Symbolic Evaluation Methods — Implementation and Applications

Lori A. Clarke
Debra J. Richardson

COINS Technical Report 81–42
1981

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Appeared in
Computer Program Testing
B. Chandrasekaran and S. Radicchi, editors,
North-Holland Publishing Company, 1981

This work was supported by the National Science Foundation under grant NSFMCS 77-02101 and the Air Force Office of Scientific Research under grant AFOSR 77-3287, and the International Business Machines Corporation under the graduate fellowship program.

Symbolic Evaluation Methods -- Implementations and Applications

Lori A. Clarke
Debra J. Richardson

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

This paper describes symbolic evaluation, a program analysis method that concisely represents a program's computations and input domain by symbolic expressions. The general concepts are explained and three related methods of symbolic evaluation are described.

The first method, symbolic execution, is a path analysis technique used mainly for program validation. There are several symbolic execution systems that have been developed and some of the major distinctions between these systems are described. The second method, dynamic symbolic evaluation, is a data dependent method that provides a representation of the executed program path. This method is used primarily as a debugging aid. The third method, global symbolic evaluation, attempts to provide a symbolic representation of the entire program. Although there are still many unresolved problems with this method, it has some direct applications to program verification and optimization.

Examples of all three methods are given. Each method's implementation approach, applications and limitations are described, as well as the status of current research in the area.

1. INTRODUCTION

The ever increasing demand for larger and more complex programs has created a need for automatic assistance in the software development process. Software engineering is concerned with establishing a more supportive environment to aid this process. Such an environment will probably encompass a wide variety of tools ranging from knowledgable text editors to formal program verifiers. Many of the tools that are being developed are directed toward the validation of software. Through program analysis, these tools detect errors, determine program consistency, and generally increase confidence in the program. Several of these validation tools employ a technique, called symbolic evaluation, that creates a symbolic representation of the program. Symbolic evaluation is a program analysis method that monitors the manipulations performed on the input data. Computations are represented as algebraic expressions over the input data, thus maintaining the relationship between the input data and the resulting values. Normal execution computes numeric values but loses information about the way in

·This work was supported by the National Science Foundation under grant NSFMCS 77-02101, the Air Force Office of Scientific Research under grant AFOSR 77-3287, and the International Business Machines Corporation under the graduate fellowship program.

which the numeric values were derived, whereas symbolic evaluation preserves this information. Symbolic evaluation has a wide range of validation applications, including testing, debugging, and formal program verification, as well as applications in program optimization and documentation.

There are three basic methods of symbolic evaluation: symbolic execution, dynamic symbolic evaluation, and global symbolic evaluation. Symbolic execution is a path-oriented evaluation method that describes data dependencies for a 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.

This paper introduces the basic concepts of symbolic evaluation and presents notations for uniformly representing and comparing the three methods. More work has been done in the area of symbolic execution, so a detailed description of symbolic execution is given first. The other symbolic evaluation methods are then described. Examples of the three methods are given to demonstrate their corresponding strengths and weaknesses, and several applications of each method are discussed.

```
          procedure TRANSACT (DAYS:in integer;
                              AMOUNT:in real;
                              BALANCE:in out real;
                              INTEREST:out real;
                              BELOWMIN:out boolean;
                              OVERDRAFT:out boolean) is
              NEWBAL:real; -- new balance
              RATE:constant real := 0.06; -- interest rate
              MINBAL:constant real := 100.00; -- minimum balance
              BMCHARGE:constant real := 0.10; -- below minimum charge
              ODCHARGE:constant real := 4.00; -- overdraft charge

      s   begin
      1       OVERDRAFT := false;
      2       BELOWMIN := false;
      3       NEWBAL := BALANCE * (1+RATE/365) ** DAYS;
      4       INTEREST := NEWBAL - BALANCE;
      5       if AMOUNT > 0.0 then -- process deposit
      6          NEWBAL := NEWBAL + AMOUNT;
              endif;
      7       if AMOUNT < 0.0 then -- process check
      8          if -AMOUNT > NEWBAL then
      9             OVERDRAFT := true;
     10             NEWBAL := NEWBAL - ODCHARGE;
                 else
     11             NEWBAL := NEWBAL + AMOUNT;
                 endif;
     12          if NEWBAL < MINBAL then
     13             BELOWMIN := true;
     14             NEWBAL := NEWBAL - BMCHARGE;
                 endif;
              endif;
     15       BALANCE := NEWBAL;
      f   end TRANSACT;
```

Figure 1. Routine Transact

2. BASIC CONCEPTS

This section presents some concepts fundamental to symbolic evaluation. Some terminology is introduced and the general evaluation technique that is used by the three methods is described. Initially, the description is 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 of symbolic evaluation and are not necessary for the actual analysis performed by the three methods. The modifications necessary to handle routine invocations and input and output statements will be addressed later. The concepts presented in this section will be illustrated for the routine TRANSACT, shown in Figure 1, which handles a transaction for an interest-bearing checking account.

Program analysis methods typically represent a routine R by a directed graph, called a control flow graph, that describes the possible flow of control through the routine. The nodes in the graph, $\{n_1, n_2, \ldots, n_q\}$, represent executable statements. Note that in Figure 1, the statements in TRANSACT are annotated with node numbers. Each edge is specified by an ordered pair of nodes, (n_i, n_j), which indicates that a transfer of control exists from n_i to n_j. Associated with each transfer of control are conditions under which such a transfer occurs. The branch predicate that governs traversal of the edge (n_i, n_j) is denoted by $bp(n_i, n_j)$. 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 n_i that precedes nodes n_j and n_k, the branch predicate for one edge (n_i, n_j) is the complement of the branch predicate for the other edge (n_i, n_k) — thus, $bp(n_i, n_j) = not(bp(n_i, n_k))$. Note that each IF statement, nested or otherwise, forms a pair of complementary branch predicates. In TRANSACT, for example, node 8 precedes nodes 9 and 11 and bp(8,9) = (NEWBAL + AMOUNT < 0.0) while bp(8,11) = (NEWBAL + AMOUNT >= 0.0). Some conditional statements, such as the FORTRAN computed GO TO or the Ada CASE statements, may have more than two successor nodes and each branch predicate must be represented appropriately. In this paper, the control flow graph has a single entry point, the start node n_s, and a single exit point, the final node n_f. If necessary a null node can be added to a graph for the start node, and likewise for the final node, to accomplish this single-entry, single-exit form without loss of generality. Figure 2 shows the control flow graph for TRANSACT.

A subpath in a control flow graph is a sequence of statements, $(n_{Hi}, n_{Hi+1}, \ldots, n_{Ht})$, where for all $i \leq j < t$, n_{Hj} is a node in the control flow graph such that there exists a transfer of control from n_{Hj} to n_{Hj+1}. A partial path is a subpath that begins with the start node and is denoted by P_{Hu}, where $P_{Hu} = (n_s, n_{H1}, n_{H2}, \ldots, n_{Hu})$. Hence, for any partial path P_{Hu} with $u \geq 1$, $P_{Hu} = (P_{Hu-1}, n_{Hu})$, where $P_{H0} = (n_s)$. A path is a partial path that ends with the final node and is denoted by P_H, thus $P_H = (n_s, n_{H1}, n_{H2}, \ldots, n_{Hv}, n_f)$. A routine R is composed of a set of paths $\{P_1, P_2, \ldots\}$; there may be an infinite number of paths due to program loops. The routine TRANSACT does not contain any loops, thus the paths can all be listed and are provided in Figure 3.

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. The control flow graph is a representation of all possible paths, both executable and nonexecutable, through the corresponding routine. The paths in TRANSACT that are nonexecutable are P_1, P_3, P_5, and P_7.

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

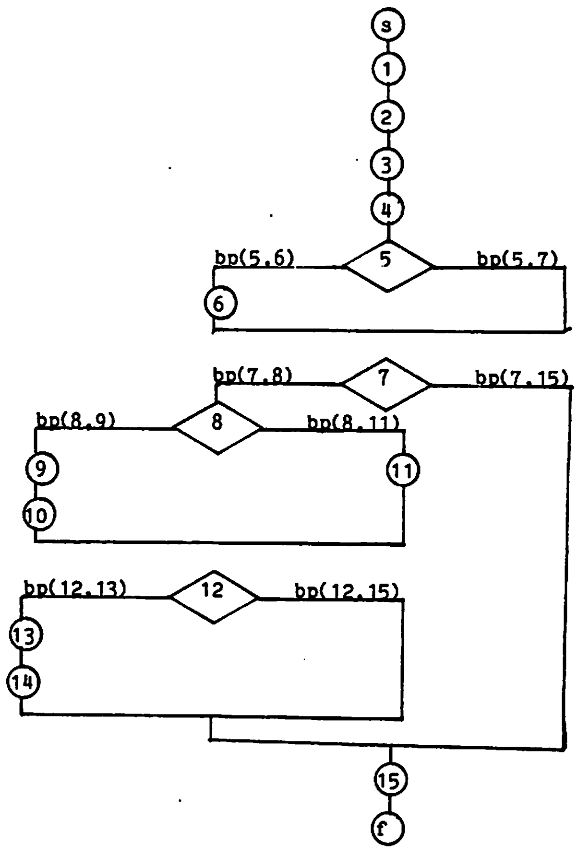

Figure 2. Control Flow Graph for TRANSACT

P_1 = (s,1,2,3,4,5,6,7,8,11,12,13,14,15,f)
P_2 = (s,1,2,3,4,5,7,8,11,12,13,14,15,f)
P_3 = (s,1,2,3,4,5,6,7,8,9,10,12,13,14,15,f)
P_4 = (s,1,2,3,4,5,7,8,9,10,12,13,14,15,f)
P_5 = (s,1,2,3,4,5,6,7,8,11,12,15,f)
P_6 = (s,1,2,3,4,5,7,8,11,12,15,f)
P_7 = (s,1,2,3,4,5,6,7,8,9,10,12,15,f)
P_8 = (s,1,2,3,4,5,7,8,9,10,12,15,f)
P_9 = (s,1,2,3,4,5,6,7,15,f)
P_{10} = (s,1,2,3,4,5,7,15,f)

Figure 3. Paths of TRANSACT

output values, are represented by a vector $y = (y_1, y_2, \ldots, y_W)$; note that a distinction is made between a variable and its value.[2]

The path domain corresponding to a path is the set of all x in X for which that path could be executed. The path domain of a nonexecutable path, therefore, is empty. Execution of a path performs a path computation 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 all functions of the executable paths in R.

Symbolic evaluation provides representations for the path domains and path computations of a routine. These symbolic representations describe the data dependencies for the paths that are analyzed. Symbolic evaluation methods use symbolic names to represent the input values. The statements on a path are interpreted and the symbolic values of all variables and branch predicates are maintained as expressions in terms of these symbolic names.

Before evaluating a path, the symbolic values of the variables are initialized at the start node. Input parameters are assigned symbolic names, variables that are initialized before execution are assigned the corresponding constant value, and all other variables are assigned the undefined value "?". Figure 4 shows the initial symbolic values assigned to the variables in TRANSACT. Note that the convention used in this paper is to refer to variable names in upper case and symbolic names in lower case, where an input parameter's name in lower case is assigned for the corresponding input value.

