Symbolic Evaluation Methods

Lori A. Clarke
Debra J. Richardson

COINS Technical Report 81-8
May 1981

(revision of COINS Technical Report 79-20)

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

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.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: University of Massachusetts COINS Technical Report TR81-8
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): Symbolic Evaluation Methods
5. TYPE OF REPORT & PERIOD COVERED: Final
6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): Lori A. Clarke, Debra J. Richardson
8. CONTRACT OR GRANT NUMBER(s): AFOSR 77-3287, NSF MCS 77-02101

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer and Information Science Department, University of Massachusetts, Amherst, Massachusetts 01003
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research, Washington, D.C.
12. REPORT DATE: October 1981
13. NUMBER OF PAGES: 81

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Symbolic Evaluation, Validation, Program Testing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes symbolic evaluation, a program analysis method that concisely represents a program's computations and 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 method 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

20.
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 interesting applications for program validation and optimization.

For each method, implementation approaches and applications are described and examples are given. In addition, the status of current research related to symbolic evaluation is discussed.

CONTENTS

ABSTRACT

This paper describes symbolic evaluation, a program analysis method that concisely represents a program's computations and 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 method 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 interesting applications for program validation and optimization.

For each method, implementation approaches and applications are described and examples are given. In addition, the status of current research related to symbolic evaluation is discussed

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. These tools detect errors, determine program consistency, and generally increase confidence in a program. Several of these validation tools employ a method, called symbolic evaluation, that creates a symbolic representation of the program. Symbolic evaluation 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 which these numeric values were derived, whereas symbolic evaluation preserves this information. Symbolic evaluation has a wide range of validation applications, including testing, debugging, and 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 first introduces the basic concepts of symbolic evaluation as well as some terminology that is used to describe the three methods. More work has been done in the area of symbolic execution, so a more detailed description of symbolic execution is given. 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. The description of symbolic evaluation is presented for readers unfamiliar with program validation methods, although the comparison of these methods is novel and may be of interest to those knowledgeable on the subject.

2. BASIC CONCEPTS

This section presents some concepts fundamental to symbolic evaluation. Some terminology is introduced and an interpretive technique that may be 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.

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

Program analysis methods typically represent a routine by a directed graph, called a control flow graph, that describes the possible flow of control through the routine. The nodes in the graph, $\{n_1, n_2, \ldots, n_q\}$, represent executable statements. Note that in Figure 1, the statements in TRANSACT are annotated with node numbers. An 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

```
      procedure TRANSACT (DAYS:in integer;
                          AMOUNT:in real;
                          BALANCE:in out real;
                          INTEREST:out real;
                          BELOWMIN:out boolean;
                          OVERDRAFT:out boolean) is

      -- TRANSACT computes the INTEREST and new BALANCE for an
      -- interest-bearing checking account when a transaction
      -- of AMOUNT dollars is made.  The INTEREST is computed
      -- based on the number of DAYS since the last transaction.
      -- If the transaction causes an overdraft, then it is
      -- denied, an overdraft charge is made and OVERDRAFT is
      -- returned true.  If the transaction causes the new
      -- BALANCE to fall below the minimum balance, then a below
      -- minimum charge is made and BELOWMIN is returned true.

          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

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 transfers control to either node n_j or 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,

$$bp(n_i, n_j) = not(bp(n_i, n_k)).$$

In TRANSACT, for example, node 8 precedes nodes 9 and 11 and

$$bp(8,9) = (NEWBAL + AMOUNT < 0.0),$$

$$bp(8,11) = (NEWBAL + AMOUNT >= 0.0).$$

Note that each IF statement, nested or otherwise, forms a pair of complementary branch predicates. Some conditional statements, such as the FORTRAN computed GO TO or the Ada CASE statements, may have more than two successor nodes and each branch predicate must be represented appropriately. To facilitate analysis, the control flow graph has a single entry point, the start node n_s, and a single exit point, the final node n_f. Without loss of generality, 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. Figure 2 shows the control flow graph for TRANSACT.

A subpath in a control flow graph is a sequence of statements, $(n_{H_i}, n_{H_{i+1}}, \ldots, n_{H_t})$, where for all j, $i \leq j < t$, n_{H_j} is a node in the control flow graph such that there exists a transfer of control from n_{H_j} to $n_{H_{j+1}}$. A partial path is a subpath that begins with the start node and is denoted by P_{H_u}, where $P_{H_u} = (n_s, n_{H_1}, n_{H_2}, \ldots, n_{H_u})$.

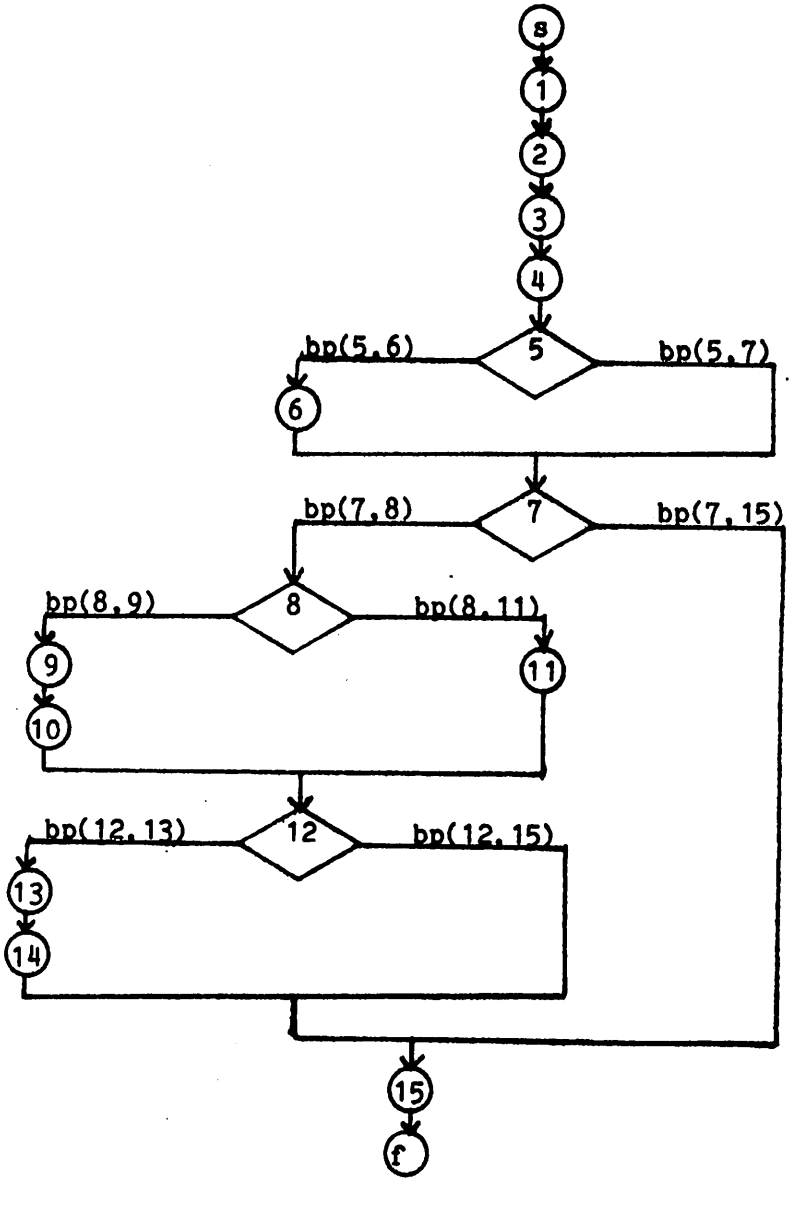

Figure 2. Control Flow Graph for TRANSACT

Hence, for any partial path P_{H_u} with $u \geq 1$, $P_{H_u} = (P_{H_{u-1}}, n_{H_u})$, where $P_{H_0} = (n_s)$. A path is a partial path that ends with the final node and is denoted by P_H, thus $P_H = (n_s, n_{H_1}, n_{H_2}, \ldots, n_{H_v}, n_f)$. 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.

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.

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

Symbolic evaluation provides symbolic 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. An overview of an interpretive technique that can be used to develop these symbolic values follows.

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 their 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 on 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 to 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.

Evaluation of a path provides symbolic representations of the path computation and path domain. The symbolic representation of the path computation consists of the symbolic values of the output parameters. These symbolic values are referred to as path values and denoted by PV. The symbolic representation of the path domain is provided by the conjunction of the symbolic values of the branch predicates. This conjunction is called the path condition and is denoted by PC. Note that only the input values that satisfy the PC could cause execution of the path. When it is necessary to specify a particular path, say P_H, the notation $PV[P_H]$ and $PC[P_H]$ will be used. In TRANSACT, the

statement or edge	interpreted branch predicates	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
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

output parameters are BALANCE, INTEREST, BELOWMIN, and OVERDRAFT, and thus

$$PV = (s(BALANCE), s(INTEREST), s(BELOWMIN), s(OVERDRAFT)).$$

For path P_2 in TRANSACT,

$$PC[P_2] = s(bp(5,7)) \text{ and } s(bp(7,8)) \text{ and } s(bp(8,11)) \text{ and } s(bp(12,13)).$$

Figure 5 shows the PV and PC resulting from symbolic evaluation of path P_2.

Each of the three methods of symbolic evaluation can use an interpretive technique similar to that described above to develop symbolic representations of the path computations and path domains. Symbolic execution is a path-dependent method of symbolic evaluation that provides the PV and PC for 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 numeric output values that result from the execution of a path, dynamic symbolic evaluation provides the PV and PC. 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 numeric output values

(BALANCE=90.23, INTEREST=0.33,
BELOWMIN=true, OVERDRAFT=false),

