 0))
and forall< k:= 1..k_e−1 | (int(n/(6*k−1))*(6*k−1) − n ≠ 0)
and (int(n/(6*k+1))*(6*k+1) − n ≠ 0) and (36*k**2 + 60*k − n ≤ −25) > >
$C[P_3]$: false

P_4: (s,1,3,4,$(5,6,8)^+$,5,9,10,f)
$D[P_4]$: (n ≥ 25) and (int(n/2)*2 − n ≠ 0) and (int(n/3)*3 − n ≠ 0)
and exists< k_e:= 1... | (int(n/(6*k_e−1))*(6*k_e−1) − n ≠ 0)
and (int(n/(6*k_e+1))*(6*k_e+1) − n ≠ 0) and (36*k_e**2 + 60*k_e − n > −25)
and forall< k:= 1..k_e−1 | (int(n/(6*k−1))*(6*k−1) − n ≠ 0)
and (int(n/(6*k+1))*(6*k+1) − n ≠ 0) and ((36*k**2 + 60*k − n ≤ − 25) > >
$C[P_4]$: true

Figure 12: Implementation Partition of PRIME.

S_1: (s,1,2,f)
$D[S_1]$: (n = 2)
$C[S_1]$: true

S_2: (s,3,4,f)
$D[S_2]$: (n ≥ 3)
$C[S_2]$: forall< i:= 2..n−1 | (int(n/i)*i − n ≠ 0) >

Figure 13: Specification Partition of PRIME.

---

excellent illustration of the variation that can occur between the different levels of abstraction. The one element in the "N = 2" subspec domain and some of the elements in the "otherwise" subspec domain, those for which N is divisible by 2 or 3, are grouped in the "N mod 2 = 0 or N mod 3 = 0" path domain, hence a path domain overlaps with more than one subspec domain. The other elements in the "otherwise" subspec domain, those for which N is not divisible by 2 or 3, are in the remaining path domains, hence a subspec domain overlaps with more than one path domain.

It is clearly not adequate to use either the specification partition alone or the implementation partition alone as the basis for demonstrating program reliability. Both partitions must be considered or potentially useful information is lost. The procedure partition, which takes into account both the specification and the implementation, is constructed by overlaying, or intersecting, these two partitions. Each subdomain so formed is the set of input data for which a subspec and a path are mutually applicable. These subdomains are constructed by

taking the pairwise intersection of the set of subspec domains and the set of path domains. The nonempty intersection of a subspec domain $D[S_I]$ and a path domain $D[P_J]$ is referred to as a procedure subdomain, and denoted D_{IJ} – that is, $D_{IJ} = D[S_I] \cap D[P_J] \neq \emptyset$. To help conceptualize the formation of this partition, Figure 14 shows a hypothetical example of the procedure subdomains that would result by overlaying partitions of the specification and implementation domains.

Associated with each procedure subdomain are two computations, the subspec computation and the path computation, which are intended to specify equal output values for all elements to which they both apply – that is, all elements in the procedure subdomain. The computation difference C_{IJ} for a procedure subdomain D_{IJ} is the difference between the subspec computation $C[S_I]$ and the path computation $C[P_J]$ – thus, $C_{IJ} = C[S_I] - C[P_J]$. In general, the computation difference is a vector, where each component is derived by subtracting a component in the path computation from the corresponding component in the subspec computation. Thus, for the output value z, $C_{IJ}.z = C[S_I].z - C[P_J].z$. In procedure PRIME, for example, the only output value is for the parameter PRIME and the computation difference for a procedure subdomain D_{IJ} is $C_{IJ}.PRIME = C[S_I].PRIME - C[P_J].PRIME$.

The procedure partition is composed of the procedure subdomains and the associated computation differences,

$$\{ (D_{IJ}, C_{IJ}) \mid 1 \leq I \leq M \text{ and } 1 \leq J \leq N \text{ and } D[S_I] \cap D[P_J] \neq \emptyset \}.$$

The procedure partition created for PRIME appears in Figure 15. The procedure partition provides the basis for the application of both verification and testing techniques within the partition analysis method. Each procedure subdomain and its associated computation difference are of interest and should be verified and tested independently of the rest of the procedure partition. Procedure subdomains appear to be the largest units of input data that can be analyzed independently; yet they provide a practical decomposition of the testing and verification process.