Throughout symbolic evaluation, each statement or a path is interpreted by substituting the current symbolic value of a variable wherever that variable is referenced. Thus, wherever the variable y_I is referenced, its current symbolic value, which is denoted by $s(y_I)$, is used. When an assignment statement, such as $y_J = y_K * y_L$, is interpreted, the algebraic expression $s(y_K) * s(y_L)$ is generated and provides the new symbolic value for y_J. For the assignment statement at node 3 in TRANSACT, for example, the current symbolic values of BALANCE, RATE, and DAYS are substituted into the expression, resulting in the symbolic value

balance*(1+0.06/365)**days,

which is assigned for the variable NEWBAL. When interpreting a branch predicate, such as $bp(n_i, n_j) = (y_K > y_L)$, the conditional expression $(s(y_K) > s(y_L))$ is generated and provides a symbolic value for the branch predicate, which is denoted by $s(bp(n_i, n_j))$. To interpret bp(8,11) on path P_2 in TRANSACT, the current symbolic values of AMOUNT and NEWBAL are substituted into the branch predicate, resulting in the conditional expression

-amount <= balance*(1+0.06/365)**days

The interpretations of all the statements on path P_2 in TRANSACT are shown in Figure 4.

Following the symbolic evaluation of a path P_H, the symbolic values for the output parameters define the path computation, which is denoted $C[P_H]$. In TRANSACT, the output parameters are BALANCE, INTEREST, BELOWMIN, and OVERDRAFT, and thus for each path P_H, $C[P_H]$ = (s(BALANCE), s(INTEREST), s(BELOWMIN), s(OVERDRAFT)). The conjunction of the symbolic values of the branch predicates defines the path domain and is referred to as the path condition, denoted by $PC[P_H]$. Only the input values that satisfy the path condition could cause execution of the path. For path P_2 in TRANSACT, $PC[P_2]$ = s(bp(5,7)) and s(bp(7,8)) and s(bp(8,11)) and s(bp(12,13)). Figure 5 shows the path computation and path condition that result from symbolic evaluation of path P_2.

The three methods of symbolic evaluation each use an interpretive technique similar to that described above to develop representations of the path computations and path conditions. Symbolic execution is a path-dependent method of symbolic evaluation that provides the path computation and path condition for

statement or edge	interpreted branch predicate	interpreted assignments	
s	true	DAYS=days, BALANCE=balance, BELOWMIN=?, NEWBAL=?, MINBAL=100.0, ODCHARGE=4.0	AMOUNT=amount, INTEREST=?, OVERDRAFT=?, RATE=0.06, BMCHARGE=0.1,
1		OVERDRAFT=false	
2		BELOWMIN=false	
3		NEWBAL=balance*(1+0.06/365)**days	
4		INTEREST=balance*(1+0.06/365)**days-balance	
(5,7)	amount <= 0.0		
(7,8)	amount < 0.0		
(8,11)	-amount <= balance*(1+0.06/365)**days		
11		NEWBAL=balance*(1+0.06/365)**days+amount	
(12,13)	balance*(1+0.06/365)**days+ amount < 100.0		
13		BELOWMIN=true	
14		NEWBAL=balance*(1+0.06/365)**days+amount-0.1	
15		BALANCE=balance*(1+0.06/365)**days+amount-0.1	
f			

Figure 4. Symbolic Evaluation of Path P_2 in TRANSACT

<u>Path Computation</u> C[P_2]

```
    BALANCE = balance*(1+0.06/365)**days + amount - 0.1
    INTEREST = balance*(1+0.06/365)**days - balance
    BELOWMIN = true
    OVERDRAFT = false
```

<u>Path Condition</u> PC[P_2]

```
    (amount <= 0.0) and (amount < 0.0) and
    (-amount <= balance*(1+0.06/365)**days) and
    (balance*(1+0.06/365)**days + amount < 100.0)
```

Figure 5. Path Computation and Path Condition for Path P_2 in TRANSACT

a given path. The final results produced for a path are similar to those shown for path P_2 of TRANSACT in Figure 5. Dynamic symbolic evaluation is a data-dependent method that analyzes a path while the routine is actually being executed for specific input data. This method interpets the statements that are executed on the path. In addition to supplying the actual output values that result from the execution, dynamic symbolic evaluation provides symbolic representations of the path computation and path condition. If input data (DAYS=20, AMOUNT=-10.00, BALANCE=100.00) is supplied for TRANSACT, path P_2 is executed. Dynamic symbolic evaluation would then provide the actual output values (BALANCE=90.23, INTEREST=0.33, BELOWMIN=true, OVERDRAFT=false), as well as the path computation and path condition shown in Figure 5. Both symbolic execution and dynamic symbolic evaluation analyze a routine on a path-by-path basis. Dynamic symbolic evaluation chooses the paths based on the supplied test data, while symbolic execution requires some other method for selecting the paths.

Rather than evaluate a routine on a path-by-path basis, global symbolic evaluation creates a case expression[1] that encompasses all paths. For a routine that contains no loops, such as TRANSACT, the results of global symbolic evaluation are equivalent to the results produced by symbolic execution when all paths are selected. For such a routine the case expression consists of a case for each path, where each case is represented by the path condition and the path computation. Figure 6 shows this expression for the executable paths in TRANSACT. For a routine that contains a loop, global symbolic evaluation uses a loop analysis technique to develop a closed form representation of the effects of the loop. This allows paths that differ only by the number of loop iterations to be grouped as a class of paths. Thus, in general, a case may be represented by the path condition for a class of paths and the path computation associated with that class.

All three methods of symbolic evaluation create symbolic representations of the path condition and path computation. The information that is gathered to achieve these representations differs significantly, thus affecting the types of program analysis that can be performed. Dynamic symbolic evaluation maintains only the information required to develop the final symbolic represertions and its applications are usually restricted to program debugging. Symbolic execution maintains more general information about a path and thus has a more extensive range of applications, including test data generation, error detection, and program validation. Global symbolic evaluation analyzes all paths and maintains a global representation of a routine and, in addition to the applications of symbolic execution, thereby has applications to program optimization and verification. It is not surprising that the more powerful the method, the more costly its implementation. All three methods of symbolic evaluation, basic approaches for their implementation and their primary applications, will be explained and compared in the sections that follow.

3. SYMBOLIC EXECUTION

Symbolic execution analyzes distinct paths. In general, symbolic execution 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 symbolic execution that follows is independent of the method of path selection; it is assumed that path selection information is provided externally. This section first describes and compares several techniques used in implementing symbolic execution. Then a discussion of the applications of symbolic execution is presented and several methods for selecting the paths to be analyzed are described.

```
case
   (amount <= 0.0) and (amount < 0.0) and
   (-amount <= balance*(1+0.06/365)**days) and
   (balance*(1+0.06/365)**days + amount < 100.0):
      BALANCE = balance*(1+0.06/365)**days + amount - 0.1
      INTEREST = balance*(1+0.06/365)**days - balance
      BELOWMIN = true
      OVERDRAFT = false

   (amount <= 0.0) and (amount < 0.0) and
   (-amount > balance*(1+0.06/365)**days) and
   (balance*(1+0.06/365)**days - 4.0 < 100.0):
      BALANCE = balance*(1+0.06/365)**days - 4.0 - 0.1
      INTEREST = balance*(1+0.06/365)**days - balance
      BELOWMIN = true
      OVERDRAFT = true

   (amount <= 0.0) and (amount < 0.0) and
   (-amount <= balance*(1+0.06/365)**days) and
   (balance*(1+0.06/365)*days + amount >= 100.0):
      BALANCE = balance*(1+0.06/365)**days + amount
      INTEREST = balance*(1+0.06/365)**days - balance
      BELOWMIN = false
      OVERDRAFT = false

   (amount <= 0.0) and (amount < 0.0) and
   (-amount > balance*(1+0.06/365)**days) and
   (balance*(1+0.06/365)**days - 4.0 >= 100.0):
      BALANCE = balance*(1+0.06/365)**days - 4.0
      INTEREST = balance*(1+0.06/365)**days - balance
      BELOWMIN = false
      OVERDRAFT = true

   (amount > 0.0) and (amount >= 0.0):
      BALANCE = balance*(1+0.06/365)**days + amount
      INTEREST = balance*(1+0.06/365)**days - balance
      BELOWMIN = false
      OVERDRAFT = false

   (amount <= 0.0) and (amount >= 0.0):
      BALANCE = balance*(1+0.06/365)**days
      INTEREST = balance*(1+0.06/365)**days - balance
      BELOWMIN = false
      OVERDRAFT = false
endcase
```

Figure 6. Global Symbolic Evaluation of TRANSACT

3.1 Implementation Approaches

Several symbolic execution systems have been described [BOYE75,CLAR76b,HOWD77, HUAN75,KING76,MILL75,RAMA76,VOGE80]. These systems employ either of two evaluation techniques, forward expansion or backward substitution. In addition, some of these systems try to determine path condition consistency [BOYE75,CLAR76b,KING76,RAMA76], and again two different techniques, algebraic or axiomatic, have successfully been applied. This section describes the implementation approach taken by ATTEST [CLAR78], which uses forward expansion to develop the symbolic representations and employs an algebraic technique to determine the consistency of the path condition. The backward substitution and axiomatic techniques are also discussed and compared to their respective alternatives.

Forward expansion is the most intuitive approach to creating the symbolic representations and is the interpretive technique outlined in the previous section. Forward expansion begins with the start node and builds symbolic expressions as each statement in the path is interpreted. To facilitate this interpretation, the ATTEST system first translates the source code into an intermediate form of binary expressions, each containing an operator and two operands. During forward expansion, the binary expressions of the interpreted statements are used to form an acyclic directed graph, called the computation graph, which maintains the symbolic values of the variables. When a variable is assigned a value, it is actually assigned a pointer into this graph. The node of the computation graph that is pointed to by a variable can be treated as the root of a binary expression tree. Traversing this tree in in-order (i.e., left subtree, root, right subtree) provides the symbolic value for this variable. The symbolic value of a branch predicate is similarly maintained in the computation graph as a binary expression tree. Figure 7 shows the computation graph at two stages during symbolic execution of path P_2 in TRANSACT. There is a close similarity between the forward expansion technique described here and the technique of common subexpression elimination used by optimizing compilers [COCK70].