```
Path Values PV[P2]
    BALANCE = balance*(1+0.06/365)**days + amount - 0.1
    INTEREST = balance*(1+0.06/365)**days - balance
    BELOWMIN = true
    OVERDRAFT = false

Path Condition PC[P2]
    (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 Values and Path Condition for Path P_2 in TRANSACT

as well as the PV and PC 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* 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 analyzed. For such a routine the case expression consists of a case for each path, where each case consists of a PC and the associated PV. 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 their number of loop iterations to be grouped as a class of paths. Thus, in general, a case consists of the PC for a class of paths and the PV associated with that class.

Although the symbolic representations provided by each of the three methods of symbolic evaluation are similiar, the information that is gathered to achieve these representations differs significantly and thus affects the types of program analysis that can be performed. Dynamic

---

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

```
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

symbolic evaluation maintains only the information required to develop the final symbolic representations 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 and error detection. Global symbolic evaluation analyzes all paths and maintains a global representation of a routine and thereby has applications to program optimization and verification in addition to the applications of symbolic execution. 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.

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 PC 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 PC. 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 develops the symbolic representations 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 common subexpression elimination techniques used by some optimizing compilers [COCK70].

After evaluation of a path, the PV is obtained by traversing the binary expression trees for the output parameters. The PC is created by traversing the binary expression trees for the interpreted branch predicates and conjoining the resulting symbolic values. In the purest sense, the PC and PV 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 PC.

Simplification can be done by converting the PC and PV into canonical forms. There are several available algebraic manipulation systems [BOGE75,BROW73,RICH78] that can be used to accomplish this simplification. A canonical form for the symbolic value of each output parameter in the PV might be

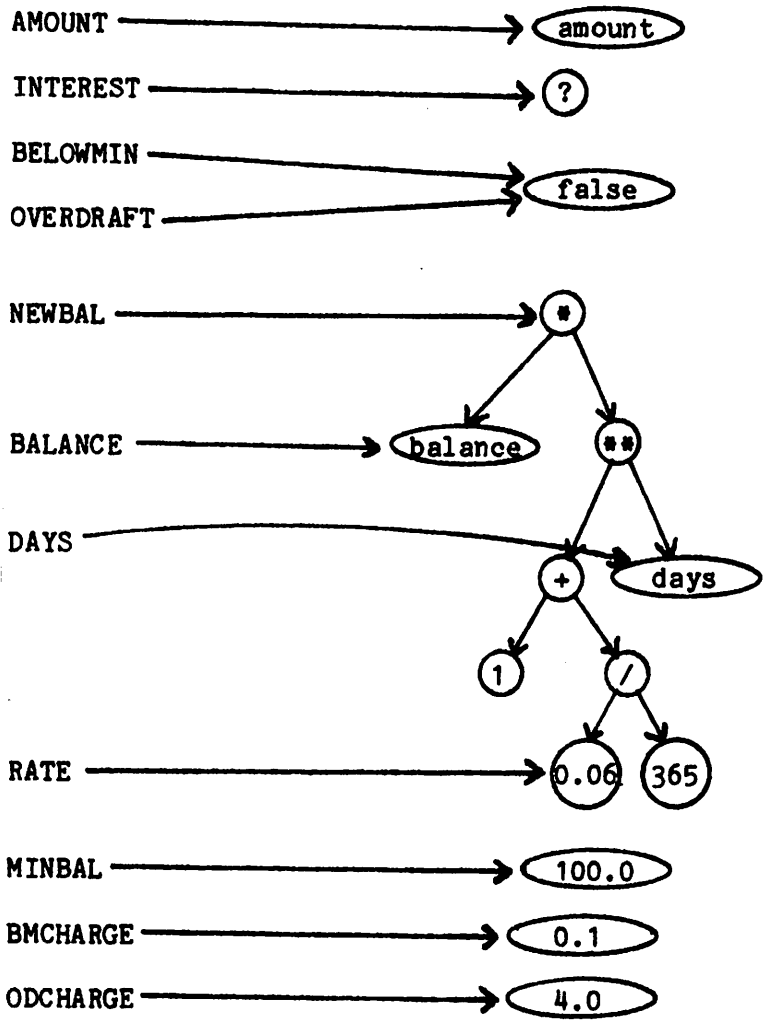

Figure 7a. Computation Graph after Interpretation of Statements s,1,2,3 in TRANSACT

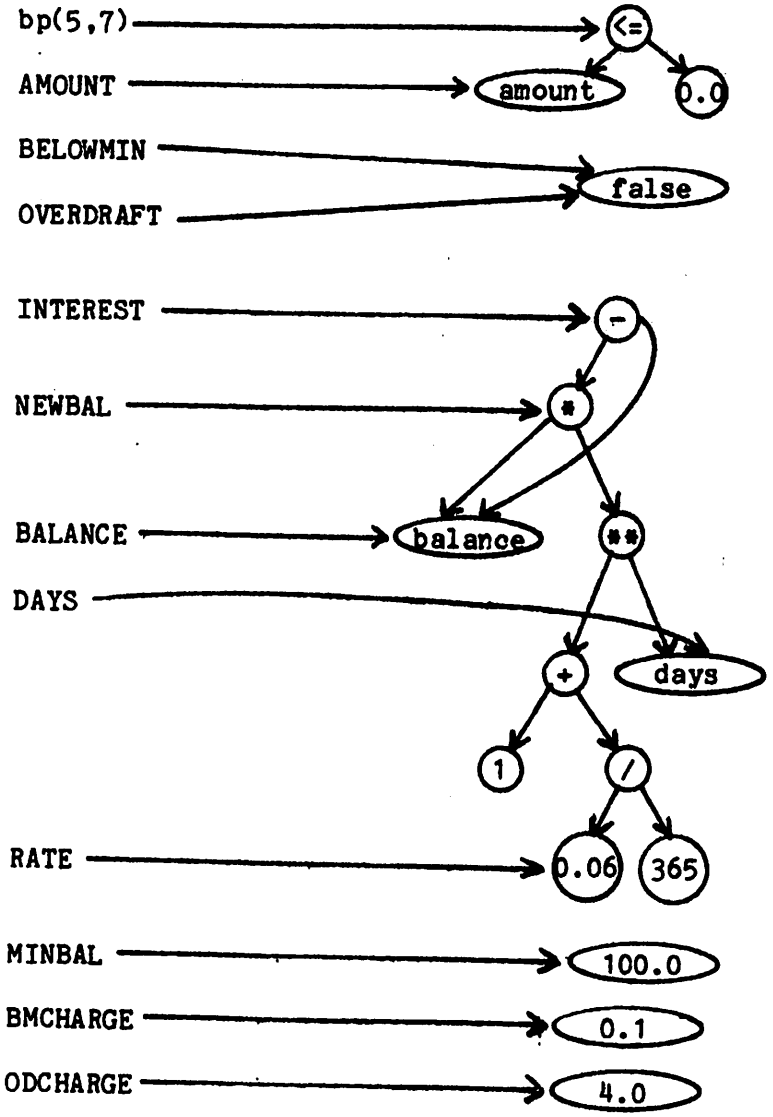

Figure 7b. Computation Graph after Interpretation of Statements s,1,2,3,4,5,7 of TRANSACT

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 might be

amount + 1.00016**days*balance - 0.1.

The PC might be put into conjunctive normal form and each relational expression put into a canonical form. This canonical form might be one in which the constant term is on the right-hand-side of the relational operator and, on the left-hand-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 PC is consistent. One approach to this problem employs a theorem proving system. We refer to this as the axiomatic technique since it is based upon the axioms of predicate calculus. Another approach, referred to as the algebraic technique, treats the PC as a system of constraints and uses one of several algebraic methods -- such as a gradient hill-climbing or a linear programming algorithm -- to solve this system of constraints. The ATTEST system uses a linear programming algorithm [LAND73] and thus employs the algebraic technique. The advantage of choosing this technique is that a solution is provided when the PC is determined to be consistent. This solution 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, PC 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 PC as each branch predicate is interpreted. ATTEST develops the PC as the statements on a path are interpreted. Thus at any point in this interpretation, there is a symbolic representation of the domain for the partial path that has been evaluated so far. For partial path $P_{H_u} = (n_s, n_{H_1}, \ldots, n_{H_u})$, the path condition is denoted $PC[P_{H_u}]$. When a node $n_{H_{u+1}}$ is considered as an extension to the partial path P_{H_u}, the interpreted branch predicate $s(bp(n_{H_u}, n_{H_{u+1}}))$ is first simplified and then examined for consistency with $PC[P_{H_u}]$. Unless inconsistency is determined, the interpreted branch predicate is conjoined to $PC[P_{H_u}]$, creating

