84-20
Renumbered 84-9

Applications of Symbolic Evaluation*

Lori A. Clarke
Debra J. Richardson
University of Massachusetts

Symbolic evaluation is a program analysis method that represents a program's computations and domain by symbolic expressions. In this paper a general functional model of a program is first presented. Then, three related methods of symbolic evaluation, which create this functional description from a program, are described: path-dependent symbolic evaluation provides a representation of a specified path; dynamic symbolic evaluation, which is more restrictive but less costly than path-dependent symbolic evaluation, is a data-dependent method; and global symbolic evaluation, which is the most general yet most costly method, captures the functional behavior of an entire program when successful. All three methods have been implemented in experimental systems. Some of the major implementation concerns, which include effectively representing loops, determining path feasibility, dealing with compound data structures, and handling routine invocations, are explained. The remainder of the paper surveys the range of applications to which symbolic evaluation techniques are being applied. The current and potential role of symbolic evaluation in verification, testing, debugging, optimization, and software development is explored.

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. The primary components of such an environment will include validation tools to detect errors and determine consistency, as well as development tools to assist in design, construction, and optimization. The use of such tools will reduce the development costs and improve the reliability of the resulting program. Several of the tools presently being developed employ a method, called *symbolic evaluation*, that creates a symbolic representation of the functional behavior of a program. This paper describes symbolic evaluation and surveys many of the current applications of this method.

Symbolic evaluation is a program analysis technique that derives an algebraic representation, over the input values, of the computations and their applicable domain. Thus symbolic evaluation describes the relationship between the input data and the resulting values, whereas normal execution computes numeric values but loses information about the way in which these numeric values were derived. There are three basic methods of symbolic evaluation: *path-dependent symbolic evaluation* describes data dependencies for a specified path; *dynamic symbolic evaluation* produces a trace of the data dependencies for particular input data; *global symbolic evaluation* represents the data dependencies for all paths in a program. When further analyzed, the algebraic representations produced by symbolic evaluation provide the basis for a wide range of applications, including verification, testing, debugging, program optimization, and program development.

Formal verification techniques have typically applied symbolic evaluation techniques to develop verification conditions. (Formal verification has been extensively described in the literature and is not discussed further in this paper.) There are a number of less comprehensive verification techniques that have used symbolic evaluation to certify the correctness of selected program properties. In addition, some current work is being directed at developing methods that integrate testing and formal verification, based upon; symbolic evaluation.

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

*This work was supported by the National Science Foundation under grants NSFMCS 81-04202 and 83-03320

Address correspondence to Professor Lori Clarke, Computer and Information Science Department, University of Massachusetts, Amherst, MA 01003.

The Journal of Systems and Software 5, 15–35 (1985)

0164-1212/85/$3.30

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, there has been recent research on developing systematic test data selection techniques that can either eliminate certain classes of errors or provide a quantifiable error bound. Many of these path selection and test data selection techniques base their analysis on the information provided by symbolic evaluation. Moreover, if testing reveals an error, debugging techniques that are based on symbolic evaluation can be used to search for the cause of the error.

Symbolic evaluation also provides information that is useful in program optimization and, if applied early in the software development process, in program development. It is thus a tool that can be employed throughout the software development lifecycle and made wide use of within an automated programming environment.

The next section of this paper introduces the basic concepts of symbolic evaluation as well as some terminology. The three methods of symbolic evaluation are then described. Examples of the three methods are given to demonstrate their corresponding strengths and weaknesses. The third section discusses implementation considerations related to all three methods, while the fourth section describes some of the applications of symbolic evaluation.

2. GENERAL METHODS

This section presents some concepts fundamental to symbolic evaluation. Some terminology is introduced and general descriptions of each of the three methods are provided. Initially, these descriptions are restricted to single routines and to routines whose input and output are done only via parameters. These restrictions are made merely to simplify the presentation. The modifications necessary to eliminate these restrictions are addressed later.

2.1 Basic Concepts

A routine R can be viewed as a function that maps elements in a domain X into elements in a range Z. An element in X is a vector x with specific input values, $x = (x_1, x_2, \ldots, x_M)$, and corresponds to a single point in the M-dimensional input space X. Likewise, R(x) in Z is a vector z with specific output values, $z = (z_1, z_2, \ldots, z_N)$, and corresponds to a single point in the N-dimensional output space Z. A routine's variables, which store input, intermediate and output values, are represented by a vector $y = (y_1, y_2, \ldots, y_W)$.

Program analysis methods typically represent a routine by a directed graph, called a *control flow graph* that describes the possible flow of control through the routine. The nodes in the graph, {1, 2, . . . , q}, represent executable statements. Figure 1 presents RECTANGLE, a routine that is used below to illustrate symbolic evaluation; note that the statements in RECTANGLE are annotated with node numbers. An edge is specified by an ordered pair of nodes, (i, j) that indicates that a transfer of control exists from node i to node j. Associated with each transfer of control are conditions under which such a transfer occurs. The branch predicate that governs traversal of the edge (i, j) is denoted by bp(i, j). For a sequential transfer of control, the branch predicate has the constant value true and thus need not be considered. For a binary condition at node i that transfers control to either node j or k, the branch predicate for edge (i, j) is the complement of the branch predicate for the edge (i, k)—thus,

bp(i, j) = not (bp(i,k)).

In RECTANGLE for example, node 1 precedes nodes 2 and 3 and

bp(1,2) = (H > B − A),
bp(1,3) = (H ≤ B − A).

Note that each IF statement, nested or otherwise, forms a pair of complementary branch predicates. Some conditional statements, such as the FORTRAN computed GO TO or the Pascal and Ada CASE statements, may have more than two successor nodes, and each branch predicate must be represented appropriately. To facilitate analysis, the control flow graph has a single entry point, the start node s, and a single exit point, the final node f. Without loss of generality, a null node can be added to a graph for the start node, and likewise for the final node, if necessary, to accomplish this single-entry, single-exit form. Figure 2 shows the control flow graph for RECTANGLE.

A *subpath* in a control flow graph is a sequence of

Figure 1. Procedure RECTANGLE.

```
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;
1    begin
        -- check for valid input
2       if H > B - A then
3          ERROR := true;
4       else
5          ERROR := false;
6          X := A;
7          AREA := F[0] + F[1]*X + F[2]*X**2;
8          while X + H <= B loop
9             X := X + H;
10            Y := F[0] + F[1]*X + F[2]*X**2;
11            AREA := AREA + Y;
12         end loop;
13         AREA := AREA*H;
14      end if;
15   end RECTANGLE;
```

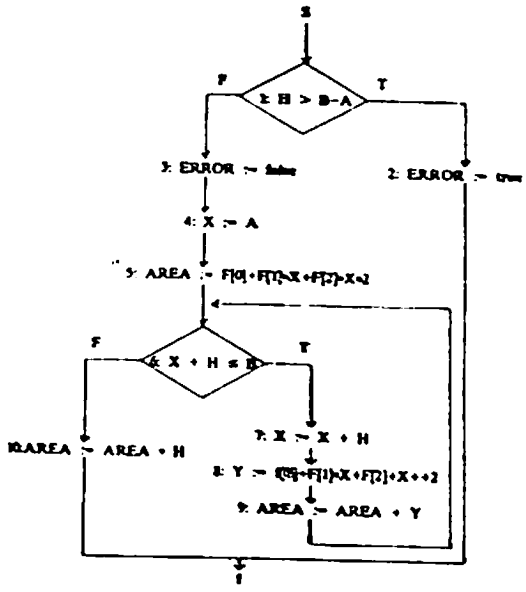


Figure 2. Control flow graph for RECTANGLE.

statements, $(J_i, J_{i+1}, \ldots, J_k)$, where for all k, $i \leq k < t$, J_k is a node in the control flow graph such that there exists edge (J_k, J_{k+1}). A *partial path* is a subpath that begins with the start node and is denoted by P_k, where $P_k = (s, J_1, J_2, \ldots, J_u)$. Hence, for any partial path P_k with $u \geq 1$, $P_k = (P_{k-1}, J_u)$, where $P_0 = (s)$. A *path* is a partial path that ends with the final node and is denoted by P_j, thus $P_j = (s, J_1, J_2, \ldots, J_m, f)$. A routine R is composed of a set of paths $\{P_1, P_2, \ldots, P_R \mid 1 \leq R \leq \infty\}$; there may be an infinite number of paths due to program loops. The routine RECTANGLE contains a loop whose iteration count is dependent on unbounded input values; there are, therefore, an infinite number of paths through RECTANGLE.