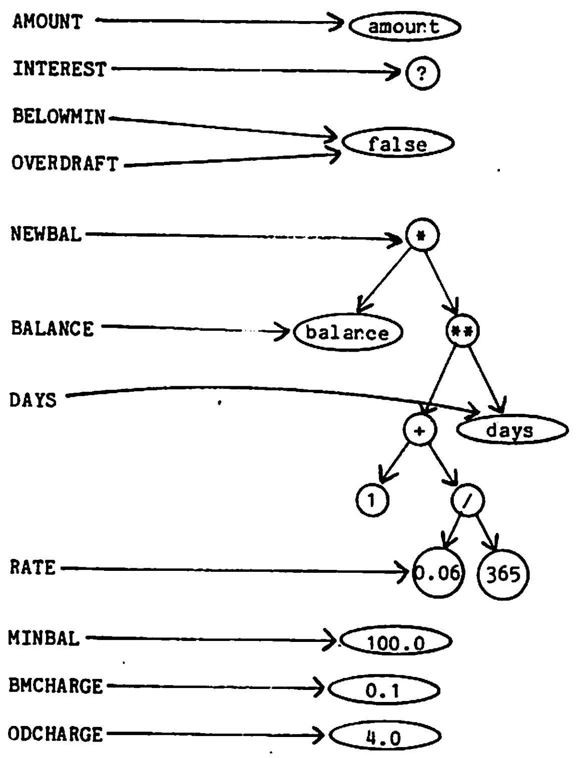

Figure 7a. Computation Graph after Interpretation of Statements s,1,2,3 in TRANSACT

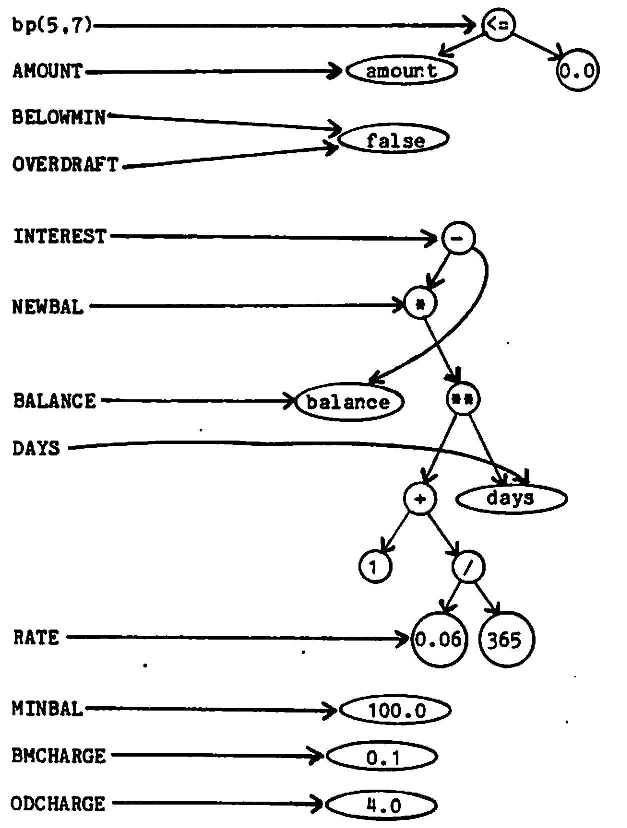

Figure 7b. Computation Graph after Interpretation of Statements s,1,2,3,4,5,7 of TRANSACT

After evaluation of a path, the path computation is obtained by traversing the binary expression trees for the output parameters. The path condition is created by traversing the binary expression trees for the interpreted branch predicates and conjoining the resulting symbolic values. In the purest sense, the path computation and path condition are all that need be provided by symbolic execution. To do further analysis, however, it is desirable to simplify the symbolic representations and to determine the consistency of the path condition.

Simplification can be done by converting the path computation and path condition into canonical forms. There are several available algebraic manipulation systems [BOGE75,BROW73,RICH78] that can be used to accomplish this simplification. A canonical form for the symbolic values for the output parameters, and thus for the path computation, might be one in which like terms are grouped together and terms are ordered lexically. For example, the simplified symbolic value for BALANCE for path P_2 is

amount + 1.00016**days*balance - 0.10.

The path condition 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 righthand side of the relational operator and, on the lelfthand side, like terms are grouped together and terms are ordered lexically.

In most cases, only a subset of the paths in a program are executable and, therefore, it is desirable to determine whether or not the path condition is consistent. One approach to this problem employs a theorem proving system. We refer to this as the axiomatic technique since it is based upon the axioms of predicate calculus. Another approach, referred to as the algebraic technique, treats the path condition as a system of constraints and uses one of several algebraic methods -- such as a gradient hill-climbing algorithm or linear programming -- to solve this system of constraints. The ATTEST system uses a linear programming system [LAND73] and thus employs the algebraic technique. The advantage of choosing this technique is that a solution is provided when the path condition is determined to be consistent. This solution provides test data to execute the path. Both the axiomatic and algebraic techniques work well on the simple constraints that are generally created during symbolic execution [CLAR76a]. No method, however, can solve all arbitrary systems of constraints [DAVI73]. In some instances, path condition consistency can not be determined; the symbolic representations for such a path can be provided, but whether or not the path can be executed is unknown.

During symbolic execution it is desirable not only to recognize nonexecutable paths but to recognize the inconsistency as soon as possible. Early detection of a nonexecutable path prevents worthless, yet costly, symbolic execution. The ATTEST system attempts to detect a nonexecutable path as soon as possibe by examining the evolving path condition as each branch predicate is interpreted. ATTEST develops the path condition as the statements on a path are interpreted. Thus at any point in the interpretation, there is a representation of the path condition for the partial path that has been evaluated so far. For partial path $P_{Hu} = (n_s, n_{H1}, \ldots, n_{Hu})$, the path condition is denoted $PC[P_{Hu}]$. When a node n_{Hu+1} is considered as an extension to the partial path P_{Hu}, the interpreted branch predicate $s(bp(n_{Hu}, n_{Hu+1}))$ is first simplified and then examined for consistency with the existing path condition $PC[P_{Hu}]$. Unless inconsistency is determined, the interpreted branch predicate is conjoined to the path condition, creating

$$PC[P_{Hu+1}] = PC[P_{Hu}] \text{ and } s(bp(n_{Hu}, n_{Hu+1})).$$

Consistency or inconsistency may be determined by performing simple reductions on the path condition [DEUT73,DILL81]. On the one hand, it may be possible to determine that $s(bp(n_{Hu}, n_{Hu+1})$ is dominated by relational expressions in $PC[P_{Hu}]$, in which case $PC[P_{Hu+1}]$ must be consistent, since $PC[P_{Hu}]$ is consistent. In the evaluation of path P_9, for example, s(bp(7,15)) = (amount >= 0.0) is dominated by s(bp(5,6)) = (amount > 0.0) and thus PC[s,1,2,3,4,5,6,7,15] is consistent. On the other hand, $s(bp(n_{Hu}, n_{Hu+1}))$ may be contradicted by a relational expression in $PC[P_{Hu}]$, in which case $PC[P_{Hu+1}]$ is inconsistent. In the evaluation of path P_1, for example, s(bp(7,8)) = (amount < 0.0) is contradicted by s(bp(5,6)) = (amount > 0.0), and thus PC[s,1,2,3,4,5,6,7,8] is inconsistent. While such reductions are sometimes applicable, it is often necessary to rely on more sophisticated techniques, such as those described above.

In addition to detecting nonexecutable paths early in the symbolic execution process, the incremental development of the path condition as implemented by ATTEST 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 to an inconsistent branch predicate can usually be salvaged. For example, the nonexecutable partial path (s,1,2,3,4,5,6,7,8) in TRANSACT, shown in Figure 8, was terminated as soon as the inconsistent path condition was discovered. The symbolic value of the branch predicate for the edge (7,8), where the inconsistency occurred, is replaced by the symbolic value of the branch predicate for the edge (7,15), and analysis continues.

In general, when there is more than one successor node to the last node on the partial path, each may be considered as an alternative extension of the existing partial path. Note that the alternative branch predicates either all evaluate to

statement or edge	simplified, evolving path condition	simplified, interpreted assignments
s	true	DAYS=days, AMOUNT=amount, BALANCE=balance, INTEREST=?, BELOWMIN=?, OVERDRAFT=?, NEWBAL=?, RATE=0.06, MINBAL=100.0, BMCHARGE=0.1, ODCHARGE=4.0
1		OVERDRAFT=false
2		BELOWMIN=false
3		NEWBAL=balance*(1+0.06/365)**days =1.00016**days*balance
4		INTEREST=1.00016**days*balance-balance =(1.00016**days-1.0)*balance
(5,6)	true and amount>0.0 =amount>0.0	
6		NEWBAL=1.00016**days*balance+amount =amount+1.00016**days*balance
(7,8)	amount > 0.0 and amount < 0.0 ***inconsistent***	
delete	amount > 0.0	
alternative edge (7;15)	amount > 0.0 and amount >= 0.0 = amount > 0.0	
15		BALANCE=amount+1.00016**days*balance
f		

Figure 8. Detection of Inconsistent Path Condition and Continuation with Executable Path P_9

constant boolean values or all evaluate to symbolic expressions over the input values. If the branch predicate for a selected edge evaluates to a boolean constant then consistency determination is trivial. Otherwise, the techniques described above may be employed to determine the consistency of the interpreted branch predicate with the existing path condition. When determining consistency, there are three possible outcomes: 1) none of the alternatives is consistent; 2) only one of the alternatives is consistent; and 3) more than one of the alternatives are consistent. The graph shown in Figure 9 demonstrates all three cases. Note that the first case implies a program error and can only occur for multi-conditional statements without an otherwise clause, like the Pascal CASE statement. When all of the branch predicates evaluate to symbolic expressions, any of the three cases can occur. When they all evaluate to boolean constants, at most one of the alternatives can evaluate to true, and thus only case 1 or 2 can occur.

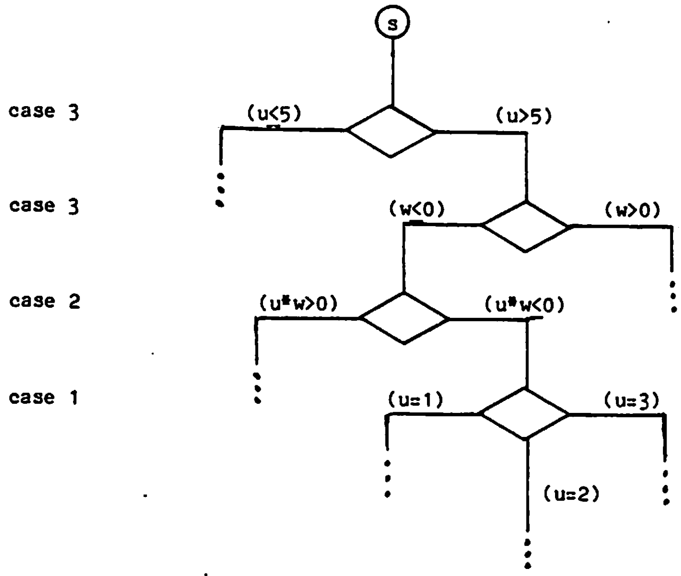

Figure 9. Examples of the 3 Cases that Can Occur During Consistency Determination

The forward expansion technique begins with the start node and works toward the final node, while the backward substitution technique [HOWD75,HUAN75] begins with the final node and works toward the start node. Backward substitution was proposed for systems concerned with creating only the path condition and not the path computation. While traversing backwards along a path, each encountered branch predicate is recorded. When an assignment to a variable referenced in any of the recorded branch predicates is encountered, the assignment expression is substituted for all occurences of that variable in the recorded branch predicates. For example, if the branch predicate $(y_J > 5)$ were traversed, and thus recorded, and then the assignment $y_J = y_K + 1$ were encountered, the recorded branch predicate would be modified to $(y_K + 1 > 5)$. Note that with backward substitution symbolic names are not assigned until the start node is encountered. After the start node is reached, the path condition is formed by conjoining all of the recorded, and duly modified, branch predicates for the path. An example of backward substitution for path P_2 in TRANSACT is shown in Figure 10. In this figure, the evolving symbolic values of the recorded branch predicates are listed at each modification point. Note that many of the assignment statements, specifically those that do not modify variables referenced in the branch predicates, can be ignored using backward substitution when only the path condition is desired. In the example of Figure 10, assignment statements 15, 14, 13, 4, 2, and 1 are ignored. In a general symbolic execution system where both the path condition and path computation are desired, the two approaches interpret each statement and produce equivalent expressions for the path condition and path computation.