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

Consistency or inconsistency may possibly be determined by performing simple reductions [DEUT73,DILL81] on the PC. On the one hand, it may be possible to determine that $s(bp(n_{H_u}, n_{H_{u+1}})$ is dominated by relational expressions in $PC[P_{H_u}]$, in which case $PC[P_{H_{u+1}}]$ must be consistent, since

$PC[P_{H_u}]$ 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), thus PC[s,1,2,3,4,5,6,7,15] is consistent. On the other hand, $s(bp(n_{H_u}, n_{H_{u+1}}))$ may be contradicted by a relational expression in $PC[P_{H_u}]$, in which case $PC[P_{H_{u+1}}]$ 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), 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 costly techniques, such as the axiomatic or algebraic techniques mentioned above.

In addition to detecting nonexecutable paths early in the symbolic execution process, the incremental development of the PC 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 PC 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 alternative edge (7,15), and analysis continues.

statement or edge	simplified, evolving PC	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 an Inconsistent PC in TRANSACT and Continuation with Executable Path P_9

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 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 PC. 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. 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. The graph shown in Figure 9 demonstrates all three cases. 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.

Thus far symbolic execution has been described in terms of the forward expansion technique. Backward substitution [HOWD75,HUAN75] is an alternative technique that has been proposed for systems concerned with creating only the PC and not the PV. While the forward expansion technique begins

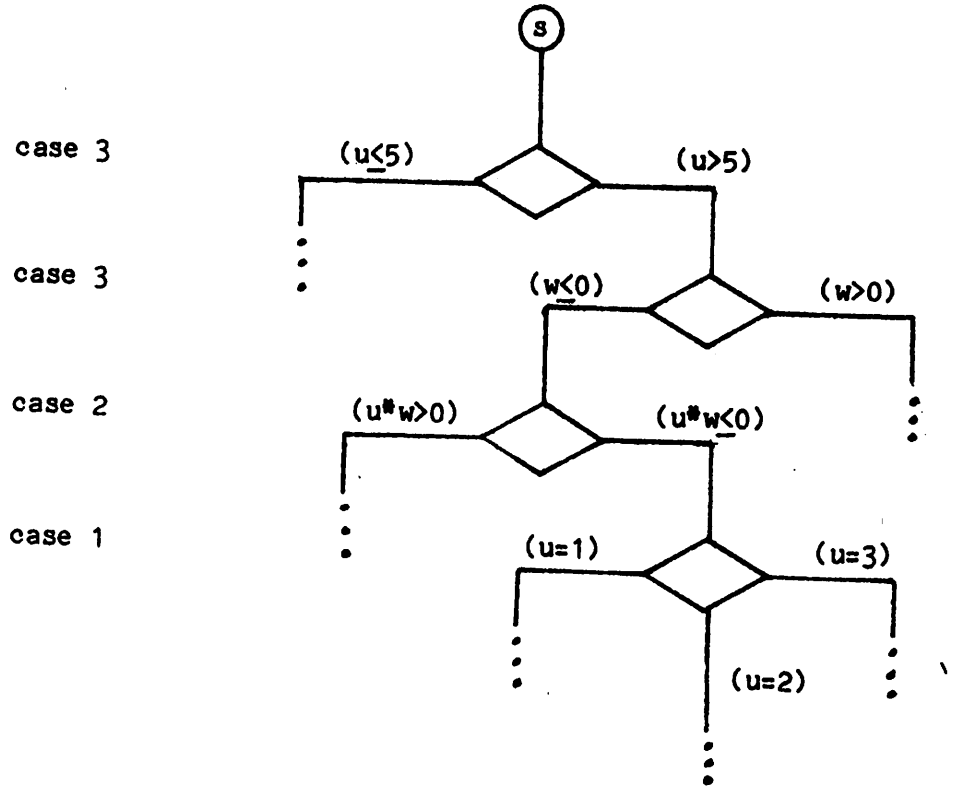

Figure 9. Examples of the 3 Cases that Can Occur During Consistency Determination

with the start node and works toward the final node, the backward substitution technique begins with the final node and works toward the start node. During backward substitution, 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 PC 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 when only the PC is desired, many of the assignment statements, specifically those that do not modify variables referenced in the recorded branch predicates, can be ignored. 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 PC and PV are desired, the two approaches interpret each statement and produce equivalent symbolic representations.

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+0.06/365)**days amount < 0.0 amount <= 0.0

Figure 10. Backward Substitution for Path P_2 in TRANSACT

Forward expansion is a more efficient technique of symbolic execution than backward substitution when early detection of nonexecutable paths is supported. When a branch predicate is encountered during forward expansion, it can be interpreted, simplified, and checked for consistency with the existing PC. In backward substitution, the recorded branch predicates can be conjoined and treated as the evolving PC of the subpath. To check for consistency when a branch predicate is encountered, a process similar to that used by forward expansion could be employed. Additional processing of the PC, however, must be done whenever an assignment is made to any variable referenced in the PC. When this occurs, each modified branch predicate must be resimplified and PC consistency determined. In Figure 10, note that at every modification point, resimplification and PC consistency determination is required. This additional processing during backward substitution is costly. Further, when the PC is found to be inconsistent, there is no efficient approach to salvaging the subpath that has been evaluated so far. Since an assignment may modify any of the recorded branch predicates, a branch predicate recorded much earlier on the subpath may become inconsistent. Finding the source (or sources) of an inconsistent PC and backing up to an appropriate point in the evaluation is usually not cost effective. The control flow graph in Figure 11 illustrates this problem. When A is assigned the value 10, the recorded branch predicate (A > 0) becomes inconsistent. Since node 20 is the only node on the