There is no guarantee that a sequence of statements representing a path is executable; a path may be nonexecutable due to contradictory conditions governing the transfers of control along the path. Path (s,1,3,4,5,6,10,f) in RECTANGLE is an example of a nonexecutable path, while (s,1,3,4,5,6,7,8,9,6,10,f) is an executable path. The control flow graph is a representation of all possible paths, both executable and nonexecutable, through the corresponding routine.

The *path domain* $D[P_j]$ is the set of all $x \in X$ for which the path P_j could be executed. The path domain of a nonexecutable path, therefore, is empty. Execution of path P_j performs a *path computation* $C[P_j]$ that provides $R(x) = z$ in Z. For each executable path, the path domain and the path computation define the function of the path. Since the executable paths of a routine divide the domain X into disjoint subdomains, the function of a routine R is composed of the set of functions of all executable paths in R.

Symbolic evaluation provides symbolic representations for the path domains and path computations of a routine. For any path, these symbolic representations can be developed incrementally as the statements on a path are interpreted. 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. At any point in the evaluation of path P_j, some partial path $P_k = (s, J_1, J_2, \ldots, J_u)$ has been evaluated. The symbolic values of the variables after evaluation of that partial path are referred to as the *path values* and denoted $PV[P_k]$. The PV (the partial path will not be referenced when unnecessary) is a vector $(s(y_1), s(y_2), \ldots, s(y_w))$, where $s(y_i)$ denotes the current symbolic value of variable y_i. After interpretation of the entire path P_j, the path computation $C[P_j]$ is represented by the components of $PV[P_j]$ that corresponds to the output parameters. The symbolic representation of the path domain can also be formed incrementally by maintaining a representation of the domain of input values for the partial path that has been interpreted so far. This is done by interpreting the branch predicates for the conditional statements on a path. Thus, each such branch predicate is represented by constraints in terms of the symbolic names for the input value. The conjunction of these constraints is called the *path condition* and is denoted $PC[P_k]$. $PC[P_0] = s(bp(s, J_1))$ and $s(bp(J_1, J_2))$ and ... and $s(bp(J_{u-1}, J_u))$, where $s(bp(J_m, J_{m+1}))$, $1 \leq m < u$, denotes the symoblic value of the branch predicate $bp(J_m, J_{m+1})$ when evaluated over the values of the program variables preceding traversal of the edge (J_m, J_{m+1})—that is, over $PV[P_{m-1}]$. The path domain is represented by the path condition after interpretation of the entire path $PC[P_j]$. For nonexecutable paths, the PC is inconsistent, thus no input values exist that could cause execution of the path.

The next three subsections demonstrate how this technique can be employed to derive the symbolic representations of the path computation and path domain in the context of path-dependent symbolic evaluation, dynamic symbolic evaluation, and global symbolic evaluation. The methods differ primarily in their techniques for selecting the paths to be analyzed. With path-dependent symbolic evaluation, each path to be analyzed is chosen by the user or selected by heuristics employed by the system. Dynamic symbolic evaluation is a data-dependent method that analyzes a path while it is ac-

tually being executed for specific input data. Rather than analyze a routine on a path-by-path basis, global symbolic evaluation attempts to create a closed-form expression that represents all paths.

2.2 Path-Dependent Symbolic Evaluation

Path-dependent symbolic evaluation analyzes distinct paths. In general, path-dependent symbolic evaluation is attempted on only a subset of the paths in a routine since a routine containing a loop may have an effectively infinite number of paths. The description of path-dependent symbolic evaluation that follows is independent of the method of path selection; it is assumed that path selection information is provided externally. This section provides an overview of the way path-dependent symbolic evaluation systems develop the symbolic representation of a given path.

Several path-dependent symbolic evaluation systems have been described [4,10,36,39,42,48,53,65]. These systems employ either of two evaluation techniques: forward expansion or backward substitution. The forward expansion technique [4,10,42] begins at the start node and develops the symolic representations as each statement on a path is interpreted. The backward substitution technique [39,36] begins with the final node and works toward the start node. While both techniques produce equivalent results, backward substitution requires additional processing when further analysis, such as determining path condition consistency, is desired. Thus, forward expansion is the technique outlined below. The path-dependent symbolic evaluation of the feasible path (s,1,3,4,5,6,7,8,9,6,10,f) is described below, and Figure 3 shows the expressions that are generated.

Forward expansion begins at the start node, where the path condition is initialized to the value true and the path values 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 3, where variable names are written in upper case and symbolic names in lower case.

After initializing the path values 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, when an assignment statement, such as $y_j := y_k * y_i$, is interpreted, the algebraic expression $s(y_k) * s(y_i)$ is generated and provides the new symbolic value for y_j, updating the corresponding element in PV. For the assignment statement at node 5 in RECTANGLE, for

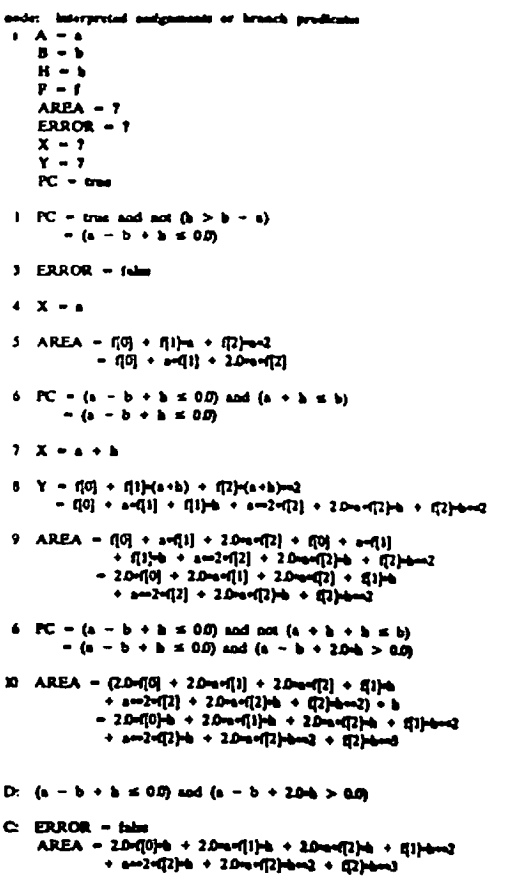


Figure 3. Path-dependent symbolic evaluation of path in RECTANGLE.

example, the current symbolic values of X and F after interpretation of statements (s,1,3,4) are substituted into the expression on the right-hand side, resulting in

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

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. When interpreting a branch predicate, such as $bp(i, j) = (y_k > y_l)$, the conditional expression $(s(y_k) > s(y_l))$ is generated and provides a symbolic value for the branch predicate $s(bp(i, j))$, which is conjoined to the evolving PC. When interpreting node 1 in RECTANGLE, 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

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

It is possible that the new PC is inconsistent, which implies that the path is nonexecutable. Methods for determining PC consistency are discussed in Section 3.1.

In RECTANGLE, the output parameters are ERROR and AREA, and thus the path computation is represented by

(s(ERROR), s(AREA)).

For path = (s,1,3,4,5,6,7,8,9,10,6,10,f) in RECTANGLE, the path domain is represented by

s(bp(1,3)) and s(bp(6,7)) and s(bp(6,10)).