Forward expansion is a more efficient technique of symbolic execution than backward substitution when early detection of nonexecutable paths is supported. In forward expansion, a branch predicate is interpreted, simplified, and checked for consistency with the existing path condition when it is encountered. In

statement or edge	recorded branch predicates
f,15,14,13	no effect
(12,13)	NEWBAL < MINBAL
11	NEWBAL + AMOUNT < MINBAL
(8,11)	NEWBAL + AMOUNT < MINBAL -AMOUNT <= NEWBAL
(7,8)	NEWBAL + AMOUNT < MINBAL -AMOUNT <= NEWBAL AMOUNT < 0.0
(5,7)	NEWBAL + AMOUNT < MINBAL -AMOUNT <= NEWBAL AMOUNT < 0.0 AMOUNT <= 0.0
4	no effect
3	BALANCE*(1+RATE/365)**DAYS + AMOUNT < MINBAL -AMOUNT <= BALANCE*(1+RATE/365)**DAYS AMOUNT < 0.0 AMOUNT <= 0.0
2,1	no effect
s	balance*(1+0.06/365)**days + amount < 100.0 -amount <= balance*(1+rate/365)**days amount < 0.0 amount <= 0.0

Figure 10. Backward Substitution for Path P_2 in TRANSACT

backward substitution, the identical process would occur when a branch predicate is encountered, but additional processing of the path condition must be done whenever an assignment is made to any variable referenced in the path condition. When this occurs, each modified branch predicate must be resimplified and path condition consistency redetermined. This additional processing during backward substitution is costly.

3.2 Applications

Symbolic execution systems have several interesting applications. This section considers three applications: validation and documentation, error detection, and test data generation. In addition, the last part of this section considers methods of path selection.

The symbolic representations that are generated for a path can quite naturally be used for validation and documentation. 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 $(z_1, \ldots, z_N)$ for particular input values $(x_1, \ldots, x_M)$. It is possible for the output values to be correct while the path computation is incorrect. This is referred to as coincidental correctness. As an example, suppose the exponent operator in statement 3 of TRANSACT is erroneously replaced by a multiply operator. This causes the path computation for BALANCE along path P_2 in TRANSACT to be

balance*(1+0.06/365)*days + amount - 0.10

rather than the intended computation, which is shown in Figure 5. If the path is always executed for days = 1., then the actual resulting value and the intended value agree. While this is a contrived example, coincidental correctness is a common phenomenon of testing. Examination of the path computation, as well as the path condition is often useful in uncovering program errors [HOWD76]. This is a particularly beneficial feature for scientific applications, where it is often extremely difficult to manually compute the intended result accurately due to the complexity of the computation as well as the number of significant digits required for the input values. This method of examining the path computation and the path condition is referred to as symbolic testing.

Symbolic execution can also be actively applied to the detection of program errors. At appropriate points in a routine, expressions describing error conditions can be interpreted and checked for consistency with the path condition just as branch predicates are interpreted. 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. This demonstrates another advantage of symbolic execution over normal execution. Normal execution of a path may not uncover a run time error, while symbolic execution of a path can detect the presence or guarantee the absence of some errors.

The ATTEST system automatically generates expressions for predefined error conditions whenever it encounters the corresponding program constructs. For example, 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 path condition. If the resulting path condition is consistent, then input values exist that would cause a division by zero error; an error report is issued. If the resulting path condition is inconsistent, then this potential run-time error could not occur on this path. After determining consistency, the expression for the error condition is removed from the path condition before symbolic execution continues.

Path verification of assertions is another method of error detection. Instead of predefining the error conditions, user-created assertions define conditions that must 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 execution, the complement of the assertion is interpreted and conjoined to the path condition. Inconsistency of the resulting path condition implies that the assertion is valid for the path, while consistency implies that the assertion is invalid for the routine.

Test data generation is another natural application of symbolic execution. The path condition is examined to determine a solution -- that is, test data to execute the path. Symbolic execution, like most other methods of program validation, 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 execution system may cause an erroneous result. In addition, testing a routine demonstrates its actual performance characteristics. SELECT[BOYE75] and ATTEST are two symbolic execution systems that attempt to generate test data. Since an actual solution to the path condition is desired and not just path condition consistency, these two systems employ an algebraic technique to solve the path condition.

Moreover, the path computation and path condition can be used to guide in astutely selecting test data. Error sensitive testing strategies have been described [FOST80,MEYE79,WEYU80] that examine the statements or intent of a routine and select test data to detect likely errors. Error sensitive testing strategies have also been applied to the path computation and path condition created by symbolic execution. Functional testing [HOWD80] modifies this approach by first decomposing a routine into small sections before applying symbolic execution and error sensitive testing. Although error sensitive testing, for the most part has been intuitive, there have been some theoretical results showing that more rigorous applications of these strategies can guarantee the absence of certain types of errors. For example, if the symbolic value for an output parameter is a polynomial, its degree can be used to determine the number of test data points needed to guarantee the correctness of this polynomial [HOWD78b]. Further, it has been argued that the selection of boundary points of the path domain can guarantee the correctness of the interpreted branch predicates within a quantifiable error bound [HASS80,WHIT80]. This approach is referred to as domain testing. A recent extension to this approach requires that the symbolic values of a branch predicate over the paths already tested be examined to determine when yet another path is needed to sufficiently test the predicate [ZEIL81]. In general, the symbolic representations created by symbolic execution provide valuable guidance in selecting test data, but further work in this area is needed.

This section assumed that the paths to be analyzed by symbolic execution are provided. These paths can either be chosen by the user or be selected automatically by a component of the symbolic execution system. Most symbolic execution systems support an interactive path selection facility that allows the user to "walk through" a program, statement by statement. This feature is useful for debugging since the evolution of the path computation and path condition can be observed. More extensive program coverage requires an automated path selection facility for choosing a set of paths based on some criterion, which is dependent on the intended application of symbolic execution.

Three criteria that are often 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. Testing the program on a set of paths satisfying this criterion is called statement testing. Likewise, branch coverage requires that each branch predicate occurs at least once on one of the selected paths and testing such a set of paths is called branch testing. Path coverage requires that all paths be selected; this is referred to as path testing. Branch coverage implies statement coverage, while path coverage implies branch coverage. Path coverage, in fact, implies the selection of all feasible combinations of branch predicates, which may require an infinite number of paths. Because of the impracticality of path coverage, alternative criteria have been proposed that limit loop iterations. For example, the EFFIGY system [KING76] puts an arbitrary bound on the number of loop iterations. Howden has proposed an approximate path coverage criterion that requires 0, 1, and 2 iterations of all loops. The ATTEST system tries to select paths that traverse each loop a minimum and maximum number of times.

Automatically selecting a set of paths to satisfy any one of these criteria is nontrivial since nonexecutable paths must be excluded [GABO76]. The ATTEST system, for example, uses a dynamic, goal-oriented method of path selection. In this system, a path is selected, statement by statement, as symbolic execution proceeds. A statement is selected based on its potential for satisfying the path selection criterion, which can be statement, branch, or path coverage. As described above, when an infeasible path is encountered, ATTEST chooses one of the alternative statements. When there is more than one consistent alternative, this choice is based on the selection criterion. A more complete description of path selection methods for symbolic execution systems can be found in [WOOD80].

4. DYNAMIC SYMBOLIC EVALUATION

Dynamic symbolic evaluation is one of the features provided by dynamic testing systems [BALZ69,FAIR75]. Using test data to determine the path, dynamic symbolic evaluation systems provide symbolic representations of the path computation. This section gives a brief overview of dynamic testing systems and then describes dynamic symbolic evaluation, its implementation techniques and its applications.

4.1 Implementation Approach

Dynamic testing systems monitor a routine's behavior during execution to create a profile of that execution. Some of the types of information in a profile include the number of times each statement was executed, the number of times each edge was traversed, the minimum and maximum number of times each loop was traversed, the minimum and maximum values assigned to variables, and the path that was executed. In addition, some of these systems create an accumulated profile of all execution runs.

To collect the information in a profile, dynamic testing systems usually insert calls to analysis procedures at appropriate places in the code. This process, which is referred to as instrumentation, is generally done by a preprocessor [HUAN78]. Dynamic testing systems also provide a driver program, or test harness, to initialize the parameters and global variables of the instrumented routine to values supplied by the user.

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 values, 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 actual computed values. As with symbolic execution, 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 symbolic execution is a natural approach for creating these symbolic values. These expressions can be maintained internally as a computation graph similar to that shown for symbolic execution. The computation graph is augmented, however, to include the actual value computed for each node.

After executing path P_H, the symbolic value for each output parameter is shown, providing $C[P_H]$. Generally, dynamic symbolic evaluation systems display these expressions 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 TRANSACT are shown in Figure 11. Note that these input values cause path P_2 to be executed. Figure 12 shows the final results that might be produced.

Existing dynamic symbolic evaluation systems are only concerned with the path computation. Since the input values are known, each branch predicate evaluates to the constant value true (or a run-time error is encountered). The path condition is, therefore, equal to true and thus it is not necessary to check for path condition consistency. It would be easy to extend dynamic symbolic evaluation to provide a symbolic representation of the path condition. Note that the actual value is known for each output parameter, but the symbolic representation of the path computation is still provided. Examination of the path condition, like examination of the path computation, may uncover errors in the routine. An erroneous path condition would imply an erroneous branch predicate or erroneous calculation affecting a branch predicate.