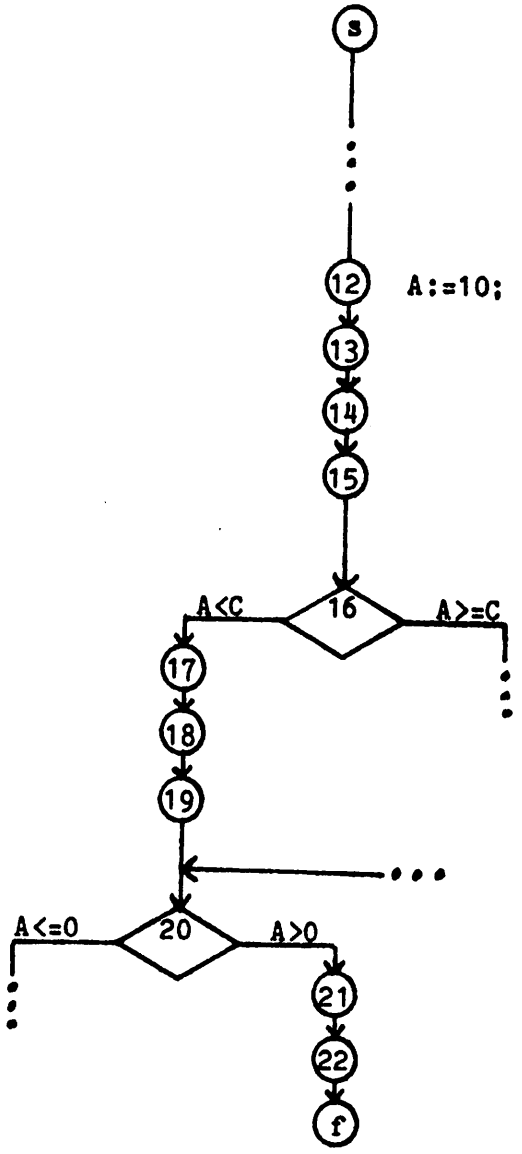

Figure 11. Detection of an Inconsistent PC during Backward Substitution

subpath with more than one predecessor node, the evaluation of all assignments and branch predicates for nodes 12 through 19 must be removed if any of the evaluated subpath is to be salvaged. A similar problem occurs when a branch predicate is recorded and the evolving PC becomes inconsistent. In general, to salvage any of the evaluation once an inconsistency in the evolving PC is detected requires the backward substitution technique to undo all the assignments and branch predicates up to the most recently evaluated node with more than one predecessor node. Because of this excessive overhead, the backward substitution technique should discard a subpath with an inconsistent PC and start anew. Thus, the inability to efficiently salvage part of the evaluation of a subpath with an inconsistent PC, as well as the additional processing required to determine PC consistency, makes the backward substitution technique less desirable than the forward expansion technique.

3.2. Applications

Symbolic execution systems have several interesting program validation applications. This section considers three applications: certification 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 certification and documentation. The PV 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 numeric output values to be correct while the path 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 symbolic value for BALANCE after execution of 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 = 1.0, 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 PV as well as the PC 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 certifying the PV and the PC 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 PC 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 potential 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 PC. If the resulting PC is consistent, then input values exist that would cause a division by zero error and an error report is issued. If the resulting PC is inconsistent, then this potential run-time error could not occur on this path. After checking for consistency, the expression for the error condition is removed from the PC before symbolic execution continues.

Path verification of assertions is another method of error detection. Instead of predefining the error conditions, user-created assertions define conditions that should be true at designated points in the routine. An error exists if an assertion is not true for all elements of the path domain. When an assertion is encountered during symbolic execution, the complement of the assertion is

interpreted and conjoined to the PC. Inconsistency of the resulting PC implies that the assertion is valid for the path, while consistency implies that the assertion is invalid for the routine.

Test data generation is another natural application of symbolic execution. The PC can be 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 PV 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 run-time performance characteristics. SELECT [BOYE75] and ATTEST are two symbolic execution systems that attempt to generate test data. Since an actual solution to the PC is desired and not just PC consistency, these two systems employ an algebraic technique.

The symbolic representations provided by symbolic execution can also be used to guide in astutely selecting test data. Error sensitive testing [FOST80,MEYE79,WEYU80] examines the statements or intent of a routine and selects test data to detect likely errors. Error sensitive testing is enhanced by considering the symbolic representations created by symbolic execution. This strategy selects data