The path domain and path computation resulting from path-dependent symbolic evaluation of path (s,1,3,4,5,6,7,8,9,6,10,f) are shown in Figure 3.

The paths to be evaluated by path-dependent symbolic evaluation can be either chosen by the use or selected automatically by a component of the system. Most path-dependent symbolic evaluation systems support an interactive path selection facility that allows the user to "walk through" a program, statement by statement. Such capabilities have been described for DISSECT [36] and ATTEST [10,71]. This feature is useful for debugging since the evolution of the PC and PV can be observed. More extensive program coverage can be expedited by an automated path selection facility for choosing a set of paths based on some coverage criterion. Several coverage criteria are discussed in Section 4.2.

2.3 Dynamic Symbolic Evaluation

Dynamic symbolic evaluation is one of the features often provided by dynamic testing systems [1,24,63]. Using test data to determine the path, the dynamic symbolic evaluation method monitors the execution of the path and provides symbolic representations of the results created by executing the path.

The dynamic symbolic evaluation component of dynamic testing systems provides a symbolic representation of the computation of each executed path. In addition to the user-supplied test data, symbolic names are associated with the input values. Throughout the execution, dynamic symbolic evaluation maintains the symbolic values of all variables as well as their usual computed values. As with path-dependent symbolic evaluation, the symbolic values are represented as algebraic expressions in terms of the symbolic names. Since dynamic testing systems monitor the normal execution process, the forward expansion technique described for path-dependent symbolic evaluation is a natural approach for creating these symbolic values.

After executing path P_j, the symbolic value for each output parameter is shown, providing the path computation. With dynamic symbolic evaluation, these expressions are generally displayed as trees instead of as algebraic expressions, although both or either form could be displayed. The computation trees that would be created for the specified input values to RECTANGLE are shown in Figure 4. Note that these input values cause path (s,1,3,4,5,6,7,8,9,6,10,f) to be executed.

Most dynamic symbolic evaluation systems are only concerned with providing the path computation. Since the input values are known, each interpreted branch predicate evaluates to the constant value true (or a run-time error is encountered). The PC is, therefore, equal to true and thus it is not necessary to check for PC consistency. Since the PC is often useful in validating the path, dynamic symbolic evaluation systems may also provide the symbolic representation of the path domain.

2.4 Global Symbolic Evaluation

The goal of global symbolic evaluation [8,52] is the derivation of a global representation of a routine—a symbolic representation of the domain and computation

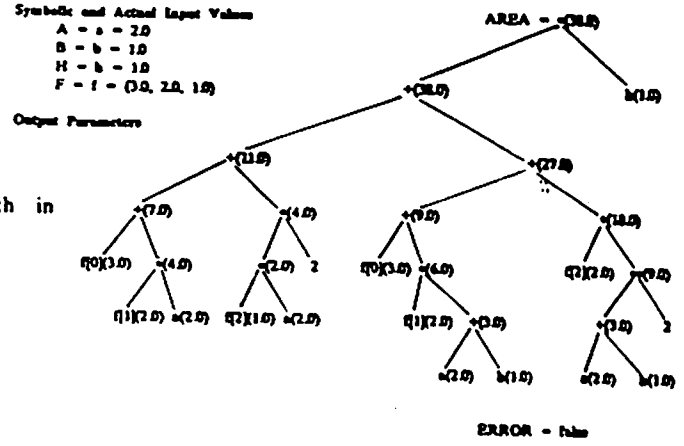


Figure 4. Dynamic symbolic evaluation of path in RECTANGLE.

for all paths, rather than along one specific path. Since there may be an effectively infinite number of executable paths in a routine, using path-dependent symbolic evaluation is unreasonable. Instead, global symbolic evaluation attempts to replace each loop with a closed form expression that captures the effect of that loop [8,12]. Using this technique, a path may then represent a class of paths in which each member differs from the others only by its number of loop iterations.

Global symbolic evaluation, like path-dependent symbolic evaluation, uses the control flow graph of a routine to guide evaluation. Loops are evaluated first by a loop analysis technique. For each loop, this technique attempts to create a loop expression, which is a closed form representation encompassing the effects of the loop. An analyzed loop can be replaced by the resulting loop expression, which can thereafter be evaluated as a single node. Thus, inner loops must be analyzed before outer loops. After all loops have been analyzed, the control flow graph has been reduced to a directed acyclic graph. In this section, an efficient interpretive technique for acyclic programs is described and then loop analysis, which also uses this interpretive technique, is explained.

For acyclic programs, or programs that have been made acyclic by using loop analysis, a more efficient interpretive technique than the forward expansion technique described above can be used. This technique interprets each node only once but in the context of all its predecessors and then saves this interpreted representation to be used when interpreting any of its successor nodes. To do this, a node cannot be interpreted until all its predecessors have been interpreted. Thus, global symbolic evaluation starts by interpreting the start node, then all nodes that have only the start node as a predecessor, and so on. For a node in the control flow graph, a case expression[1] is maintained, where each subcase represents one partial path reaching that node. Each subcase is composed of the PC for a partial path, as well as the symbolic values of all the variables computed along that partial path.

To see how a node is interpreted, consider a particular node m, with predecessor nodes i, . . . , j, which have been previously interpreted. Control may reach m via any of the edges (i, m), . . . , (j, m), and the transfer from a predecessor node occurs under the conditions of the corresponding branch predicate. Thus, when m is interpreted, each subcase of the case expression of each predecessor node must be considered independently. For predecessor node i, for instance, the branch predicate bp(i, m) is evaluated in the context of each subcase for node i, and for a particular subcase, bp(i, m) is interpreted in terms of the symbolic values of the variables for this subcase. This interpreted branch predicate is then conjoined to the PC for the partial path associated with this subcase of predecessor node i. As with path-dependent symbolic evaluation, it is desirable to check the consistency of the PC. If the PC is found to be inconsistent, this subcase is discarded, otherwise, the statement at node m must be interpreted in the context of this subcase for node i. After all the subcases for node i have been considered, this same procedure is followed for all other predecessor nodes of m. Finally, the subcase expressions derived from evaluating all the subcases of the predecessor nodes are combined and the resulting case expression represents all executable partial paths reaching node m. To illustrate this technique, Figure 5 shows a fragment of a control flow graph, gives a hypothetical case expression for node 11 in the graph, and shows the resulting case expressions for nodes 13, 14, and 15.

In global symbolic evaluation, a global representation of all paths is only possible when the loop analysis technique can create closed-form representations for all loops in the program. This 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 both the final iteration count and the symbolic values of the variables at entry to the loop. Figure 6 shows the results from loop analysis; these results as well as the loop analysis technique are explained in the remainder of this section.

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[2] contains only one backward branch. If we temporarily ignore this one branch, the loop body can be represented as an acyclic directed graph to which the interpretation technique described above can be applied. To initiate this interpretation, 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,

[1] In the case expression used by global symbolic evaluation, a subcase consists of an arbitrary boolean expression followed by the symbolic values assigned to the variables.

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

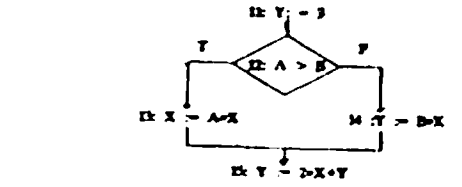


Figure 5. Hypothetical interpretation with global symbolic evaluation.

assuming it is for the kth iteration, is identical to the process described above, except that the symbolic name initially assigned to each variable is its value after execution of iteration $k - 1$—that is, the assumed value for y is y_{k-1} if y is changed in the loop and y_0 otherwise. The result of this interpretation is a set of recurrence relations that are 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 6 shows the results of this evaluation for the WHILE loop in RECTANGLE. (Since the loop in RECTANGLE only contains straight line code, each node only has one predecessor and so no case expression need be formed.)