Symbolic and Actual Input Values
DAYS = days = 20
AMOUNT = amount = -10.00
BALANCE = balance = 100.00

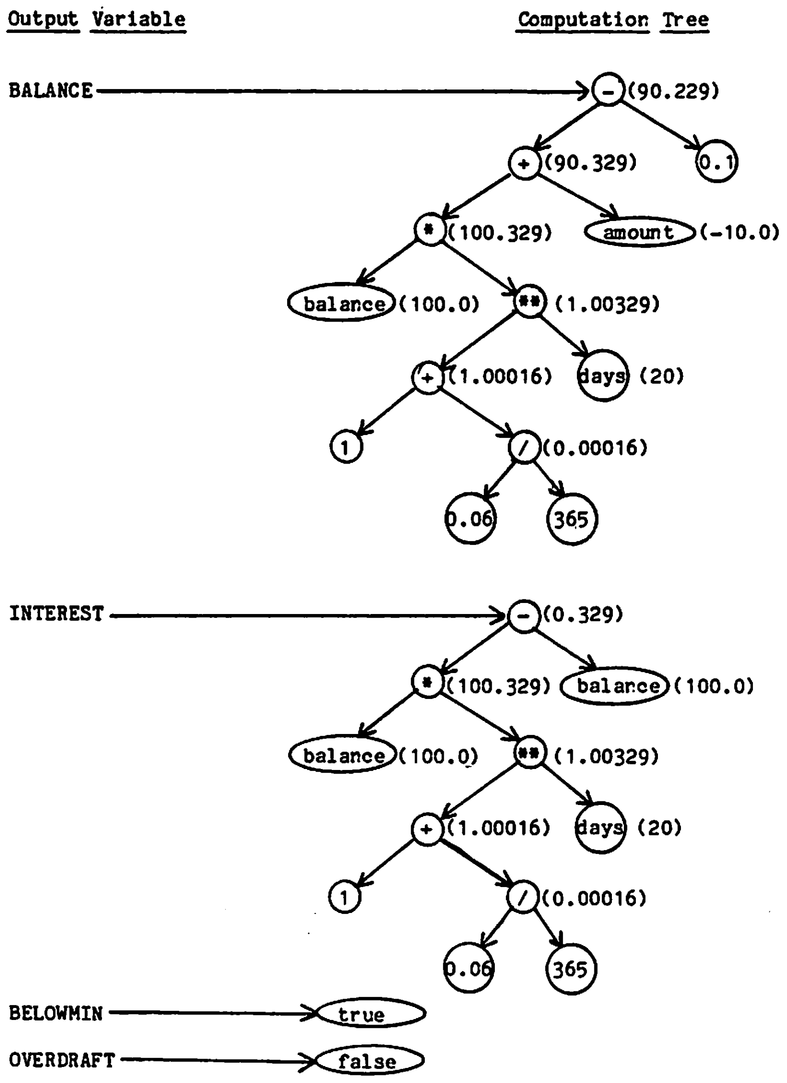

Figure 11. Computation Trees Produced by Dynamic Symbolic Evaluation of Path P2 of TRANSACT

```
Symbolic and Actual Input Values
   DAYS = days = 20
   AMOUNT = amount = -10.00
   BALANCE = balance = 100.0

Statements Executed
   (s, 1, 2, 3, 4, 5, 7, 8, 11, 12, 13, 14, 15, f)

Symbolic and Actual Output Values
   BALANCE = balance*(1+0.06/365)**days + amount - 0.1 = 90.229
   INTEREST = balance*(1+0.06/365)**days - balance = 0.329
   BELOWMIN = true
   OVERDRAFT = false
```

Figure 12. Final Results of Dynamic Symbolic Evaluation of Path P_2 of TRANSACT

4.2 Applications

The primary application of dynamic symbolic evaluation is program debugging. When an error is uncovered in a routine, dynamic symbolic evaluation systems provide a picture of the path computation, which can be examined to help isolate the cause of the error. To assist in debugging, these systems provide a capability for examining the computation trees while they are being constructed statement-by-statement. These systems also 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 preceeding 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 [FAIR75] have shown that both of these features are beneficial for debugging.

Another feature sometimes provided by dynamic testing systems is the ability to check user-created assertions at run time [FAIR75,STUC73]. Unlike the assertion checking done by sumbolic execution systems, dynamic assertion checking is done just for the supplied input values and not for the entire path domain. Either the assertion is true and thus valid for the input values, or the assertion is false and thus invalid for the routine, in which case an error message is issued.

Dynamic symbolic evaluation can assist in statement, branch, and path testing. The execution profile provided by dynamic symbolic evaluation systems usually contains statement execution counts, edge traversal counts, and descriptions of the paths executed. This information is helpful in determining when a program has been tested sufficiently, based on any one of these testing strategies. The responsibility of achieving this coverage, however, falls on the user.

The symbolic representations provided by dynamic symbolic evaluation provide valuable guidance in selecting test data. As with symbolic execution, these representations can be used in applying error sensitive testing strategies. Further, the simplification of the relational expressions and determination of path condition consistency, which we argued is a desirable although expensive enhancement to symbolic execution, is not needed to determine whether or not a path is executable. To provide further analysis, such as automated error detection, however, dynamic symbolic evaluation must also include these capabilities and incur the associated expense.

5. GLOBAL SYMBOLIC EVALUATION

The goal of global symbolic evaluation [CHEA79a,PLOE79] is the derivation of a global representation of a routine -- a representation of all variables for all paths, rather than along a specific path. In other words, global symbolic evaluation results in a global representation of an entire routine. This representation is acheived by classifying paths so that the paths in a class differ only by their number of loop iterations. This section describes the interpretive technique employed by global symbolic evaluation and explains the loop analysis technique used to classify loops. Several applications of global symbolic evaluation are also discussed.

5.1 Implementation Approach

Global symbolic evaluation, like symbolic execution, 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. Inner loops must be analyzed before outer loops. An analyzed loop can be replaced by the resulting loop expression, which can thereafter be evaluated as a single node. After all loops have been analyzed, the control flow graph has been reduced to a directed acyclic graph. Global symbolic evaluation then selects the order in which the nodes are to be interpreted so that all predecessors of a node are interpreted before that node is interpreted. In this section, the interpretive technique of global symbolic evaluation is first described for routines without loops. Then the loop analysis technique is introduced along with the technique for incorporating its results, thereby describing the application of global symbolic evaluation to routines with loops.

In global symbolic evaluation, as in symbolic execution, the input values are represented by symbolic names, and throughout the analysis the symbolic values of all variables are represented as algebraic expressions in terms of these symbolic names. Furthermore, these symbolic values can be maintained as a computation graph similar to that described for symbolic execution. The technique used to interpret a statement is the same as that employed by symbolic execution. In interpreting a particular node, however, symbolic execution only considers the evaluation of one partial path reaching that node, whereas global symbolic evaluation considers the evaluation of all such partial paths. For a node in the control flow graph, global symbolic evaluation maintains a case expression, where each case represents one partial path reaching the node. Each case is composed of the path condition 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 n_k, with predecessor nodes $n_i, \ldots, n_j$, which have been previously interpreted. Control may reach n_k via any of the edges $(n_i, n_k), \ldots, (n_j, n_k)$, and the transfer from a predecessor node occurs under the conditions of the corresponding branch predicate. Thus, when n_k is interpreted, the case expressions of all predecessor nodes must be considered. For node n_i, the branch predicate $bp(n_i, n_k)$ is evaluated in the context of each case in the case expression. For a particular case, $bp(n_i, n_k)$ is interpreted in terms of the symbolic values of the variables for this case. This interpreted branch predicate is then conjoined to the path condition for the associated partial path. As with symbolic execution, it is desirable to check the consistency of the path condition. For routines without loops, the techniques described for symbolic execution could be applied. If the path condition is found to be inconsistent, this case is discarded. Otherwise, the statement at node n_k must be interpreted in the context of this case. After all the cases for node n_i have been considered, the same procedure is followed for all other predecessor nodes. The updated case expressions associated with the predecessor nodes are combined and the resulting case expression represents all executable partial paths reaching node n_k.

Figure 13 shows the global symbolic evaluation for TRANSACT; only the start node, the final node, and the nodes corresponding to conditional statements are shown. For these nodes, each case is annotated with the corresponding partial path. Note that the initial values of all variables are shown at the start node, but thereafter the symbolic values are shown only for the variables that can be modified. Observe that node 7 has two cases in its case expression. Since node 7 is a predecessor to node 8, both of these cases are considered in evaluating node 8. Thus, the conditional statement -AMOUNT > NEWBAL is interpreted in terms of the symbolic values of the first case for node 7, providing one case for node 8. This conditional statement is also interpreted in the context of the second case for node 7, providing the second case for node 8. However, s(bp(7,8)) is inconsistent with the path condition associated with the first case. The resulting inconsistent path condition is shown in the case expression for node 8, but is not carried further in the analysis.

The routine TRANSACT does not contain a loop, so the global symbolic evaluation is as described. For routines with loops, however, the described case expression representing all partial paths into a node is only possible because of the loop analysis technique employed by global symbolic evaluation. This loop analysis technique attempts to represent each loop by a loop expression, a closed form representation describing the effects of that loop. For each analyzed loop, a conditional expression is created representing the final iteration count for any arbitrary execution of the loop. The final iteration count is expressed in terms of the symbolic values of the variables at entry to the loop. In addition, for each variable modified within the loop, its symbolic value at exit from the loop is created. Each such expression is in terms of the final iteration count as well as the symbolic values of the variables at entry to the loop. The routine

```
s       case
          true:
             DAYS = days
             AMOUNT = amount
             BALANCE = balance
             INTEREST = ?
             BELOWMIN = ?
             OVERDRAFT = ?
             NEWBAL = ?
             RATE = 0.06
             MINBAL = 100.0
             BMCHARGE = 0.1
             ODCHARGE = 4.0
        endcase

5       case
          -- (s,1,2,3,4,5)
          true:
             BALANCE = balance
             INTEREST = 1.00016**days*balance - balance
                      = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL = balance*(1+0.06/365)**days
                    = 1.00016**days*balance
        endcase
```

Figure 13. Global Symbolic Evaluation of TRANSACT

```
7       case
          -- (s,1,2,3,4,5,6,7)
          (amount > 0.0):
             BALANCE = balance
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL = 1.00016**days*balance + amount
                    = amount + 1.00016**days*balance

          -- (s,1,2,3,4,5,7)
          (amount <= 0.0):
             BALANCE = balance
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL= 1.00016**days*balance
        endcase

8       case
          -- (s,1,2,3,4,5,6,7,8)
          (amount > 0.0) and (amount < 0.0)
        = false:

          -- (s,1,2,3,4,5,7,8)
          (amount <= 0.0) and (amount < 0.0)
        = (amount < 0.0):
             BALANCE = balance
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL= 1.00016**days*balance
        endcase

12      case
          -- (s,1,2,3,4,5,7,8,11,12)
          (amount < 0.0) and (-amount <= 1.00016**days*balance)
        = (amount < 0.0) and (amount + 1.00016**days*balance >= 0.0):
             BALANCE = balance
             INTEREST = (1.0016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL = 1.00016**days*balance + amount
                    = amount + 1.00016**days*balance

          -- (s,1,2,3,4,5,7,8,9,10,12)
          (amount < 0.0) and (-amount > 1.00016**days*balance)
        = (amount < 0.0) and (amount + 1.00016**days*balance < 0.0):
             BALANCE = balance
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = true
             NEWBAL = 1.00016**days*balance - 4.0
        endcase
```