for which the PC and PV appear sensitive to errors. Functional testing [HOWD80] further modifies this strategy by 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 ideas 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]. This polynomial testing strategy and testing strategies that focus on the selection of computationally sensitive data values are collectively referred to as computational testing. 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 test data selection strategy is called domain testing. A recent extension to domain testing 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 be either chosen by the user or 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 PC and PV can be observed. More extensive program coverage can be expedited by an automated path selection facility for choosing a set of paths based on some criterion, which is dependent on the intended application of symbolic execution.

Three criteria for selecting paths 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: the EFFIGY system [KING76] puts an arbitrary bound on the number of loop iterations; an approximate path coverage criterion proposed by Howden requires 0, 1, and 2 iterations of all loops; and 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 approach to 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 some 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, including a possible implementation approach and primary applications.

4.1. Implementation Approach

Dynamic testing systems monitor a routine's behavior during execution creating 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 [HUAN78], is generally done by a preprocessor. 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 numeric 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 would be augmented, however, to include the numeric value computed for each node.

After executing path P_H, the symbolic value for each output parameter is shown, providing $PV[P_H]$. With dynamic symbolic evaluation, these expressions generally are displayed as trees instead of as algebraic expressions, although both or either form could be displayed. The computation trees that would be created for the specified input values to TRANSACT are shown in Figure 12. Note that these input values cause path P_2 to be executed.

Existing dynamic symbolic evaluation systems are only concerned with providing the PV. Since the input values are known, each interpreted branch predicate evaluates to the constant value true (or a run-time error is encountered). The PC is, therefore, equal to true and thus it is not necessary to check for PC consistency. It would be easy to extend dynamic symbolic evaluation to provide a symbolic representation of the path domain. Note that the numeric value is known for each output parameter, but the symbolic

Input Variable	Symbolic Value	Numeric Value
DAYS	days	20
AMOUNT	amount	-10.00
BALANCE	balance	100.00

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

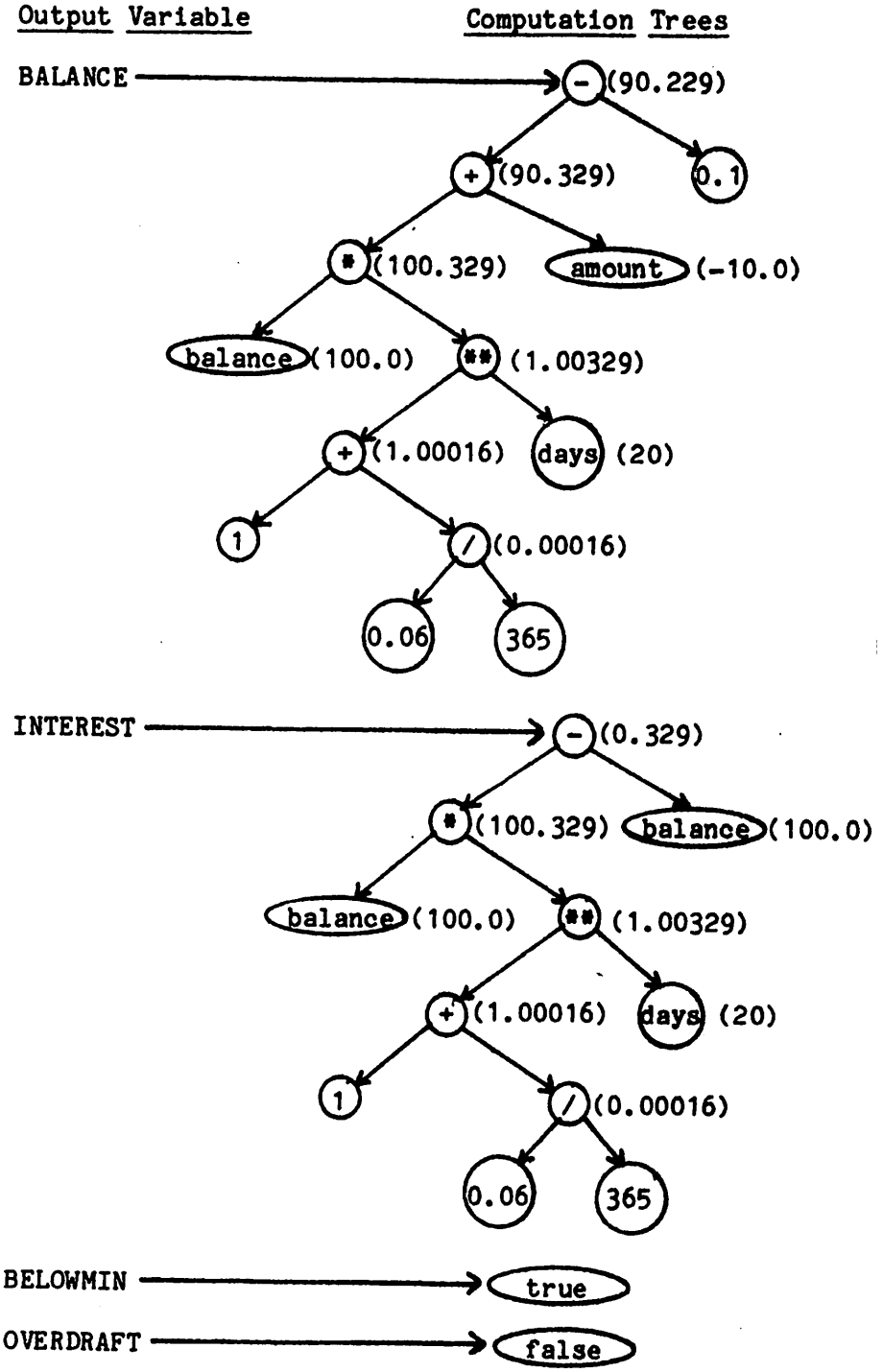

Figure 12. Final Results of Dynamic Symbolic Evaluation for One Test Run of TRANSACT

representation of the path computation is still provided. Examination of the PC, like examination of the PV, is helpful in uncovering errors in the routine. An erroneous PC would imply an erroneous branch predicate or erroneous calculation affecting a branch predicate. Thus, providing the PC would be a useful and natural enhancement to dynamic symbolic evaluation.

4.2. Applications

The primary application of dynamic testing systems is program debugging. When an error is uncovered in a routine, dynamic symbolic evaluation provides 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 symbolic 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 testing 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 PC 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 symbolic representation of the domain and computation for all paths, rather than along a specific path. This representation is achieved 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 PC for a partial path, as well as the symbolic values of all the variables computed along that partial path.

To see how a node is interpreted, consider a particular node 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 predecessor 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 predecessor case. This interpreted branch predicate is then conjoined to the PC for the associated partial path. As with symbolic execution, it is desirable to check the consistency of the PC. For routines without loops, the techniques described for symbolic execution could be applied. If the PC is found to be inconsistent, this case is discarded. Otherwise, the statement at node n_k must be interpreted in the context of the predecessor 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