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 6 provides the solutions to the recurrence relations for the loop in RECTANGLE. Although not illustrated in this example, two subcases must sometimes 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

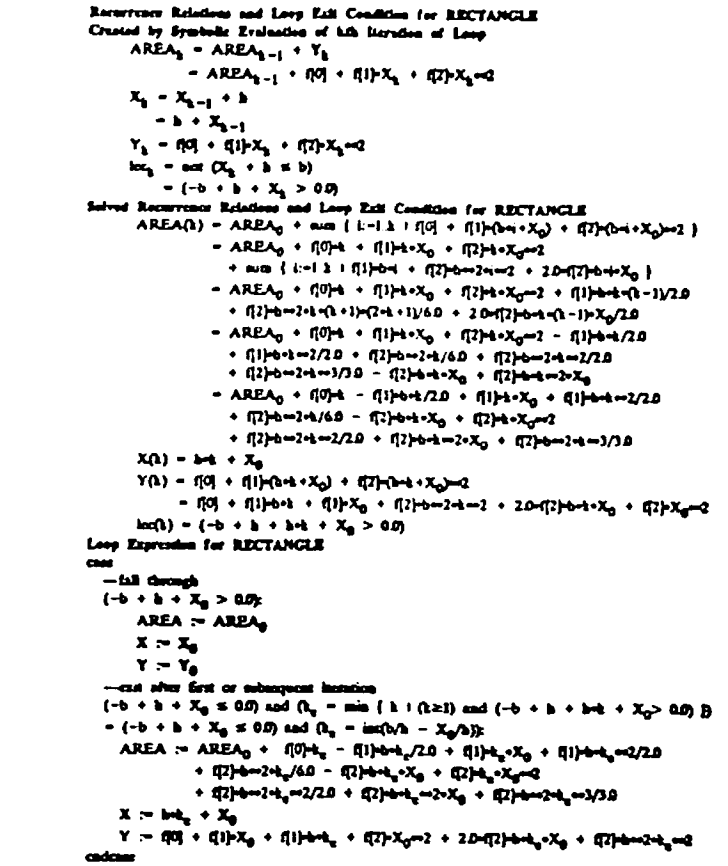


Figure 6. Loop analysis of RECTANGLE.

loop expression for the loop in RECTANGLE appears in the last part of Figure 6. 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 that 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, 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. Thus, for this subcase the condition is

$$\text{not}(\text{lec}(0)) \text{ and } (k_e = \min\{k \mid (k \geq 1) \text{ and } \text{lec}(k)\})$$

and the value for each variable y_i at exit from the loop is represented by $y_i(k_e)$. In this example, it is possible to precisely represent k_e by $\text{int}(b/h - X_0/h)$. Since the loop expression is a closed-form representation capturing the effects of the loop, the nodes in the loop can be replaced by a single node, annotated by this loop expression. If the loop body contains nodes i through j, this single node is denoted (i — j).

When a loop is encountered during global symbolic evaluation, each subcase in the loop expression must be considered in the context of each subcase of each predecessor node. Consider the interpretation of one subcase of the loop expression in the context of one subcase of a predecessor node. The results of this interpretation will be a single subcase for the interpreted loop node. The symbolic values of the variables of the predecessor subcase provide the values of the variables at entry to the loop. Thus, for variable y, the symbolic value of y in the subcase of the predecessor node is the value to be substituted for y_0. The PC of the loop node subcase is developed by interpreting the condition from the loop expression subcase and conjoining it with the PC of the predecessor subcase. The symbolic values of the variables of the loop node subcase are developed by interpreting the assignments specified by the loop expression subcase.

The above process is repeated for each subcase in the

loop expression with each subcase of each predecessor node. The resulting subcases are then combined to form the case expression for the interpreted loop node. Global symbolic evaluation can proceed as usual from this point. Figure 7 demonstrates the global symbolic evaluation of RECTANGLE. Here, only the start node, the final node, the nodes corresponding to conditional statements, the node preceeding the loop, and the loop node are shown. The symbolic values of variables that cannot be modified are shown only at the start node. Note that node 5 is the only predecessor node to the loop and node (6–9) provides the case expression resulting from interpretation of the loop expression. The final output of global symbolic evaluation of RECTANGLE also appears in Figure 7, where path P_1 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 if this condition is represented 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 [66] may be easier and more efficient than invoking general axiomatic and algebraic mechanisms to solve recurrence relations.

3. IMPLEMENTATION CONSIDERATIONS

The above section described the general methods associated with symbolic evaluation. When implementing a symbolic evaluation system there are many additional issues to be considered. This section discusses several of these issues, some of which are well understood and others that remain areas of current research.

3.1. Further Analysis of the Symbolic Representations

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 often desirable to simplify the symbolic representations, determine the consistency of the PC, and find alternative solutions for the PC that serve as test data.

Simplification can be done by converting the symbolic expressions into canonical forms. There are several available algebraic manipulation systems [3,6,56] 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, for path-dependent symbolic evaluation or global symbolic evaluation it is desirable to determine whether or not the PC is consistent. Not only is it desirable to recognize nonexecutable paths but also to recognize the inconsistency as soon as possible. Early detection of a nonexecutable path prevents worthless, yet costly, symbolic evaluation. A nonexecutable path can be detected as soon as possible by developing the PC as the statements on a path are interpreted and examining the evolving PC for consistency as each branch predicate is interpreted. For partial path $P_{j_k} = (s, J_{j_1}, \ldots, J_{j_k})$, the path condition is denoted $PC[P_{j_k}]$. When a node $J_{j_{k+1}}$ is considered as an extension to the partial path P_{j_k}, the interpreted branch predicate $s(bp(J_{j_k}, J_{j_{k+1}}))$ is first simplified and then examined for consistency with $PC[P_{j_k}]$. Unless inconsistency is determined, the interpreted branch predicate is conjoined to $PC[P_{j_k}]$, creating

$$PC[P_{j_{k+1}}] = PC[P_{j_k}] \text{ and } s(bp(J_{j_k}, J_{j_{k+1}})).$$

Thus at any point in this interpretation, there is a symbolic representation of the domain for the partial path that has been evaluated so far.

When used with the path-dependent symbolic evaluation, the incremental development of the PC allows an alternative edge to be selected on a partial path when an inconsistent branch predicate is initially encountered. Thus, the evaluation of the partial path up

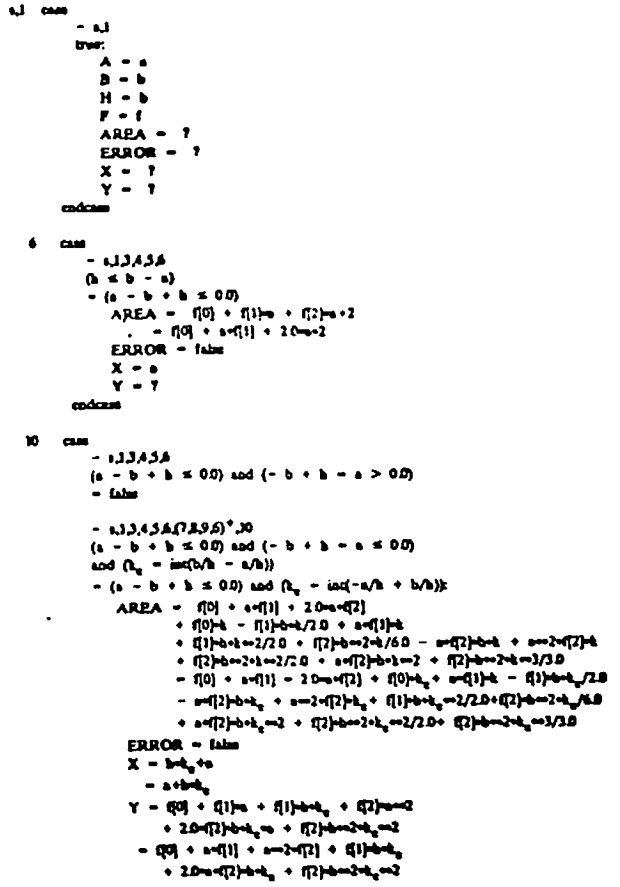


Figure 7. Path domains and computations for RECTANGLE.

to an inconsistent branch predicate can usually be salvaged. For example, the nonexecutable partial path (s,1,3,4,5,6,10) in RECTANGLE can be terminated as soon as the inconsistent PC is discovered. The symbolic value of the branch predicate for the edge (6,10), where the inconsistency occurred, is replaced by the symbolic value of the branch predicate for the alternative edge (6,8), and analysis continues.