Figure 13. (continued)

```
f       case
          -- (s,1,2,3,4,5,7,8,11,12,13,14,15,f)
          (amount < 0.0) and (amount + 1.00016**days*balance >= 0.0) and
          (amount + 1.0016**days*balance < 100.0):
             BALANCE = amount + 1.00016**days*balance - 0.1
             INTEREST = (1.0016**days-1.0)*balance
             BELOWMIN = true
             OVERDRAFT = false
             NEWBAL = amount + 1.00016**days*balance - 0.1

          -- (s,1,2,3,4,5,7,8,9,10,12,13,14,15,f)
          (amount < 0.0) and (amount + 1.00016**days*balance < 0.0) and
          (1.00016**days*balance - 4.0 < 100.0)
        = (amount < 0.0) and (amount + 1.00016**days*balance < 0.0) and
          (1.00016**days*balance < 104.0):
             BALANCE = 1.00016**days*balance - 4.1
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = true
             OVERDRAFT = true
             NEWBAL = 1.00016**days*balance - 4.0 - 0.1
                    = 1.00016**days*balance - 4.1

          -- (s,1,2,3,4,5,7,8,11,12,15,f)
          (amount < 0.0) and (amount + 1.00016**days*balance >= 0.0) and
          (amount + 1.0016**days*balance >= 100.0):
             BALANCE = amount + 1.00016**days*balance
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL = amount + 1.00016**days*balance

          -- (s,1,2,3,4,5,7,8,9,10,12,15,f)
          (amount < 0.0) and (amount + 1.00016**days*balance < 0.0) and
          (1.00016**days*balance - 4.0 >= 100.0)
        = (amount < 0.0) and (amount + 1.00016**days*balance < 0.0) and
          (1.00016**days*balance >= 104.0):
             BALANCE = 1.00016**days*balance - 4.0
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = true
             NEWBAL = 1.00016**days*balance - 4.0

          -- (s,1,2,3,4,5,6,7,15,f)
          (amount > 0.0) and (amount >= 0.0)
        = (amount > 0.0):
             BALANCE = amount + 1.00016**days*balance
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL = amount + 1.00016**days*balance

          -- (s,1,2,3,4,5,7,15,f)
          (amount <= 0.0) and (amount >= 0.0)
        = (amount = 0.0):
             BALANCE = 1.00016**days*balance
             INTEREST = (1.00016**days-1.0)*balance
             BELOWMIN = false
             OVERDRAFT = false
             NEWBAL = 1.00016**days*balance
        endcase
```

Figure 13. (continued)

```
procedure TRAP( A, B: in real; N: in integer;
                AREA: out real; ERROR: out boolean) is
-- TRAP is an implementation of the trapezoidal rule.
-- TRAP approximates the area between the curve
-- F(x) and the x-axis from x = A to x = B.
-- The approximation uses N intervals of size (B-A)/N.
-- An error is returned if N is less than 1.
   H: real; -- approximation interval
   X: real; -- value along x-axis
   YOLD: real; -- value of F(X-H)
   YNEW: real; -- value of F(X)
s begin
1   if N < 1 then
2      ERROR := true;
    else
3      ERROR := false;
4      AREA := 0.0;
5      if A /= B then
6         H := (B-A)/N;
7         X := A;
8         YOLD := F(X);
9         while X < B loop
10            X := X + H;
11            YNEW := F(X);
12            AREA := AREA + (YOLD + YNEW) / 2.0;
13            YOLD := YNEW;
          endloop;
14        AREA := AREA*H;
15        if A > B then
16           AREA := -AREA;
          endif;
      endif;
    endif;
f end TRAP;
```

Figure 14. Routine TRAP

TRAP, shown in Figure 14, contains a loop, and Figure 15 shows the results from loop analysis. This example will be explained throughout the remainder of this section.

Loop analysis proceeds from the inner-most loops outward. 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 and the statements within the loop can be interpreted by a technique similar to that previously described for loop-free routines.

To initiate loop analysis, an iteration counter, say k, is associated with the loop. For a variable V whose value may change within the loop, a special symbolic name V_k is used to represent the value of the variable V on exit from the kth iteration of the loop. Note that V_0 represents the value of the variable V on entry to the first iteration of the loop. The branch predicates controlling exit from the loop are interpreted in terms of the symbolic names V_k. This provides a representation for the loop exit condition, denoted lec_k, which is the condition under which the loop will be exited after the kth iteration. Next, symbolic evaluation of a representative iteration, k, is performed by interpreting the statements in the loop. This provides a recurrence relation for each V_k, $k \geq 1$, which is in terms of the values of the variables at iterations k-1 and k. Figure 15a shows the results of this evaluation for the WHILE loop in TRAP.

```
$X_k = X_{k-1} + H$
$YNEW_k = F(X_k)$
$AREA_k = AREA_{k-1} + (YOLD_{k-1} + YNEW_k) / 2.0$
$YOLD_k = YNEW_k$
$lec_k = not (X_k < B)$
```

Figure 15a. Recurrence Relations and
Loop Exit Condition for TRAP (k>=1)

```
case
   (k=1): -- exit from first iteration
      $X_1 = H + X_0$
      $YNEW_1 = F(H+X_0)$
      $AREA_1 = AREA_0 + YOLD_0/2.0 + F(H+X_0)/2.0$
      $YOLD_1 = F(H+X_0)$
      $lec_1 = (-B + H + X_0 >= 0.0)$

   (k>1): -- exit from second or subsequent iteration
      $X_k = H + X_{k-1}$
      $YNEW_k = F(H+X_{k-1})$
      $AREA_k = AREA_{k-1} + F(X_{k-1})/2.0 + F(H+X_{k-1})/2.0$
      $YOLD_k = F(H+X_{k-1})$
      $lec_k = (-B + X_k >= 0.0)$
endcase
```

Figure 15b. Simplified Recurrence Relations and
Simplified Loop Exit Condition for TRAP

```
case
   (k=1): -- exit from first iteration
      X(1) = $H + X_0$
      YNEW(1) = $F(H+X_0)$
      AREA(1) = $AREA_0 + YOLD_0/2.0 + F(H+X_0)/2.0$
      YOLD(1) = $F(H+X_0)$
      lec(1) = $(-B + H + X_0 >= 0.0)$

   (k>1): -- exit from second or subsequent iterations
      X(k) = $k*H + X_0$
      YNEW(k) = $F(k*H+X_0)$
      AREA(k) = $AREA_0 + YOLD_0/2.0 + F(H+X_0)/2.0 +$
                sum< i:=2..k | $F(-H+i*H+X_0)/2.0 + F(i*H+X_0)/2.0$ >
              = $AREA_0 + YOLD_0/2.0 + F(k*H+X_0)/2.0 +$
                sum< i:=1..k-1 | $F(i*H+X_0)$ >
      YOLD(k) = $F(k*H+X_0)$
      lec(k) = $(-B + k*H + X_0 >= 0.0)$
endcase
```

Figure 15c. Solved Recurrence Relations and
Solved Loop Exit Condition for TRAP

```
case
   -- fall through
   (-B + X_0 >= 0.0):
      X = X_0
      YNEW = YNEW_0
      AREA = AREA_0
      YOLD = YOLD_0

   -- exit after first iteration
   (-B + X_0 < 0.0) and
   (-B + H + X_0 >= 0.0):
      X = H + X_0
      YNEW = F(H+X_0)
      AREA = AREA_0 + YOLD_0/2.0 + F(H+X_0)/2.0
      YOLD = F(H+X_0)

   -- exit after second or subsequent iteration
   (-B + X_0 < 0.0) and
   (-B + H + X_0 < 0.0) and
   (k_e = min< k| (k > 1) and (-B + k*H + X_0 >= 0.0) >:
      X = k_e*H + X_0
      YNEW = F(k_e*H + X_0)
      AREA = AREA_0 + YOLD_0/2.0 + F(k_e*H+X_0)/2.0 +
                sum< i:=1..k_e-1 | F(i*H+X_0) >
      YOLD = F(k_e*H + X_0)
endcase
```

Figure 15d. Loop Expression for TRAP

The next step involves the simplification of the results from the symbolic evaluation of the loop. Two cases must be considered independently: 1) the first iteration of the loop, 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, where the recurrence relations and loop exit condition depend on the values computed by the previous iteration. The recurrence relations in both cases must be simplified so that the values for variables on exit from an iteration are in terms of values on entry to this iteration. Simplification is accomplished by simplifying those recurrence relations that do not reference other variables whose relations are as yet unsimplified. The simplified symbolic value for a variable is substituted into all other recurrence relations that reference that variable. This process is repeated until no more simplifications can be performed. For each case, the simplified symbolic values are then substituted into the corresponding loop exit condition and this condition is then simplified and reduced. Figure 15b shows the results of simplification for the loop in TRAP. Often the case for the first iteration and the case for subsequent iterations are identical and can be expressed as a single case.

Loop analysis now attempts to solve the recurrence relations for each case, in terms of the values of the variables on entry to the loop. The solution to the recurrence relation for V_k is denoted by V(k) and represents the symbolic value of the variable V on exit from the kth iteration of the loop. The loop exit condition lec_k is then solved by replacing each V_k referenced in the condition by its solution V(k) and simplifying. This provides lec(k), the symbolic representation of the condition under which the loop will be exited after execution of the kth iteration. Figure 15c provides the solutions for both cases of the loop in TRAP.

Finally, the loop expression can be created. The loop expression for the loop in TRAP appears in Figure 15d. Each case represents the loop exit condition and the values of the variables at exit from the loop. The first case in this figure represents the fall-through condition, which must be included for any while loop or similar loop construct. For this case, 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 case represents a single iteration of the loop and is derived from the case for k=1 shown in Figure 15c. The condition for this case is

not(lec(0)) and lec(1).

The third case represents more than one iteration of the loop and is derived from the case for k>1. For this last case, the final iteration count, call it k_e, is the minimum k, such that the loop exit condition is true. Thus, the condition for this last case is

not(lec(0)) and not(lec(1)) and (k_e = min< k ¦ (k $\geq$ 1) and lec(k) >).

For this case, the symbolic values of the variables at exit from the loop are represented by V(k_e).

The loop expression is a closed form representation capturing the effects of the loop. Thus, the nodes in the loop can be replaced by a single node, annotated by this loop expression. If the loop body contains nodes n_i through n_j, this single node is denoted $n_{(i,j)}$. When a loop is encountered during symbolic evaluation, each case in the loop expression must be considered in the context of each case of each predecessor node.

Consider the evaluation of one case of the loop expression in the context of one case of a predecessor node. The results of this evaluation will be a single case for the evaluated loop node. The symbolic values of the variables of the predecessor case provide the symbolic values of the variables at entry to the loop. Thus, the case for the evaluated loop node is obtained by evaluating the loop expression case in terms of these symbolic values. The path condition of the evaluated loop node case is developed by interpreting the condition from the loop expression case and conjoining it to the path condition of the predecessor case. The symbolic values of the variables of the evaluated loop node case are developed by interpreting the assignments specified by the loop expression case.

The above process is repeated for each case in the loop expression with each case of each predecessor node. The resulting cases are then combined to form the case expression for the evaluated loop node. Global symbolic evaluation can proceed as usual from this point. Figure 16 demonstrates the global symbolic evaluation of TRAP. 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. Once again, the symbolic values of variables that cannot be modified are shown only at the start node. Note that node 8 is the only predecessor node to the loop and node (9,13) provides the case expression resulting from evaluation of the loop expression. The final output of global symbolic evaluation of TRAP would be a case expression, like that shown in for the final node in figure 16, except that only the cases with consistent path conditions and the the symbolic values for the two output parameters, AREA and ERROR, would be shown.