```
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

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
```

Figure 13. Global Symbolic Evaluation of TRANSACT

```
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

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
```

Figure 13. (continued)

```
    -- (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)

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. The branch predicate bp(7,8) = (-AMOUNT > NEWBAL) is interpreted in terms of the symbolic values of the first case for node 7, providing one case for node 8. This branch predicate 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 PC associated with the first case. The resulting inconsistent PC 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, a 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. These symbolic values are in terms of the final iteration count and the symbolic values of the

variables at entry to the loop. The routine TRAP, shown in Figure 14, contains a loop, and Figure 15 shows the results from loop analysis. Note that TRAP invokes a routine F with one parameter. For the discussion that follows, each invocation of F will be represented symbolically as F(s(X)), where s(X) is the symbolic value of the argument X at the point of invocation. Alternative approaches for handling routine invocations are discussed later. The global symbolic evaluation of TRAP is 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* 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 each variable y_I, y_{I_0} represents the value of the variable y_I on entry to the first iteration of the loop and y_{I_k}, k>=1, represents the value of the variable y_I after execution of the kth iteration of the loop. The body of the loop is then symbolically evaluated to get a representation of a typical iteration. This evaluation, suppose it is for the kth iteration, is identical to the normal global symbolic evaluation process for a loop-free routine, except that the

---

* Only single-entry, single-exit loops are considered here.

```
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.
-- ERROR is true 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