Consistency or inconsistency may possibly be determined by performing simple reductions [21,22] on the newly interpreted branch predicate $s(bp(J_i,I_{i+1}))$ in the context of the existing consistent PC. On the one hand, it may be possible to determine that $s(bp(J_i, J_{i+1}))$ is dominated by relational expressions in $PC[P_k]$, in which case $PC[P_{k,i}]$ must not be inconsistent, since $PC[P_k]$ is not inconsistent. On the other hand, $s(bp(J_i, J_{i+1}))$ may be contradicted by a relational expression in $PC[P_k]$, in which case $PC[P_{k,i}]$ is inconsistent. In the evaluation of path P_2 in RECTANGLE, for example, $s(bp(6,10)) = (a - b + h > 0.0)$ is contradicted by $s(bp(1,3)) = (a - b + h \leq 0.0)$, thus PC[s,1,3,4,5,6,10] is inconsistent. While such reductions are sometimes applicable, it is often necessary to rely on more costly techniques, such as an automatic theorem prover [5] or one of a number of algebraic techniques. The ATTEST system [10,11], for example, uses a linear programming algorithm [43]. The advantage of choosing an algebraic 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. The next section discusses more sophisticated strategies for selecting test data for the PC that are aimed at detecting errors on the path. Both automatic theorem provers 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 [19]. In some instances, PC consistency or inconsistency cannot be determined; the symbolic representations for such a path can be provided, but whether or not the path can be executed is unknown.

3.2 Arrays

Array element determination causes a problem whenever the subscript of an array depends on input values, in which case, the element that is being referenced or defined in the array is unknown. The flow charts in Figure 8 illustrate this problem. The first part of Figure 8 shows indeterminate array subscripts. Note that at nodes 5 and 6 there is a constraint on the range of values for the subscript due to the PC. In the second part of Figure 8, the subscript values are constant and thus cause no problems. Although an indeterminate array

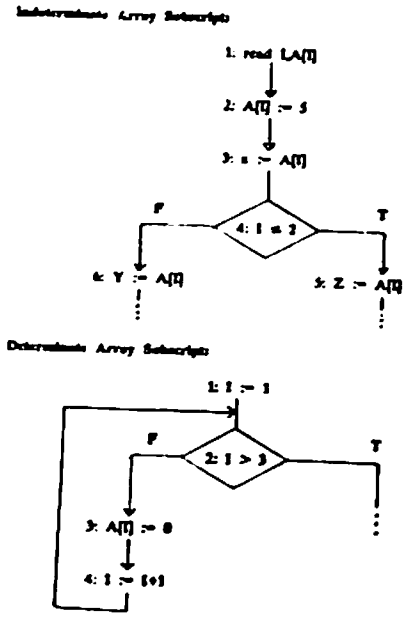


Figure 8. Array element determination.

element can be represented symbolically, determining PC consistency may become extremely complicated when such an occurrence affects the PC. This problem occurs frequently during both path-dependent symbolic evaluation and global symbolic evaluation. It cannot occur during dynamic symbolic evaluation since all values, including subscript values, are known.

Inefficient solutions for determining an appropriate array element exist, for in the worst case all possible subscript values can be enumerated. Though there has been some work on this problem [4,10,53], the results are still unsatisfactory. Efficient solutions requiring a minimal amount of backtracking are still being explored.

3.3 Routine Invocation

Several approaches to routine invocation during symbolic evaluation have been proposed. The simplest approach, which is not applicable for dynamic symbolic evaluation, is to represent the results of a routine invocation symbolically. For a procedure, such an approach might assign unique symbolic names for the output parameters each time the procedure is called. For a function (with no side effects), this approach might represent each invocation by the function name along with the arguments' symbolic values at the point of invocation. The advantage of this approach is that the calling routine can be evaluated even when the called routine

is not available. Thus this approach supports unit testing.

Another straightforward approach to routine invocation is to symbolically evaluate a path (or paths in the case of global symbolic evaluation) through the called routine by passing information to and from the called routine via the parameters. This approach is similar to normal execution. When a routine invocation is encountered, the symbolic values of the arguments are passed to the called routine. Any branch predicates that are interpreted within the called routine are conjoined to the PC in the usual manner. The symbolic values of the parameters are updated by the interpretation of assignment statements on the path in the called routine. When control returns to the calling routine, the symbolic value of each parameter is returned and assigned to the corresponding argument. This is the only approach to routine invocation that is applicable for dynamic symbolic evaluation.

The drawback of the first approach is that the precise effect of the invocation is unknown and this loss of information may degrade the results of any subsequent analysis. The drawback of the second approach is the inefficiency of interpretating a routine each time that routine is invoked. A third approach, called *subroutine substitution*, may avoid these drawbacks by utilizing the previously created symbolic representations of a routine. With path-dependent symbolic evaluation, the PC and PV of a path in a routine are saved for substitution. Later, when the routine is invoked, the symbolic values of the arguments are substituted for the symbolic names that were assigned to the parameters in the saved PC and PV of the called routine. The updated PC of the called routine is then conjoined to the existing PC of the calling routine. If this conjunction is consistent, then the corresponding path through the called routine could be executed, and this conjunction is the new PC. In addition, the symbolic values of the output parameters, which are represented in the PV of the called routine, are returned to the calling routine. With global symbolic evaluation, the global representation of the called routine is substituted into the global representation of the calling routine. Each subcase of the called routine must be evaluated in the context of each subcase of the calling routine at the point of invocation. For each such combination, this evaluation is similar to subroutine substitution during path-dependent symbolic evaluation.

Using subroutine substitution involves expensive reformulation and simplification of the symbolic representations. Unfortunately, it may not always be more efficient than reevaluation of the path(s) [72]. When arguments are functions or large arrays, these problems are further aggravated. Moreover, for path-dependent symbolic evaluation several evaluations of the called routine must be saved to make this a viable approach. For either path-dependent symbolic evaluation or global symbolic evaluation this approach assumes a bottom-up testing environment, where called routines must be tested before the calling routine.

A variation of subroutine substitution allows the specification of a called routine to be supplied in place of the source code of that routine. Such a specification would describe the function of the routine by providing the intended path domains and their associated path computations. There are a number of specification techniques that could be used, as described in Section 4.6. The evaluated specification could then be substituted as described for subroutine substitution. Such an approach has some of the drawbacks of subroutine substitution but allows for top-down testing and incremental development of software.

3.4 Input/Output

So far in this paper, we have only described symbolic evaluation for routines whose input and output are done only via parameters. Only minor modifications are necessary to handle input and output at arbitrary points in a routine. To handle input along a path, symbolic names representing the input values are assigned to the input variables whenever an input statement is encountered. The convention previously described for representing input values must be modified slightly, however, since input may occur more than once for a variable. One approach that maintains the association between input values and variables is to suffix each symbolic name with an index notation when necessary. For example, if a variable, say AMOUNT, is assigned input twice along a path, the first input value might be represented by amount.1 and the second by amount.2. To handle output along a path, the symbolic values of the output variables are provided whenever an output statement is encountered. With these extensions, the variables assigned input values, the variables whose values are output, as well as the number of inputs and outputs, may vary from path to path because different input and output statements may be encountered on different paths. Moreover, for global symbolic evaluation, the number of inputs and outputs may depend on the final loop iteration counts for the routine. Although input and output along a path requires no substantial changes to the interpretive techniques originally described, the functional conceptualization of a routine must allow for an arbitrary, and perhaps varying, number of inputs and outputs.