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. When there are simultaneous recurrence relations, several variables that may be dependent are modified within the loop. In particular, the dependence may by cyclic -- V may depend on W, which depends on V -- 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. Moreover, loops tend to cause an explosion in the size and complexity of the global representation of a routine. Nested loops exacerbate this problem. In addition, determining path condition consistency for a path condition

```
s,1     case
          true:
             A = a
             B = b
             N = n
             AREA = ?
             ERROR = ?
             H = ?
             X = ?
             YOLD = ?
             YNEW = ?
        endcase

5       case
           -- (s,1,3,4,5)
           (n >= 1):
             AREA = 0.0
             ERROR = false
             H = ?
             X = ?
             YOLD = ?
             YNEW = ?
        endcase

8       case
           -- (s,1,3,4,5,6,7,8)
           (n >= 1) and (a /= b)
         = (n >= 1) and (a - b /= 0.0):
             AREA = 0.0
             ERROR = false
             H = (b-a)/n = -a/n + b/n
             X = a
             YOLD = F(a)
             YNEW = ?
        endcase

(9-13)case
           -- (s,1,3,4,5,6,7,8,9)
           (n >= 1) and (a - b /= 0.0) and (-b + a >= 0.0)
         = (n >= 1) and (a - b > 0.0):
             AREA = 0.0
             ERROR = false
             H = -a/n + b/n
             X = a
             YOLD = F(a)
             YNEW = ?

           -- (s,1,3,4,5,6,7,8,9,10,11,12,13)
           (n >= 1) and (a - b /= 0.0) and
           (-b + a < 0.0) and (-b - a/n + b/n + a >= 0.0)
         = (n = 1) and (a - b < 0.0):
             AREA = 0.0 + F(a)/2.0 + F(-a/n+b/n+a)/2.0
                  = F(a)/2.0 + F(b)/2.0
             ERROR = false
             H = -a/n + b/n = -a + b
             X = -a/n + b/n + a = b
             YOLD = F(-a/n + b/r + a) = F(b)
             YNEW = F(-a/n + b/r + a) = F(b)
```

Figure 16. Global Symbolic Evaluation of TRAP

```
      -- (s,1,3,4,5,6,7,8,(9,10,11,12,13))
      (n >= 1) and (a - b /= 0.0) and
      (-b + a < 0.0) and (-b - a/n + b/n + a < 0.0) and
      (k_e = min< k| (k>1) and (-b + k*(-a/n+b/n) + a >= 0.0) >)
    = (n > 1) and (a - b < 0.0) and (k_e = n):
         AREA = 0.0 + F(a)/2.0 + F(k_e*(-a/n+b/n)+a)/2.0 +
                sum< i:=1..k_e-1 | F(i*(-a/n+b/n)+a) >
              = F(a)/2.0 + F(b)/2.0 +
            sum< i:=1..n-1 | F(a-a*i/n+b*i/n) >
         ERROR = false
         H = -a/n + b/n
         X = k_e*(-a/n+b/n) + a = b
         YOLD = F(k_e*(-a/n+b/n)+a) = F(b)
         YNEW = F(k_e*(-a/n+b/n)+a) = F(b)
   endcase


15 case
      -- (s,1,3,4,5,6,7,8,9,14,15)
      (n >= 1) and (a - b > 0.0):
         AREA = 0.0 * (-a/n + b/n) = 0.0
         ERROR = false
         H = -a/n + b/n
         X = a
         YOLD = F(a)
         YNEW = ?

      -- (s,1,3,4,5,6,7,8,9,10,11,12,13,14,15)
      (n = 1) and (a - b < 0.0):
         AREA = (F(a)/2.0 + F(b)/2.0) * (-a + b)
              = -a*F(a)/2.0 + b*F(a)/2.0 - a*F(b)/2.0 + b*F(b)/2.0
         ERROR = false
         H = -a + b
         X = b
         YOLD = F(b)
         YNEW = F(b)

      -- (s,1,3,4,5,6,7,8,(9,10,11,12,13),14,15)
      (n > 1) and (a - b < 0.0) and (k_e = n):
         AREA = (F(a)/2.0 + F(b)/2.0 + sum< i:=1..n-1 | F(a-a*i/n+b*i/n) >) *
                (-a/n + b/n)
              = -a*F(a)/2.0*n + b*F(a)/2.0*n - a*F(b)/2.0*n + b*F(b)/2.0*n -
                a*sum< i:=1..n-1 | F(a-a*i/n+b*i/n) >/n +
                b*sum< i:=1..n-1 | F(a-a*i/n+b*i/n) >/n
         ERROR = false
         H = -a/n + b/n
         X = b
         YOLD = F(b)
         YNEW = F(b)
   endcase
```

Figure 16. (continued)

```
f      case
          -- (s,1,3,4,5,6,7,8,9,14,15,16,f)
          (n >= 1) and (a - b > 0.0) and (a > b)
        = (n >= 1) and (a - b > 0.0):
             AREA = -0.0 = 0.0
             ERROR = false
             H =-a/n + b/n
             X = a
             YOLD = F(a)
             YNEW = ?

          -- (s,1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,f)
          (n = 1) and (a - b < 0.0) and (a > b)
        = false:

          -- (s,1,3,4,5,6,7,8,(9,10,11,12,13),14,15,16,f)
          (n > 1) and (a - b < 0.0) and (k_e = n) and (a > b)
        = false:

          -- (s,1,3,4,5,6,7,8,9,14,15,f)
          (n >= 1) and (a - b > 0.0) and (a <= b)
        = false:

          -- (s,1,3,4,5,6,7,8,9,10,11,12,13,14,15,f)
          (n = 1) and (a - b < 0.0) and (a <= b)
        = (n = 1) and (a - b < 0.0):
             AREA = -a*F(a)/2.0 + b*F(a)/2.0 - a*F(b)/2.0 + b*F(b)/2.0
             ERROR = false
             H = -a + b
             X = b
             YOLD = F(b)
             YNEW = F(b)

          -- (s,1,3,4,5,6,7,8,(9,10,11,12,13),14,15,f)
          (n > 1) and (a - b < 0.0) and (k_e = n) and (a <= b)
        = (n > 1) and (a - b < 0.0) and (k_e = n):
             AREA = -a*F(a)/2.0*n + b*F(a)/2.0*n - a*F(b)/2.0*n + b*F(b)/2.0*n -
                    a*sum< i:=1..n-1 | F(a-a*i/n+b*i/n) >/n +
                    b*sum< i:=1..n-1 | F(a-a*i/n+b*i/n) >/n
             ERROR = false
             H = -a/n + b/n
             X = b
             YOLD = F(b)
             YNEW = F(b)

          -- (s,1,3,4,5,f)
          (n >= 1) and (a = b)
        = (n >= 1) and (a - b = 0.0):
             AREA = 0.0
             ERROR = false
             H = ?
             X = ?
             YOLD = ?
             YNEW = ?

          -- (s,1,2,f)
          (n < 1):
             AREA = ?
             ERROR = true
             H = ?
             X = ?
             YOLD = ?
             YNEW = ?
       endcase
```

Figure 16. (continued)

incorporating a loop exit condition is even more complex than that discussed for symbolic execution. This is due to the possible representation of a final loop iteration count in terms of conditional expressions or a minimum value expression, or both. Deciding the existence of these minimum values is essentially proving routine termination. When none of these problems arise, the loop analysis technique employed by global symbolic evaluation provides a more general evaluation of a loop than the techniques employed by symbolic execution or dynamic evaluation systems, which only evaluate a loop for a specific number of iterations.

5.2 Applications

Global symbolic evaluation has several possible applications, many of which are similar to those of symbolic execution. The global representation provides a concise representation of a routine, which is often useful in detecting errors. In TRAP, for example, examination of the path condition and the path computation for the first case of the final node in figure 16 reveals a fairly subtle error. In implementing the loop, the incorrect assumption is made that A < B and this is the loop exit condition. This error is uncovered because global representation states that AREA = 0.0 when A < B, which is clearly incorrect. The routine could be corrected by changing the loop exit condition to

((A>B) and (A>B)) or ((A<B) and (A<B)).

Another natural application of global symbolic evaluation is test data generation, which could be performed by finding solutions for individual path conditions associated with the final cases in the global representation of a routine. New methods must be developed, however, for solving a path condition that contains expressions for the final loop iteration counts. Global symbolic evaluation could also be used to automatically generate and check error conditions as it analyzes a routine. Similarly, user-provided assertions can be checked for validity. With global symbolic evaluation, however, it may be possible to check the validity of these assertions for all paths rather than a specific path. If a routine is annotated with assertions that specify the intended function of the routine and these are shown to be valid, the correctness of the routine has been verified.

Global symbolic evaluation also has applications in program optimization [TOWN76]. As in optimizing compilers, the existence of a computation graph [COCK70] makes common subexpression elimination and constant folding relatively straightforward. In addition, several types of loop optimizations may often be performed when the loop expressions are obtainable. 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 manipulated in the second loop are not computed 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 partition analysis method [RICH81] uses global symbolic evaluation in comparing a routine with its specification. Global symbolic evaluation is applied to a routine as well as to its specification, thus creating global representations of both. By coalescing the resulting case expressions, a partition of the input domain is provided. An evaluation of this partition guides in the selection of test data that is based on information derived not only from the routine but from the specification as well. In addition, this partition is employed in verifying the routine's consistency with the specification.

6. CONCLUSION

Symbolic evaluation has several applications for program validation and testing. Since it is a relatively new method of program analysis, there are several unsolved problems and directions for future research. Initial studies of its effectiveness have only recently been conducted. This section describes the results from one such study and sets forth several areas for future research.

6.1 Effectiveness

The major use of symbolic evaluation is in the testing and analysis of programs. Howden has investigated the effectiveness of several program testing and analysis methods [HOWD78a]. Included in the methods were branch testing, path testing, and symbolic testing, each of which can be aided by symbolic evaluation. For twenty-eight errors occurring in six programs, the reliability of each method was determined.

A testing strategy that involves actual execution is reliable for an error only if every test data set that satisfies the criterion of that strategy is guaranteed to reveal the error. The statistics obtained in the study indicate that the path testing strategy was reliable for eighteen of the twenty-eight errors. Path testing involved the testing of every path, which generally is impractical since a routine may have an infinite number of paths. A strategy that approximates path testing was found to be reliable for twelve of the errors. This strategy required the execution of all paths with no more than two iterations of any loop. Branch testing was reliable for only six errors. These results indicate that the detection of errors is often dependent on testing particular combinations of branch predicates. Although the statement testing strategy was not analyzed in this study, experience has shown that this strategy is, in general, less effective than branch or path testing.

Symbolic testing is a method whereby the symbolic representations produced by symbolic evaluation are examined for errors. Symbolic testing is considered reliable for an error if the symbolic representations of the path computation and path condition reveal the presence of the error in an obvious way that would catch the attention of the programmer. Symbolic testing of the set of paths chosen to approximate path testing guaranteed the detection of seventeen of the twenty-eight errors.

Symbolic evaluation methods can be used to assist in all of the testing strategies mentioned above. All three methods of symbolic evaluation can be used to perform symbolic testing. In addition, symbolic execution can be used to generate test data to meet the criterion of statement, branch, or path testing. Dynamic symbolic evaluation does not actively generate test data but can monitor progress towards meeting the criterion of a testing strategy. The output produced by dynamic symbolic evaluation shows the results of both symbolic and actual testing. Global symbolic evaluation assists in symbolic testing for all program paths and also classifies paths in a way that could be useful in actual testing.

Howden's study found that combining both symbolic testing and actual testing was reliable for more errors than either method used alone. It is important to note that this study only considered a method reliable for an error if it guaranteed the detection of the error for every test data set that satisfies the testing criterion. If this requirement were relaxed and data sets were astutely selected based on the information in the symbolic representations, more errors would be detected. Moreover, the systematic application of testing strategies, such as domain testing and error sensitive testing, would increase the number of detected errors.