4.2.2. Partition Analysis Verification

Partition analysis verification attempts to determine consistency between an implementation and a specification. To realize the function described by the specification, the implementation must compute the specified output values for each input value in the domain. This property is referred to as equivalence.

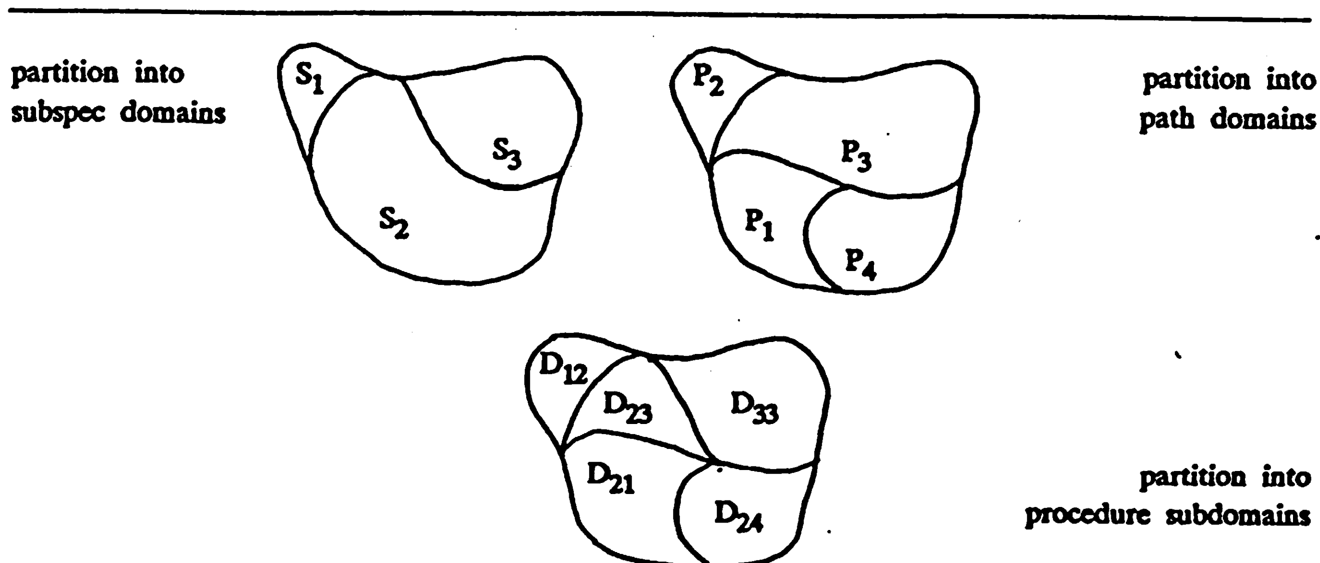

Figure 14: Hypothetical Procedure Subdomains.

$D_{11} = (n = 2)$
$C_{11} = (\text{true}) - (n < 4)$

$D_{21} = (n \geq 3) \text{ and } ((\text{int}(n/2)*2 - n = 0) \text{ or } (\text{int}(n/3)*3 - n = 0))$
$C_{21} = (\text{forall}< i:= 2..n-1 \mid (\text{int}(n/i)*i - n \neq 0) >) - (n < 4)$

$D_{22} = (n \geq 3) \text{ and } (n < 25) \text{ and } (\text{int}(n/2)*2 - n \neq 0) \text{ and } (\text{int}(n/3)*3 - n \neq 0)$
$C_{22} = (\text{forall}< i:= 2..n-1 \mid (\text{int}(n/i)*i - n \neq 0) >) - (\text{true})$