```
case
   true:
      AREA_k = AREA_k-1 + (YOLD_k-1 + F(H_k-1 + X_k-1)) / 2.0
             = AREA_k-1 + F(H_k-1 + X_k-1)/2.0 + YOLD_k-1/2.0
      ERROR_k = ERROR_k-1
      H_k = H_k-1
      X_k = X_k-1 + H_k-1 = H_k-1 + X_k-1
      YOLD_k = F(H_k-1 + X_k-1)
      YNEW_k = F(H_k-1 + X_k-1)
      lec_k = not (H_k-1 + X_k-1 < B) = (-B + H_k-1 + X_k-1 >= 0.0)
endcase
```

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

```
case
   true:
      AREA(k) = AREA_0 + F(H_0+X_0)/2.0 + YOLD_0/2.0 +
                sum< i:=2..k | F((i-1)*H_0+X_0)/2.0 + F(i*H_0+X_0)/2.0 >
              = AREA_0 + F(k*H_0+X_0)/2.0 + YOLD_0/2.0 +
                sum< i:=1..k-1 | F(i*H_0+X_0) >
      ERROR(k) = ERROR_0
      H(k) = H_0
      X(k) = k*H_0 + X_0
      YOLD(k) = F(k*H_0+X_0)
      YNEW(k) = F(k*H_0+X_0)
      lec(k) = (-B + k*H_0 + X_0 >= 0.0)
endcase
```

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

```
case
   -- fall through
   (-B + X_0 >= 0.0):
      ERROR = ERROR_0
      H = H_0
      AREA = AREA_0
      X = X_0
      YOLD = YOLD_0
      YNEW = YNEW_0

   -- exit after first or subsequent iteration
   (-B + X_0 < 0.0) and
   (k_e = min< k| (k >= 1) and (-B + k*H_0 + X_0 >= 0.0) >):
      AREA = AREA_0 + F(k_e*H_0+X_0)/2.0 + YOLD_0/2.0 +
             sum< i:=1..k_e-1 | F(i*H_0+X_0) >
      ERROR = ERROR_0
      H = H_0
      X = k_e*H_0 + X_0
      YOLD = F(k_e*H_0 + X_0)
      YNEW = F(k_e*H_0 + X_0)
endcase
```

Figure 15c. Loop Expression for TRAP

symbolic name initially assigned to each variable is its value after execution of iteration k-1 -- that is, the initial value for y_I is $y_{I_{k-1}}$. This provides a recurrence relation for each y_{I_k}, k>=1, which is in terms of the values of the variables after iteration k-1. Next, the branch predicate controlling exit from the loop is interpreted in terms of the values of the variables after execution of the kth iteration. This provides the loop exit condition, denoted lec_k, which represents the condition under which the loop will be exited after the kth iteration. Figure 15a shows the results of this evaluation for the WHILE loop in TRAP.

Loop analysis now attempts to find solutions to the recurrence relations for each variable in terms of the values of the variables on entry to the loop. The solution to the recurrence relation for y_{I_k} is denoted by $y_I(k)$ and represents the value of the variable y_I on exit from the kth iteration of the loop. Solutions are found first for those variables that do not reference other variables whose recurrence relations are as yet unsolved. Once a solution is found for a variable, it is substituted for all references to it in the remaining recurrence relations. This process is repeated, if possible, until all recurrence relations are solved. The loop exit condition lec_k is then solved by replacing each y_{I_k} referenced in the condition by its solution $y_I(k)$ and simplifying. This provides lec(k), the condition under which the loop will be exited after execution of the kth iteration. Figure 15b provides the

solutions to the recurrence relations for the loop in TRAP. Note that the solution for AREA assumes that sum<i:=1..m|...> is zero when 1>m. Although not illustrated in this example, sometimes two cases must be considered independently: 1) the first iteration of the loop (k=1), where the recurrence relations and loop exit condition depend on the values of the variables at entry to the loop; and 2) all subsequent iterations (k>1), where the recurrence relations and loop exit condition depend on the values computed by the previous iteration.

After solutions to the recurrence relations have been determined, the loop expression can be created. The loop expression for the loop in TRAP appears in Figure 15c. Each case consists of 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 one or more iterations of the loop and is derived from the case in Figure 15b. For this case, the final iteration count, call it k_e, is the minimum k, such that the loop exit condition is true. Thus, for this case the condition is

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

and the value for each variable y_I at exit from the loop is represented by $y_I(k_e)$.

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 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 PC of the evaluated loop node case is developed by interpreting the condition from the loop expression case and conjoining it to the PC 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,

```
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
```

Figure 16. Global Symbolic Evaluation of TRAP

```
(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
        (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(k_e * (-a/n + b/n) + a)/2.0 + F(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 - i*a/n + i*b/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) and (k_e = n):
          AREA = (F(a)/2.0 + F(b)/2.0 + sum< i:=1..n-1 | F(a - i*a/n + i*b/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 - i*a/n + i*b/n) > / n +
                 b * sum< i:=1..n-1 | F(a - i*a/n + i*b/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 (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 (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 - i*a/n + i*b/n) > / n +
               b * sum< i:=1..n-1 | F(a - i*a/n + i*b/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)

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. Note that in evaluating the loop expression, algebraic simplification techniques are employed to solve for the final loop iteration count k_e. The final output of global symbolic evaluation of TRAP would be a case expression, like that shown for the final node in figure 16, except that only the symbolic values for the two output parameters, AREA and ERROR, would be shown and each case with an inconsistent PC would not appear.

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 -- y_I may depend on y_J, which depends on y_I -- 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 consistency of a PC 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.