6.2 Other Considerations

Symbolic evaluation poses several unsolved problems and opens up several areas for future research. Two of these areas, loop analysis and path condition consistency determination, have been described previously. This section first addresses two enhancements to the symbolic evaluation methods described so far, input and output along a path and routine invocations. Next, several ongoing research efforts related to symbolic evaluation are discussed.

This paper has 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. Whenever an input statement is encountered along a path, symbolic names representing the input values are assigned to the input variables. The convention previously described for representing input values must be modified slightly, however, since input may occur more than once for a variable . To maintain the association between input values and variables, the symbolic names may be suffixed with an index notation when necessary. For example, if a variable, say AMOUNT, is assigned input values twice along a path, the first input value might be represented by amount.1 and the second by amount.2. Furthermore, the variables assigned input values as well as the number of inputs may vary from path to path, because different input statements may be encountered on different paths. Whenever an output statement is encountered on a path, the symbolic values of the output variables are provided. As with inputs, different output statements may be encountered along different paths, so the variables whose values are output, as well as the number of outputs, may vary from path to path. Moreover, for global symbolic evaluation, the number of inputs and outputs may depend on the final loop iteration counts for the routine. Input and output along a path requires no additional changes to the interpretive techniques previously described. The functional conceptualization of a routine, however, must allow for an arbitrary, and perhaps varying, number of inputs and outputs.

Routine invocation during symbolic execution and global symbolic evaluation is an area in need of further investigation. During symbolic execution, the most straightforward approach to routine invocation is similar to normal execution in that information is passed to and from the called routine via the parameters and a path through the called routine is executed. When a routine invocation is encountered, the symbolic values of the arguments are passed to the called routine. Using the implementation approach described for ATTEST, this merely involves passing a pointer into the computation graph for each argument and assigning this pointer to the appropriate parameter. This graph is independent of any routine since it only references constants and symbolic names. Any branch predicates that are interpreted within the called routine are conjoined to the path condition 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, a pointer into the computation graph for each parameter is returned and assigned to its corresponding argument.

The problem with the above approach is that it requires interpretation of the statements on some path through a routine each time that routine is invoked. A more modular approach, called subroutine substitution, utilizes previously created symbolic representations of a routine. With such an approach, the path computation and path condition resulting from the symbolic execution of a path are saved for substitution. Later, when the routine is invoked, the symbolic representations of the arguments are substituted for the symbolic names that were assigned to the parameters in the saved path computation and path condition of the called routine. The updated path condition of the called routine is conjoined to the existing path condition 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 path condition following return from the called routine. In addition, the symbolic values of the parameters, which comprise the path computation of the called routine, are returned to the calling routine.

This approach to routine invocation involves expensive reformulation and simplification of the symbolic representations and may not always be more efficient than reevaluation of the path [WOOD80]. When arguments are large arrays or functions, there are additional problems. This approach also assumes a bottom-up testing environment, where routines must be tested before those which call them. Moreover, several evaluations of the called routine must be saved to make this a viable approach.

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 intended path conditions and the associated path computations. This specification could then be substituted as described for subroutine substitution. Such an approach allows for top-down testing and incremental development of software.

Global symbolic evaluation may similarly utilize two alternative approaches to routine invocation, continued evaluation at each invocation or routine substitution of a global representation. The first approach continues the creation of the global representation throughout the called routine. The symbolic representations of the path computation and path condition are passed from each case expression at the point of invocation to the called routine and back. This can be done in a way similar to the approach described for symbolic execution.

The second approach to routine invocation during global symbolic evaluation substitutes the global representation of the called routine into the global representation of the calling routine. The representation of the called routine may be either the results from a previous global symbolic evaluation or a user-supplied specification of the routine. Each case expression of the called routine must be evaluated in the context of each case expression of the calling routine at the point of invocation. For each such combination, this evaluation is similar to routine substitution during symbolic execution. Since the number of such combinations may explode, the need for efficient techniques is paramount.

Another area of current research is array element determination. A problem occurs 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. Although an indeterminate array element can be represented symbolically, determining path condition consistency becomes extremely complicated when such an occurrence affects the path condition. This problem occurs frequently during both symbolic execution and global symbolic evaluation. (It can not occur during dynmaic symbolic evaluation since all values, including subscript values, are known.) Inefficient solutions exist, for in the worst case all possible subscript values can be enumerated. Though there has been some work on this problem [BOYE75,CLAR76a,RAMA76], the results are still unsatisfactory. Efficient solutions requiring a minimal amount of backtracking are still being explored.

General problems of efficiency plague all three symbolic evaluation methods. These methods have only been implemented in experimental systems; more efficient implementations must be explored. Osterweil [OSTE81] 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 symbolic evaluation systems. Osterweil emphasizes the need for integrating analysis methods so that each will be used where it is most effective and so that the information gathered by one method can be used to enhance another. The coordination of data flow analysis and symbolic evaluation is an area where this integration may prove fruitful. Data flow analysis methods can be used to detect paths containing suspect sequences of events but cannot determine if these paths are executable.

Symbolic execution can decide this by determining path condition consistency. Both analysis methods are strengthened by this pairing. Data flow analysis would no longer report suspect conditions about nonexecutable paths, thus decreasing extraneous information, which only dilutes its effectiveness. In addition, suspect conditions on executable paths could now be reported as errors. Symbolic execution would benefit in that it would be directed to suspect paths, thus increasing its effectiveness for detecting program errors. Osterweil describes several other interesting prospects for integrating symbolic evaluation with data flow analysis.

In this paper we have focused on the analysis of the code. Future directions of software validation will be concerned with all stages of software development. As work progresses in the areas of requirements, specifications, and design, analysis of these stages will also be considered. Symbolic evaluation methods, which provide an alternative representation of a routine, should prove useful during these earlier stages of software development [CHEA79b]. Incorporating the analysis of both the routine and a specification of its intended function to determine test data has been proposed [GOOD75,WEYU80]. The partition analysis method [RICH81] applies symbolic evaluation techniques to a routine's specification and implementation to form a partition of the input domain. Domain testing and error sensitive testing strategies are then applied to this partition to guide in the selection of test data. There are several ongoing projects in which the application of symbolic evaluation to pre-implementation stages is under investigation.

6.3 Summary

In this paper, three methods of symbolic evaluation have been described. All three methods represent the computation and domain by symbolic expressions in terms of the input values, although the methods differ in their scope of representation.

Dynamic symbolic evaluation is the most restrictive method of the three. Using input data to determine a path, dynamic symbolic evaluation represents the path computation. Though the path condition can be described symbolically, there is no need to determine its consistency. With neither simplification of the path condition nor determination of path condition consistency necessary, the implementation of dynamic symbolic evaluation is straightforward. The major application of this method is program debugging.

Symbolic execution systems do not depend on input values to determine the path, as do dynamic symbolic evaluation systems, but rather analyze a specified path. Symbolic execution systems represent the path computation and the path condition. Since many paths are not executable, some symbolic execution systems try to determine path condition consistency. The most efficient implementation approach is to use the forward expansion technique and to determine path condition consistency whenever an evaluated branch predicate is conjoined to the existing path condition. In general, path condition consistency can not always be determined; in practice, consistency can often be determined using any of several existing techniques. In addition, there is work currently being done on improving methods of solving arbitrary systems of constraints. There are several interesting applications of symbolic execution in the area of program validation, including automatic error detection and test data generation.

Global symbolic evaluation has the widest scope of analysis; it attempts to functionally represent the total routine by a symbolic expression. Since there may be an infinite number of paths in a routine, this method requires more sophisticated analysis than the mere combintation of the symbolic representations for each path. A loop analysis technique is used, which attempts to represent each loop in a closed form that is dependent on a final loop iteration count. While this approach can successfully analyze several types of loops, additional work is needed in this area. By using a closed form representation for each loop, the computation and domain for a class of paths can be represented. Each such representation is one case in the global representation provided by global symbolic evaluation. The determination of path condition consistency, which must be done for each case, is further complicated by the classification of loops. This is another area in need of further research. Global symbolic evaluation has prospective applications in the areas of program validation, program optimization, and the pre-implementation stages of software development.

Dynamic symbolic evaluation is a well-understood process that has been implemented in at least two dynamic testing systems. Symbolic execution has also been successfully implemented though there are still several implementation problems to be examined, as well as several areas of research to be explored. Global symbolic evaluation is a relatively new method and its future applicability will most likely depend on its success in loop analysis and consistency determination.

FOOTNOTES

[1] A case in the case expression used by global symbolic evaluation 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.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge Tom Cheatham and Dick Fairley for valuable discussions about their symbolic evaluation systems.

REFERENCES

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

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

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

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

CHEA79a T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs", IEEE Transactions on Software Engineering, SE-5, 4, July 1979, 402-417.

CHEA79b 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, 53-62.

CLAR76a L.A. Clarke, "Test Data Generation and Symbolic Execution of Programs as an Aid to Program Validation", Ph.D. Dissertation, University of Colorado, 1976.

CLAR76b L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, SE-2, 3, September 1976, 215-222.

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

COCK70 J. Cocke and J.T. Schwartz, Programming Languages and Their Compilers, New York University, Courant Institute of Mathematical Science, April 1970.

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

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

DILL81 L.K. Dillon, "Constraint Management in the ATTEST System," Department of Computer and Information Science, University of Massachusetts, Technical Report 81-9, May 1981.

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

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

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

GOOD75 J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, SE-1. 2. June 1975, 156-173.

HASS80 J. Hassell, L.A. Clarke, and D.J. Richardson, "A Close Look at Domain Testing," Department of Computer and Information Science, University of Massachusetts, Technical Report 80-16, October 1980.

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

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

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

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

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

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

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

HUAN78 J.C. Huang, "Program Instrumentation and Software Testing", Computer, 11, 4, April 1978, 25-32.

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

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

MILL75 E.F. Miller and R.A. Melton, "Automated Generation of Test Cast Data-Sets", Proceedings of the International Conference on Reliable Software, April 1975, 51-58.

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

OSTE81 L.J. Osterweil, "Software Engineering", Program Flow Analysis: Theory and Applications, Prentice Hall, Englewood Cliffs, New Jersey, 1981.

PLOE79 E. Ploedereder, "Pragmatic Techniques for Program Analysis and Verification," Proceedings of the 4th International Conference of Software Engineering, September 1979, 63-72.

RAMA76 C.V. Ramamorthy, S.F. Ho, and W.T. Chen, "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering, SE-2, 4, December 1976, 293-300.

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

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

STUC73 L.G. Stucki, "Automatic Generation of Self-Metric Software", Rec. 1973 Symposium on Software Reliability, April 1973, 94-100.

TOWN76 J.A. Townley, "The Harvard Program Manipulation System", Center for Research in Computing Technology, Harvard University, TR-23-76, 1976.

VOGE80 U. Voges, L. Gmeiner, and A. Amschler von Mayrhauser, "SADAT - An Automated Testing Tool," IEEE Transactions on Software Engineering, SE-6, 3, May 1980, 286-290.

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

WHIT80 L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, SE-6, 3, May 1980, 247-257.

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

ZEIL81 S.J. Zeil and L.J. White, "Sufficient Test Sets for Path Analysis Testing Strategies," Proceedings of the 5th International Conference on Software Engineering, March 1981, 184-191.