4. APPLICATIONS

Most notably symbolic evaluation has been the foundation for much of the research in program testing. Much of this work is concerned with the problem of selecting the paths that should be tested and the problem of selecting test data for those paths. The symbolic representations of the path domains and path computations have proven useful in both these aspects. In addition to testing, symbolic evaluation methods have been readily applied to other research areas of software engineering, including verification and certification, debugging, optimization, and early software development. This section discusses the application of symbolic evaluation for each of these areas.

4.1 Verification and Certification

Formal verification methods use symbolic evaluation techniques to assist in forming the verification conditions. Typically, input, output, and loop invariant assertions are 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 semantics of the programming language and any required application-dependent axioms. This process [25,32,33,47] and a number of related approaches to verification, have been frequently described in the literature and will not be discussed here. Instead, this section disucsses some less comprehensive verification techniques that are used to detect or certify the absence of particular program properties.

The symbolic representations that are generated by symbolic evaluation 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 *symbolic testing* [38]. 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 computations and the required number of significant digits.

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, for example, 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. The error conditions that can be checked by symbolic evaluation are language dependent. In FORTRAN, for example, error conditions can be created for division by zero, invalid DO loop parameters, invalid variable dimensions, and out-of-bound subscript values, among others.

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 dynamic symbolic evaluation and path-dependent symbolic evaluation provides conclusions about the occurrence of that error on a specific path. When similar capabilities are provided by global symbolic evaluation certification is done for all (classes of) paths and 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. Meth-

ods for doing this total verification are discussed further in Section 4.6.

4.2 Test Path Selection

It is usually impractical to test every path in a routine and thus it is imperative to have a method for selecting a meaningful subset of paths to be exercised. Support of the path selection process is a natural application of symbolic evaluation. Several criteria for path selection that utilize symbolic evaluation techniques are outlined below.

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. 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, however, 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 [28]. Symbolic evaluation is a useful technique, however, for aiding in the selection of executable paths. The ATTEST system, 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 [72].

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 coverage and path coverage. Some alternative criteria simply limit loop iterations. The EFFIGY system [42] 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.

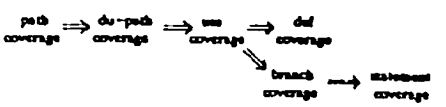
Howden has proposed the *boundary-interior* method for classifying paths [34]. With this method, two paths that differ other than in loop iterations are in different classes. In addition, two paths that differ only in the way they traverse loops are in different classes if

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 that 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 data flow information. *Data flow techniques* [44,45,50,54], require the selection of subpath(s) based on particular sequences of definitions and references to the variables in the program. Rapps and Weyuker [54] have described a partial ordering on a family of data flow techniques for path selection. Figure 9 shows part of 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 10. Def coverage requires the selection of subpaths 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 exe-

Figure 9. Data flow testing criteria.



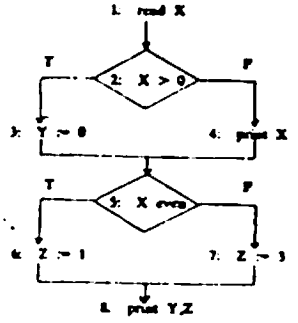


Figure 10. Data flow testing example.

cuted. 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), (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 each 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 or data flow information, path selection techniques have been developed that relate directly to the elimination of potential errors in program statements. *Perturbation testing* [31,73] 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 these possible perturbations. As an example, consider the flow chart in Figure 11. Along path (. . . ,7,9, . . .) the value of Z is the same as the value of 2•X at node 9. Any error in the predicate at node 9 that can be represented by k • (Z − 2•X), where k is a constant, could not be detected along path (. . . ,7,9, . . .). For instance, if the branch predicate at node 9 should have been Z − X > Y, the error would not be detected. Along path (. . . ,8,9, . . .), however, this equality does not hold and thus the error could be detected. In general, another

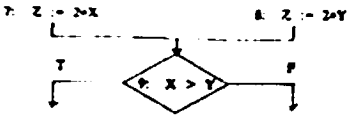


Figure 11. Perturbation testing example.

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 [7], 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 statment can be represented by using modified symbolic evaluation techniques. Perturbation testing is currently being implemented as an extension to the ATTEST symbolic evaluation system.

4.3 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.

The symbolic representation of a path can be used as the basis on which to select test data for that path. The most straightforward technique simply examines the PC to determine a solution—that is, one arbitrary test datum to execute the path. As noted previously, SELECT [4] and ATTEST are two path-dependent symbolic evaluation systems that generate such test data by using an algebraic technique for determining PC consistency.

More rigorous techniques have been proposed that attempt to capture the ideas underlying several error-sensitive heuristics [49,26,69,55]. The error-sensitive techniques attempt to characterize 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 a 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 evaluation 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 A = 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. One 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 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 based on symbolic evaluation.

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 that 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 [14]. 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.

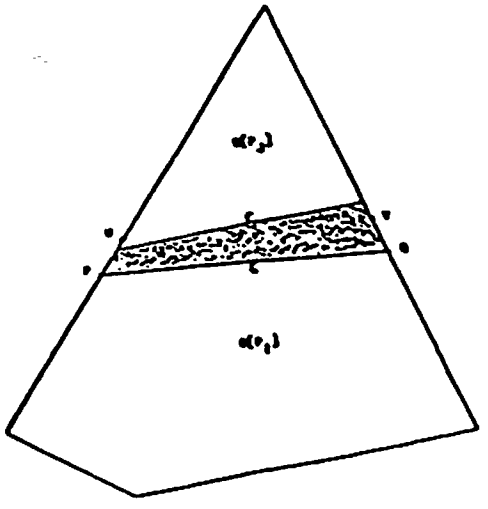
Theoretical results have shown that more rigorous computation testing techniques can guarantee the absence of certain types of computation errors when the path computations fall into well-behaved functional classes. For example, there are a few techniques that can be applied if the symbolic value for 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 [61]. Alternately, for a univariate polynomial of degree N, N + 1 test points are sufficient [37]. Probabilistic arguments have been made for reducing this number without sacrificing must confidence [20]. Similar results have been provided for multivariate polynomials.

In a similar way, when the path computations fall into other specialized categories, the computation testing guidelines can be tuned to guide in the selection of a more comprehensive set of test data. For example, if a path computation involves logic functions [27] or 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 [13,70] 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 12 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 [13]. Figure 13 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

Figure 12. Domain testing strategy.



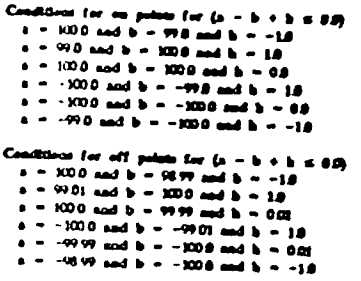


Figure 13. Conditions for satisfying domain testing strategy for RECTANGLE

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.

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 discussed in Section 4.6.

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.

4.4 Debugging

It is not surprising that symbolic evaluation, which is useful for guiding testing, can also be used to help discover the cause of an error—that is, for program debugging. When an error is revealed on a path that has been symbolically evaluated, then the symbolic representations of the path computation and path domain can be examined to obtain information about the cause of a known error. Once an error is known to exist, the

programmer knows at least one algebraic expression that is in error and can focus on that expression in search of clues to the actual cause of the error. In addition to the symbolic representations of the erroneous statements, symbolic evaluation systems could provide a list of the executed statements that affected the algebraic expression in error; only those statements need be examined to determine the cause of the error. This is similar to program slicing [67], a technique that provides a modified listing of the source program containing only the statements that could affect selected statements.