5.2. Applications

Global symbolic evaluation has several applications in program validation, many of which are similar to those of symbolic execution. The global representation provides a concise functional description of a routine, which is often useful in detecting errors. In TRAP, for example, examination of the PC and the PV for the first case of the final node in Figure 16 reveals a fairly subtle error. In implementing the loop, the incorrect assumption was made that (A < B) and thus the loop is not executed when (A > B). This error is uncovered because the global representation states that AREA = 0.0 when (A > B), which is clearly incorrect. The routine could be corrected by changing the

while loop condition to

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

Another natural application of global symbolic evaluation is test data generation, which could be performed by finding a solution for each PC associated with the final case expression in the global representation of a routine. New methods must be developed, however, for solving a PC that contains expressions for 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 could be checked for validity. Checking error conditions or user-provided assertions would also require enhancements to the techniques for determining PC consistency. With this enhancement, however, it may be possible to check error conditions or user-provided assertions for the entire routine rather than a specific path. Thus, 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 can also be applied to program specifications. The partition analysis method [RICH81] applies global symbolic evaluation to a routine, as well as to its specification, thus creating global representations of both. By comparing these two representations, a partition of the domain is determined. This partition is then utilized in verifying the routine's consistency with the specification. Information derived

from this verification process along with domain and computation testing strategies are applied to guide in the selection of test data. Unlike most testing strategies, partition analysis bases test data selection on information derived not only from the routine but also from the specification.

Global symbolic evaluation also has applications in program optimization [TOWN76]. 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 referenced in the second loop are not defined in a later iteration of the first loop. When variables modified within the loop have values that form arithmetic progressions -- that is, they are incremented by the same amount each time through the loop -- these computations can sometimes be moved out of the loop and replaced by expressions in terms of the final loop iteration count. Optimizations that perform in-line substitution of a routine may also benefit from global symbolic evaluation, since the closed form representation of the routine may enable better determination of when such substitution is useful.

6. CONCLUSION

Symbolic evaluation has several applications for program validation. 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

A primary use of symbolic evaluation is in program testing. Howden has investigated the effectiveness of several program testing strategies [HOWD78a], including 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 strategy was determined. The evaluation criteria and results of this study are summarized below.

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 Howden's 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 considered reliable for an error if the symbolic representations of the PV and PC reveal the presence of the error in an obvious way that would catch the attention of the tester. Symbolic testing of the set of paths chosen to approximate path testing guaranteed the detection of seventeen of the twenty-eight errors. Not surprisingly, it was also observed that combining both symbolic testing and actual testing was reliable for more errors than either used alone.

Symbolic evaluation methods can be used to assist in each 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.

It is important to note that in this study a strategy was considered reliable for an error only 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 such testing strategies 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 PC 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 invocation. 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. To handle input along a path, symbolic names representing the input values are assigned to the input variables whenever an input statement is encountered. The convention

previously described for representing input values must be modified slightly, however, since input may occur more than once for a variable. 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 twice along a path, the first input value might be represented by amount.1 and the second by amount.2. To handle output along a path, the symbolic values of the output variables are provided whenever an output statement is encountered. With these extensions, the variables assigned input values, the variables whose values are output, as well as the number of inputs and outputs, may vary from path to path because different input and output statements may be encountered on different paths. Moreover, for global symbolic evaluation, the number of inputs and outputs may depend on the final loop iteration counts for the routine. Although input and output along a path requires no substantial changes to the interpretive techniques originally described, the functional conceptualization of a routine must allow for an arbitrary, and perhaps varying, number of inputs and outputs.

Several approaches to routine invocation during symbolic execution and global symbolic evaluation have been proposed. During symbolic execution, the simplest approach is to represent the results of a routine invocation symbolically. For a procedure, such an approach might assign unique symbolic names for the output parameters each time the procedure is called. For a function (with no side

effects), this approach might represent each invocation by the function name along with the arguments' symbolic values at the point of invocation. The advantage of this approach is that the calling routine can be evaluated even when the called routine is not available. This approach supports unit testing, but provides less detailed information than may be needed to sufficiently analyze the routine.

Another straightforward approach to routine invocation passes information to and from the called routine via the parameters and a path through the called routine is symbolically executed. This approach is similar to normal execution. When a routine invocation is encountered, the symbolic values of the arguments are passed to the called routine. 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 PC in the usual manner. The symbolic values of the parameters are updated by the interpretation of assignment statements on the path in the called routine. When control returns to the calling routine, a pointer into the computation graph for each parameter is returned and assigned to the 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 PC and PV resulting from the symbolic execution of a path are saved for substitution. Later, when the routine is invoked, the symbolic values of the arguments are substituted for the symbolic names that were assigned to the parameters in the saved PC and PV of the called routine. The updated PC of the called routine is conjoined to the existing PC of the calling routine. If this conjunction is consistent, then the corresponding path through the called routine could be executed, and this conjunction is the PC following return from the called routine. In addition, the symbolic values of the parameters, which comprise the PV of the called routine, are returned to the calling routine. Unfortunately, subroutine substitution 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 the intended path domains and their associated path computations. This specification could then be substituted as described for subroutine substitution. Such an approach still has the drawbacks of subroutine substitution but allows for top-down testing and incremental development of software.

Global symbolic evaluation may similarly utilize three alternative approaches to routine invocation, symbolic representation of the results of an invocation, continued evaluation at an invocation, or subroutine substitution of a global representation. The first approach is the same for global symbolic evaluation as for symbolic execution. This approach was used for the global symbolic evaluation of TRAP, which invokes the function F. The second approach continues the creation of the global representation throughout the called routine. The symbolic representations of the PC and PV 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 third approach to routine invocation 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 subroutine 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 PC consistency may become extremely complicated when such an occurrence affects the PC. 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 often decide this by determining PC 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 a 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 domain. 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. Although the methods differ in their scope of representation, all three methods represent the path computations and path domains by symbolic expressions in terms of the input values.

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. Although the path domain can be represented symbolically, there is no need to determine the consistency of this representation. With neither simplification of the PC nor determination of PC 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 domain. Since many paths are not executable, some symbolic execution systems try to determine PC consistency. The most efficient implementation approach is to use the forward expansion technique and to determine PC consistency whenever an evaluated branch predicate is conjoined to the existing PC. In general, PC 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 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 describe the total routine by a symbolic representation. Since there may be an infinite number of paths in a routine, this method requires more sophisticated analysis than the mere combination of the symbolic representations for each path. Global symbolic evaluation employs a loop analysis technique, which attempts to represent each loop in a closed form that is dependent on a final loop iteration count. While this technique can

successfully analyze several types of loops, additional work is needed in this area. By developing such a closed form representationfor 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 PC consistency, which must be done for each case, is further complicated by the classification of paths. 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, although 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 PC consistency determination.

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.