$D_{23} = (n \geq 25) \text{ and } (\text{int}(n/2)*2 - n \neq 0) \text{ and } (\text{int}(n/3)*3 - n \neq 0)$
$\quad \text{and exists}< k_e:= 1... \mid ((\text{int}(n/(6*k_e-1))*(6*k_e-1) - n = 0)$
$\quad \text{or } (\text{int}(n/(6*k_e+1))*(6*k_e+1) - n = 0))$
$\quad \text{and forall}< k:= 1..k_e-1 \mid (\text{int}(n/(6*k-1))*(6*k-1) - n \neq 0)$
$\quad \text{and } (\text{int}(n/(6*k+1))*(6*k+1) - n \neq 0) \text{ and } (36*k**2 + 60*k - n \leq -25) > >$
$C_{23} = (\text{forall}< i:= 2..n-1 \mid (\text{int}(n/i)*i - n \neq 0) >) - (\text{false})$

$D_{24} = (n \geq 25) \text{ and } (\text{int}(n/2)*2 - n \neq 0) \text{ and } (\text{int}(n/3)*3 - n \neq 0)$
$\quad \text{and exists}< k_e:= 1... \mid (\text{int}(n/(6*k_e-1))*(6*k_e-1) - n \neq 0)$
$\quad \text{and } (\text{int}(n/(6*k_e+1))*(6*k_e+1) - n \neq 0) \text{ and } (36*k_e**2 + 60*k_e - n > -25)$
$\quad \text{and forall}< k:= 1..k_e-1 \mid (\text{int}(n/(6*k-1))*(6*k-1)- n \neq 0)$
$\quad \text{and } (\text{int}(n/(6*k+1))*(6*k+1) - n \neq 0) \text{ and } ((36*k**2 + 60*k - n \leq - 25) > >$
$C_{24} = (\text{forall}< i:= 2..n-1 \mid (\text{int}(n/i)*i - n \neq 0) >) - (\text{true})$

Figure 15: Procedure Partition of PRIME.

Definition: Given a specification $S: X \rightarrow Z$ and an implementation $P: X \rightarrow Z$, with $D[P] = D[S]$, P is equivalent with S if for all $x \in D[S]$, $P(x) = S(x)$.

Equivalence between a procedure implementation and a specification implies that the implementation is correct with respect to the specification. This property can be related to the procedure partition. Given an input vector x, suppose $x \in D_{IJ}$ -- thus, $x \in D[S_I]$ and $x \in D[P_J]$ and subspec S_I and path P_J are applicable for this input vector. Then $S(x) = P(x)$, if and only if the subspec S_I and the path P_J compute equal output values -- $C[S_I](x) = C[P_J](x)$. The subspec computation $C[S_I]$ and the path computation $C[P_J]$ are equal when restricted to their mutual procedure subdomain D_{IJ}, if for all $x \in D_{IJ}$, $C_{IJ}(x) = 0$; this is denoted $C_{IJ} \mid D_{IJ} = 0$. The equivalence of an implementation and a specification can thus be restated in terms of the equality of the computations over procedure subdomains.

Given a specification $S: X \rightarrow Z$ and an implementation $P: X \rightarrow Z$, with $D[P] = D[S]$, P is equivalent with S if and only if for all I and J, $1 \leq I \leq M$ and $1 \leq J \leq N$, such that $D[S_I] \cap D[P_J] \neq \emptyset$, $C_{IJ} \mid D_{IJ} = 0$.

Thus, equivalence can be demonstrated in terms of the procedure partition by proving that for each procedure subdomain, the corresponding subspec and path computations produce equal values for all elements of this mutual procedure subdomain.

The equality of the subspec computation $C[S_I]$ and the path computation $C[P_J]$ over the procedure subdomain D_{IJ} is determined by demonstrating whether or not each component of the associated computation difference C_{IJ} is zero when it is restricted to the procedure subdomain. In many cases, the simplification of the computation difference $C_{IJ}.z$ reduces that expression to zero, in which case the two computations $C[S_I].z$ and $C[P_J].z$ are symbolically identical and thus equal over any domain. Two computations $C[S_I].z$ and $C[P_J].z$ are also

equal over the associated procedure subdomain D_{IJ} if the condition defining D_{IJ} implies that the computation difference $C_{IJ}.z$ is zero. Proving that $D_{IJ} \rightarrow C_{IJ} \mid D_{IJ} = 0$ is attempted through the application of proof techniques such as those employed by program verifiers.