To assist in debugging, dynamic testing systems often provide a capability for examining the computation trees for the symbolic representations while they are being constructed statement-by-statement. Some of these systems allow the user to stop execution at any statement and "unexecute." In other words, the user can direct the system to undo part of the preceding execution. This "unexecution" would show the reverse evolution of the computation trees. Observing both the evolution and reverse evolution of the trees can help the user isolate an error. Experiments with the dynamic testing system ISMS [24] have shown that both of these features are beneficial for debugging. Similar capabilities are possible with interactive path-dependent symbolic evaluation systems.

Testing strategies that rely on symbolic evaluation information can also use that information to provide valuable assistance to the debugging process [15]. For example, if a test case that was selected with the goal of exposing a particular type of error indeed resulted in an error, then the goal and the information used to find a test case satisfying that goal would be useful during debugging. We suspect that, like the way optimization techniques have been modified to provide interesting data flow validation techniques [51], testing techniques can also be redirected to be useful debugging tools.

4.5 Program Optimization

Symbolic evaluation also has applications in program optimization [64]. The internal representation of the path values [16], can be used for common subexpression elimination and constant folding. In addition, several types of loop optimizations may sometimes be performed when the loop expressions are obtainable by global symbolic evaluation. Loop-invariant computations may be easily detected since they are independent of the iteration count of the loop; these may thus be moved outside of the loop. Loop fusion can sometimes be performed when the number of iterations performed by two loops can be determined to be the same and variables referenced in the second loop are not defined in a later iteration of the first loop. When variables modified within the loop have values that form arithmetic progressions—that is, they are incremented by the same amount each time through the loop—these computations can sometimes be moved out of the loop and replaced by expressions in terms of the final loop iteration count. Optimizations that perform in-line substitution of a routine may also benefit from global symbolic evaluation, since the closed form representation of the routine may enable better determination of when such substitution is useful.

The REDFUN system [2,23] uses symbolic evaluation to enhance the performance of LISP programs. In addition, Osterweil [51] describes a method in which data flow analysis and symbolic evaluation can be used jointly to optimize code, particularly the instrumented code created by dynamic testing systems.

4.6 Software Development

In this paper we have focused on the analysis of the code. Software validation, however, should be concerned with all stages of program development. As work progresses in the areas of requirements, specifications, and design, symbolic evaluation methods are proving to be useful for these earlier stages of software development as well [9,58].

Symbolic evaluation of a formal specification has been proposed as an alternative to early prototype development. Symbolic evaluation systems have has been built for the GIST [17] and INAJO [40] specification languages. In these systems, 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 the concrete execution of a prototype, which for each test case only tests a single path for a unique set of inputs.

Using a specification of the intended function of a program to help with testing, as opposed to only considering the code itself, has often been suggested [29,57,68]. The partition analysis method [58] attempts to accomplish this by applying global symbolic evaluation to a routine, as well as to its specification, thus creating global representations of both. By comparing these two representations, a partition of the domain is determined. This partition is then utilized in verifying the routine's consistency with the specification. Information derived from this verification process along with error-sensitive testing strategies are applied to guide in the selection of test data. Test data selection is thus based on the specification and not only the im-

plementation, as such partition analysis is one of the few testing techniques to address missing path errors. A preliminary study of this method showed it to be quite effective at discovering errors [60]. Partition analysis has been applied to several different kinds of specification languages including predicate calculus [30], state transitions, and both high-level and low-level procedural languages [59]. 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 and at various stages in the lifecycle; it can be applied to compare software specifications to designs, high-level to low-level designs, VLSI specs to VLSI designs, and so on.

Symbolic evaluation has also been used to support program construction. Tinker [46], Pygmalion [62], and Curry [18] are experimental systems that allow the user to express program requirements in terms of symbolic representations and then an attempt is made to construct the program automatically based on the examples provided. Similarly, Waters [66] uses symbolic evaluation, in conjunction with a library of common program patterns, to synthesize programs.

5. SUMMARY

Symbolic evaluation is of interest because it is the foundation for a number of software engineering techniques. This paper describes three methods of symbolic evaluation. Although the symbolic representations provided by each of the three methods are similar, they differ enough to substantially affect the cost as well as the types of subsequent program analysis that can be performed. If dynamic symbolic evaluation maintains only the information required to develop the final symbolic representations, its applications are usually restricted to program debugging. Path-dependent symbolic evaluation maintains more general information about a path and thus has a more extensive range of applications, including test path selection, test data generation, and cerfitication. Global symbolic evaluation analyzes all paths and maintains a global representation of a routine and thereby has applications to program optimization and verification in addition to the applications of path-dependent symbolic evaluation. While all three methods of symbolic evaluation have been implemented in experimental systems, efficient, more useful implementations pose several problems.

Initially, symbolic evaluation was employed by formal verification techniques to formulate the verification conditions that must be proven. As discussed in this paper, 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 integrates 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, should 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 proposed which use this approach. While the initial work in this area is quite promising, better, as well as more integrated, techniques must be developed.

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

REFERENCES

1. R. M. Balzer, EXDAMS—Extendable Debugging and Monitoring System, *1969 Spring Joint Computer Conference, AFIPS Conference Proceedings*, 34, AFIPS Press, Montvale, NJ, pp. 576–580.
2. L. Beckman, A. Haraldson, O. Oskarsson, and E. Sandewall, A Partial Evaluator and Its Use As A Programming Tool, *Artificial Intelligence 7* (1976).
3. R. Bogen, MACSYMA Reference Manual, The Mathlab Group, Project MAC, Massachusetts Institute of Technology, 1975.
4. 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* 234–244 (April 1975).
5. R. S. Boyer and J. S. Moore, *A Computational Logic*, Academic Press, New York, 1979.
6. W. S. Brown, *Altran User's Manual*, 1, Bell Telephone Laboratories, 1973.
7. T. A. Budd, The Portable Mutation Testing Suite, Department of Computer Science, University of Arizona, Technical Report 83-8, March 1983.
8. T. E. Cheatham, G. H. Holloway, and J. A. Townley, Symbolic Evaluation and the Analysis of Programs, *IEEE Trans. Software Engineering* SE-5, 402–417 (July 1979.
9. T. E. Cheatham, J. A. Townley, and G. H. Holloway, A System for Program Refinement, *Proceedings of the 4th*