The process of applying partition analysis verification to the procedure PRIME is relatively complicated, primarily due to the properties that are used in producing the efficient implementation. Just the proof developed for procedure subdomain D_{21} is discussed here and shown in Figure 16. In this proof the computation difference C_{21} will clearly vary depending on whether $(n < 4)$ or $(n \geq 4)$. This prompts the further division of the procedure subdomain into two subsets – D_{21a}, which contains those elements in D_{21} for which $(n < 4)$, and D_{21b}, which contains those elements in D_{21} for which $(n \geq 4)$. The proof is then done in two parts, $D_{21a} \rightarrow C_{21} = 0$ and $D_{21b} \rightarrow C_{21} = 0$. The condition defining D_{21a} implies that the sequence 2..n–1 contains only the element 2 (this fact is denoted 21a–1 in the proof) and also that (trunc(n/2)*2 – n ≠ 0) (21a–2). These two facts imply that forall < i: = 2..n–1 | (trunc(n/i)*i – n ≠ 0) > is true (21a–3). The condition defining D_{21a} also implies that $(n < 4)$ is true (21a–4). The facts denoted by (21a–3) and (21a–4) imply that $C_{21} = 0$. A similar proof is generated to show that $D_{21b} \rightarrow C_{21} = 0$. The two proofs serve to demonstrate that $D_{21} \rightarrow C_{21} = 0$. Because this proof was contingent on the further division of the procedure subdomain, it is reasonable to suspect that the differences between the subspec and path computations may vary between the subsets of the procedure subdomain. It is thus important to test elements in both subsets of the procedure subdomain. In general, whenever partition analysis verification must divide a procedure subdomain into subsets and prove that the computation difference is zero over each subset independently, partition analysis testing is directed to select test data from each such subset of the procedure 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 computations restricted over a domain. In general, this problem is undecidable and thus partition analysis verification suffers some of the same drawbacks as other verification approaches. Most verification approaches decompose the implementaton into sequences of statements and employ standard proof techniques to show that assertions are true at points between these sequences. By so doing, failure to prove a single assertion may cause failure to show that any of the implementation is correct. When partition analysis verification fails to prove the equality or inequality of the associated computations for a procedure subdomain, it does not affect the proofs for other procedure subdomains. When proof techniques do fail, testing can provide some assurance of the equality of the computations or find examples of their inequality.

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. Recent experimental results suggest that partition analysis verification is capable of detecting fairly subtle inconsistencies between two descriptions of a procedure.

D_{21}: $(n \ge 3)$ and $((\text{int}(n/2)*2 - n = 0)$ or $(\text{int}(n/3)*3 - n = 0))$

C_{21}: $(\text{forall}< i{:=}\ 2..n-1 \mid (\text{int}(n/i)*i - n \ne 0) >) - (n < 4)$

PARTITION ANALYSIS VERIFICATION:

$D_{21} = \{D_{21} \mid (n < 4)\}$ union $\{D_{21} \mid (n \ge 4)\}$ (21–1)

D_{21a}: $(n \ge 3)$ and $((\text{int}(n/2)*2 - n = 0)$ or $(\text{int}(n/3)*3 - n = 0))$ and $(n < 4)$
$= (n = 3)$ and $((\text{int}(n/2)*2 - n = 0)$ or $(\text{int}(n/3)*3 - n = 0))$

D_{21b}: $(n \ge 3)$ and $((\text{int}(n/2)*2 - n = 0)$ or $(\text{int}(n/3)*3 - n = 0))$ and $(n \ge 4)$
$= (n \ge 4)$ and $((\text{int}(n/2)*2 - n = 0)$ or $(\text{int}(n/3)*3 - n = 0))$

Proof of $D_{21a} \rightarrow C_{21} = 0$:

$(n = 3) \rightarrow (2..n-1 = 2)$ (21a–1)
$(n = 3) \rightarrow (\text{int}(n/2)*2 - n \ne 0)$ (21a–2)
(21a–1) and (21a–2) $\rightarrow$ $\text{forall}< i{:=}\ 2..n-1 \mid (\text{int}(n/i)*i - n \ne 0) >$ (21a–3)
$(n = 3) \rightarrow (n < 4)$ (21a–4)
(21a–3) and (21a–4) $\rightarrow D_{21a} \rightarrow C_{21} = (\text{true}) - (\text{true}) = 0$ (21a–5)

Proof of $D_{21b} \rightarrow C_{21} = 0$:

$(n \ge 4) \rightarrow (2 < n-1)$ and $(3 \le n-1)$ (21b–1)
(21b–1) and $((\text{int}(n/2)*2 - n = 0)$ or $(\text{int}(n/3)*3 - n = 0)) \rightarrow$
$\text{exists}< i{:=}\ 2..n-1 \mid (\text{int}(n/i)*i - n = 0) > \rightarrow$
not $\text{forall}< i{:=}\ 2..n-1 \mid (\text{int}(n/i)*i - n \ne 0) >$ (21b–2)
$(n \ge 4) \rightarrow$ not $(n < 4)$ (21b–3)
(21b–2) and (21b–3) $\rightarrow D_{21} \rightarrow C_{21b} = (\text{false}) - (\text{false}) = 0$ (21b–4)

Proof of $D_{21} \rightarrow C_{21} = 0$:

(21–1) and (21a–5) and (21b–4) $\rightarrow D_{21} \rightarrow C_{21} = 0$

PARTITION ANALYSIS TESTING:

Domain Testing Criteria:

N = 3	$\rightarrow$ on $(n = 3)$, off $(n \ge 4)$, on $(\text{int}(n/3)*3 - n = 0)$, off $(\text{int}(n/2)*2 - n = 0)$
N = 2	$\rightarrow$ off $(n = 3)$
N = 4	$\rightarrow$ on $(n \ge 4)$, on $(\text{int}(n/2)*2 - n = 0)$, off $(\text{int}(n/3)*3 - n = 0)$
N = 5	$\rightarrow$ off $(\text{int}(n/2)*2 - n = 0)$, off $(\text{int}(n/3)*3 - n = 0)$
N = 6	$\rightarrow$ on $(\text{int}(n/2)*2 - n = 0)$, on $(\text{int}(n/3)*3 - n = 0)$
N = 7	$\rightarrow$ off $(\text{int}(n/2)*2 - n = 0)$, off $(\text{int}(n/3)*3 - n = 0)$
N = 996	$\rightarrow$ on $(\text{int}(n/2)*2 - n = 0)$, on $(\text{int}(n/3)*3 - n = 0)$
N = 997	$\rightarrow$ off $(\text{int}(n/2)*2 - n = 0)$, off $(\text{int}(n/3)*3 - n = 0)$
N = 999	$\rightarrow$ on $(\text{int}(n/3)*3 - n = 0)$, off $(\text{int}(n/2)*2 - n = 0)$
N = 1000	$\rightarrow$ on $(\text{int}(n/2)*2 - n = 0)$, off $(\text{int}(n/3)*3 - n = 0$

Computation Testing Criteria:

N = 3	$\rightarrow C[S_3] = C[P_2] =$ true, minimum n, maximum iteration of forall
N = 4	$\rightarrow C[S_3] = C[P_2] =$ false, minimum iteration of forall
N = 1000	$\rightarrow$ maximum n

Figure 16: Partition Analysis of PRIME (Procedure Subdomain D_{21}).

4.2.3. Partition Analysis Testing

Within the partition analysis method, the verification process is complemented 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 computation and domain testing techniques. Figure 16 shows the test data selected for procedure subdomain D_{21} of PRIME along with the reason each datum was selected. Figure 17 shows all the test data selected for PRIME by partition analysis testing.

The application of computation testing to the procedure partition involves selecting data based on an analysis of the computation differences. To demonstrate the run-time properties, it is important to select test data for which the path computation itself is sensitive to error. Further, there is always a chance (a very good one at that) that the subspec computation is incorrect, thus test data for which it is sensitive to error must also be selected. Selecting sensitive test data for both computations may draw attention to an error that might otherwise remain undetected. Thus, the computation testing techniques are applied to both the subspec computation and the path computation as represented in the unsimplified form of the computation difference.