International Conference of Software Engineering, September 1979, pp. 53–62.
10. L. A. Clarke, A System to Generate Test Data and Symbolically Execute Programs, *IEEE Trans. Software Engineering* SE-2, 215–222 (Sep 1976).
11. L. A. Clarke, Automatic Test Data Selection Techniques, *Infotech State of the Art Report on Software Testing* 2, 43–64 (Sep 1978).
12. L. A. Clarke and D. J. Richardson, Symbolic Evaluation Methods—Implementations and Applications, B. Chandrasekaran and S. Radicchi, (eds)., in: *Computer Program Testing*, North-Holland, New York, 1981, pp. 65–102.
13. L. A. Clarke, J. Hassell, and D. J. Richardson, A Close Look at Domain Testing, *IEEE Trans. Software Engineering* SE-8, 380–390 (July 1982)
14. L. A. Clarke and D. J. Richardson, A Rigorous Approach to Error-Sensitive Testing, *Sixteenth Hawaii International Conference on System Sciences*, January 1983.
15. L. A. Clarke and D. J. Richardson, The Application of Error-Sensitive Testing Strategies to Debugging, *ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging*, March 1983.
16. J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, New York University, Courant Institute of Mathematical Science, April 1970.
17. D. Cohen, W. Swartout, and R. Balzer, Using Symbolic Execution To Characterize Behavior, ACM SIGSOFT Rapid Prototyping Workshop, *Software Engineering Notes* 7, 25–32 (Dec 1982).
18. G. Curry, Programming by Abstract Demonstration, Ph.D. thesis, University of Washington at Seattle, 1978.
19. M. Davis, Hilbert's Tenth Problem is Unsolvable, *American Math. Mon.* 80, 233–269 (March 1973).
20. R. A. DeMillo and R. J. Lipton, A Probabilistic Remark on Algebraic Program Testing, *Information Processing Lett.* 7 (June 1978).
21. L. P. Deutsch, An Interactive Program Verifier, Ph.D. dissertation, University of California, Berkeley, May 1973.
22. L. K. Dillon, Constraint Management in the ATTEST System, University of Massachusetts, Department of Computer and Information Science, Technical Report 81-9, May 1981.
23. P. Emanuelson, Performance Enhancement in a Well-Structured Pattern Matcher Through Partial Evaluation, *Linkoping Studies in Science and Technology Dissertations*, No. 55, Linkoping University, Sweden, 1980.
24. R. E. Fairley, An Experimental Program-Testing Facility, *IEEE Trans. on Software Engineering* SE-1, 350–357 (Dec 1975).
25. R. W. Floyd, Assigning Meaning to Programs, *Proceedings of a Symposium in Applied Mathematics* 19, American Mathematical Society, 1967, pp 19–32; *Commun. ACM*, 14, 39–45 (Jan 1971).
26. K. A. Foster, Error Sensitive Test Case Analysis (ESTCA), *IEEE Trans. Software Engineering* SE-6, 258–264 (May 1980).
27. K. A. Foster, Sensitive Test Data for Logical Expressions, *ACM SIGSOFT Software Engineering Notes* 9 (July 1984).
28. H. N. Gabow, S. N. Maheswari, and L. J. Osterweil, On Two Problems in the Generation of Program Test Paths, *IEEE Trans. Software Engineering* SE-2, 227–231 (Sep 1976).
29. J. B. Goodenough and S. L. Gerhart, Toward a Theory of Test Data Selection, *IEEE Trans. Software Engineering* SE-1, 156–173 (June 1975).
30. J. S. Gourlay, Theory of Testing Computer Programs, Ph.D. thesis, University of Michigan, 1981.
31. 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, CA, May 1982.
32. S. L. Hantler and J. C. King, An Introduction to Proving the Correctness of Programs, *Computing Surveys* 8, 331–353 (Sep 1976).
33. C. A. R. Hoare, Proof of a Program: FIND, *Commun. ACM* 14, 39–45 (Jan 1971).
34. W. E. Howden, Methodology for the Generation of Program Test Data, *IEEE Trans. Computer* C-24, 554–559 (May 1975).
35. W. E. Howden, Reliability of the Path Analysis Testing Strategy, *IEEE Trans. Software Engineering* SE-2, 208–215 (Sep 1976).
36. W. E. Howden, Symbolic Testing and the DISSECT Symbolic Evaluation System, *IEEE Trans. Software Engineering* SE-3, 266–278 (July 1977).
37. W. E. Howden, Algebraic Program Testing, *ACTA Informatica* 10 (1978).
38. W. E. Howden, An Evaluation of the Effectiveness of Symbolic Testing, *Software: Practice and Experience* 10, 381–397 (July–Aug 1978).
39. J. C. Huang, An Approach to Program Testing, *ACM Computing Surveys* 7, 113–128 (Sep 1975).
40. 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 in *IEEE Trans. Software Engineering*.
41. J. C. King, A Program Verifier, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, Sep 1969.
42. J. C. King, Symbolic Execution and Program Testing, *Commun. ACM* 19, 385–394 (July 1976).
43. A. H. Land and S. Powell, *FORTRAN Codes for Mathematical Programming*, John Wiley and Sons, New York, 1973.
44. J. W. Laski, A Hierarchical Approach to Program Testing, Department of Systems Design, University of Waterloo, Waterloo, Ontario, Canada, Technical Report No.55CFW130779.
45. J. W. Laski and B. Korel, A Data Flow Oriented Program Testing Strategy, *IEEE Transactions of Software Engineering* SE-9, 347–354 (May 1983).
46. H. Lieberman and C. Hewitt, A Session with Tinker: Interleaving Program Testing with Program Design, *1980 Lisp Conference*, Stanford University, 1980.

47. R. L. London, A View of Program Verification, *Proceedings International Conference on Reliable Software* 534-545 (April 1975).
48. E. F. Miller and R. A. Melton, Automated Generation of Test Cast Data-Sets, *Proceedings of the International Conference on Reliable Software*, 51-58 (April 1975).
49. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1979.
50. S. C. Ntafos, On Testing With Required Elements, *Proceedings of COMPSAC '81* 132-139 (Nov 1981).
51. L. J. Osterweil, Software Engineering, *Program Flow Analysis Theory and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1981.
52. E. Ploedereder, Pragmatic Techniques for Program Analysis and Verification, *Proceedings of the Fourth International Conference of Software Engineering* 63-72 (Sep 1979).
53. C. V. Ramamorthy, S. F. Ho, and W. T. Chen, On the Automated Generation of Program Test Data, *IEEE Trans. Software Engineering* SE-2, 293-300 (Dec 1976).
54. S. Rapps and E. J. Weyuker, Data Flow Analysis Techniques for Test Data Selection, *Sixth International Conference on Software Engineering* (Oct 1982).
55. S. T. Redwine, An Engineering Approach to Test Data Design, *IEEE Trans. Software Engineering* SE-9, 191-200 (March 1983).
56. 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.
57. D. J. Richardson, Theoretical Consideration in Testing Programs by Demonstrating Consistency with Specifications, *Digest of the Workshop on Software Testing and Test Documentation* 19-56 (Dec 1978).
58. D. J. Richardson and L. A. Clarke, A Partition Analysis Method to Increase Program Reliability, *Fifth International Conference on Software Engineering*, March 1981, pp. 244-253.
59. D. J. Richardson, Specifications for Partition Analysis, University of Massachusetts, Department of Computer and Information Science, Technical Report 81-34, Aug 1981.
60. 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* 529-538 (Nov 1982).
61. J. H. Rowland and P. J. Davis On the Use of Transcendentals for Program Testing, *J. Assoc. Computing Machinery* 28 181-190 (Jan 1981).
62. D. C. Smith, Pygmalion: A Creative Programming Environment, Stanford Ph D. thesis, 1975.
63. I. G. Stucki, Automatic Generation of Self-Metric Software, *Rec. 1973 Symposium on Software Reliability* 94-100 (April 1973).
64. J. A. Townley, The Harvard Program Manipulation System, Center for Research in Computing Technology, Harvard University, TR-23-76, 1976.
65. U. Voges, L. Gmeiner, and A. Amschler von Mayrhauser, SADAT—An Automated Testing Tool, *IEEE Trans. Software Engineering* SE-6, 286-290 (May 1980).
66. R. C. Waters, A Method for Analyzing Loop Programs, *IEEE Trans. Software Engineering* SE-5, 237-247 (May 1979).
67. M. Weiser, Program Slicing, *Fifth International Conference on Software Engineering*, 439-449 (March 1981).
68. E. J. Weyuker and T. J. Ostrand, Theories of Program Testing and the Application of Revealing Subdomains, *IEEE Trans. Software Engineering* SE-6, 236-246 (May 1980).
69. E. J. Weyuker, An Error-Based Testing Strategy, New York University, Computer Science Department, New York, Technical Report 027, January 1981.
70. L. J. White and E. I. Cohen, A Domain Strategy for Computer Program Testing, *IEEE Trans. Software Engineering* SE-6, 247-257 (May 1980).
71. D. Winters, N. Ogden, and L. A. Clarke, A Definition of AID: the ATTEST Interface Description Language, University of Massachusetts, Department of Computer and Information Science, Technical Report 78-15, Dec 1978.
72. J. L. Woods, Path Selection for Symbolic Execution Systems, Ph.D. dissertation, University of Massachusetts, May 1980.
73. S. J. Zeil, Testing for Perturbations of Program Statements, *IEEE Trans. Software Engineering* SE-9, 335-346 (May 1983).