D_{11}: Domain Testing Criteria:
N = 1, N = 2, N = 3
Computation Testing Criteria:
N = 2

D_{21}: Domain Testing Criteria:
N = 2, N = 3, N = 4, N = 5, N = 6, N = 7,
N = 996, N = 997, N = 999, N = 1000
Computation Testing Criteria:
N = 3, N = 4, N = 1000

D_{22}: Domain Testing Criteria:
N = 2, N = 3
Computation Testing Criteria:
N = 5, N = 23

D_{23}: Domain Testing Criteria:
N = 24, N = 25, N = 27, N = 35, N = 49,
N = 841, N = 899, N = 961, N = 997
Computation Testing Criteria:
N = 25, N = 961

D_{24}: Domain Testing Criteria:
N = 24, N = 27, N = 839, N = 841, N = 997
Computation Testing Criteria:
N = 27, N = 997

Figure 17: Partition Analysis Testing of PRIME.

The application of domain testing to the procedure partition involves analyzing the procedure subdomains and selecting test data near the boundaries of those domains. Since each border of a path domain boundary is a border of some procedure subdomain, testing the borders of the procedure subdomains necessarily tests the path domain borders. By testing the borders of the procedure subdomains rather than simply those of path domains, the differences between the path domains and the subspec domains are also tested. Moreover, the testing of procedure subdomains enables the detection of missing path errors as well as path selection errors (assuming the specification is correct).

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.

5. CONCLUSION

In this paper symbolic evaluation is described and demonstrated on two examples. Symbolic evaluation is of interest because it is widely used to aid in testing and verification.

For the path selection aspects of testing, symbolic evaluation is useful in determining path feasibility for the control 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 better, as well as more integrated, techniques must be developed.

Formal verification techniques have always employed symbolic evaluation methods to help formulate the verification conditions that are to be proven. There are several less comprehensive ways in which verification can be done in the context of symbolic evaluation. Another alternative is the partition analysis method, which is a verification technique that is applicable to a wide class of specification and design languages. Moreover, the basic method integrates testing and verification.

For the most part, current research is addressing the issues of path selection, test data selection, and verification 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.

BOGE75 R. Bogen, "MACSYMA Reference Manual", The Mathlab Group, Project MAC, Massachusetts Institute of Technology, 1975.

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.

BROW73 W.S. Brown, Altran User's Manual, 1, Bell Telephone Laboratories, 1973.

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.

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

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

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

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, pp.25-32.

DAVI73 M. Davis, "Hilbert's Tenth Problem is Unsolvable", American Math. Mon., 80, March 1973, 233-269.

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.

HANT76 S.L. Hantler and J.C. King, "An Introduction to Proving the Correctness of Programs," Computing Surveys, 8,3, September 1976, 331-353.

HOAR71 C.A.R. Hoare, "Proof of a Program: FIND," Communications of the ACM, 14,1, January 1971, 39-45.

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

HOWD76 W.E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, SE-2, 3, September 1976, 208-215.

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.

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.

LAND73 A.H. Land and S. Powell, FORTRAN Codes for Mathematical Programming, John Wiley and Sons, New York, New York, 1973.

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.

RICH78 D.J. Richardson, L.A. Clarke, and D.L. Bennett, "SYMPLR, SYmbolic Multivariate Polynomial Linearization and Reduction", University of Massachusetts, Department of Computer and Information Science, Technical Report 78-16, July 1978.

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, "Examples of the Application of the Partition Analysis Method," Department of Computer and Information Science, University of Massachusetts, TN-48, August 1981.

RICH81c D.J. Richardson, "A Partition Analysis Method to Demonstate Program Reliability," Ph.D. Dissertation, University of Massachusetts, September 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.

WATE79 R.C. Waters, "A Method for Analyzing Loop Programs", IEEE Transactions on Software Engineering, SE-5, 3, May 1979, 237-247.

WEGN80 P. Wegner, Programming with Ada: An Introduction by Means of Graduated Examples, Prentice-Hall, Inc., 1980.

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.