

Symbolic Evaluation Methods  
For Program Analysis

Lori A. Clarke  
and  
Debra J. Richardson

COINS Technical Report 79-01

February 1979

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

This work was supported by the National Science Foundation under grant  
# NSFMC 77-02101 and the U. S. Air Force under grant # AFOSR 77-3287.

## I. Introduction

Symbolic evaluation is a data flow analysis method that analyzes program behavior by monitoring the manipulations on the input data. Symbolic evaluation methods represent computations as algebraic expressions over the input data and thus maintain the relationship between the input data and the resulting values. Normal execution computes numerical values but often loses information about the way in which the numerical values were derived. An incorrect numerical result usually does not uniquely determine the location of a miscalculation. A large part of the debugging process is concerned with isolating an erroneous calculation that resulted in a wrong numerical value. Symbolic evaluation methods can be used to aid in debugging as well as in several other types of program analysis.

There are three basic methods of symbolic evaluation; symbolic execution, global symbolic evaluation, and dynamic symbolic evaluation. Symbolic execution is a path oriented evaluation method that analyzes input data dependencies for a path. Global symbolic evaluation represents all possible data dependencies at any point in a program. Dynamic symbolic evaluation produces a trace of the data dependencies for particular input data.

In this paper we first introduce a formal notation to concisely represent each of the three methods of symbolic evaluation. Each method is then explained and examples of the three methods are given to demonstrate their corresponding strengths and weaknesses. Also, several applications of each method are discussed. Symbolic execution is the best known of the three techniques so it is described first and in more detail than the other two methods. Several different implementation techniques of symbolic execution systems are compared. The other symbolic evaluation methods are then described and compared to symbolic execution.

## 2. Functional Notation

In this section we introduce some basic notation that will be used in formalizing the results of each of the three methods of symbolic evaluation.

A program  $P$  accepts input values  $(x_1, x_2, \dots, x_M)$  and computes output values  $(z_1, z_2, \dots, z_N)$ . The domain  $X$  of the program  $P$  is a cross product,  $X = X_1 \times X_2 \times \dots \times X_M$ , where each  $X_I$  is the domain for input value  $x_I$ . An element of  $X$  is a vector  $x$  with specific input values,  $x = (x_1, x_2, \dots, x_M)$ , and corresponds to a single point in the  $M$ -dimensional input space  $X$ . Likewise, the codomain  $Z$  of a program is a cross product,  $Z = Z_1 \times Z_2 \times \dots \times Z_N$ , where each  $Z_J$  is the codomain for output value  $z_J$ . An element of  $Z$  is a vector  $z$  with specific output values,  $z = (z_1, z_2, \dots, z_N)$ , and corresponds to a single point in the  $N$ -dimensional output space  $Z$ . We will also reference a vector  $y = (y_1, y_2, \dots, y_W)$  of program variables, which store the values (both intermediate and output) computed by the program, as well as the input values,  $(x_1, x_2, \dots, x_M)$ . For the purposes of this paper, we will assume that the program variables,  $y_1, y_2, \dots, y_N$ ,  $N \leq W$ , store the output values,  $(z_1, z_2, \dots, z_N)$ .

The program  $P$  computes some function  $F$ , which has the same domain and codomain as does  $P$ ; hence,  $F$  is a mapping from  $X$  to  $Z$ ,  $F : X \rightarrow Z$ . In general,  $F$  is composed of a set of *partial functions*, which are defined over disjoint subsets of the domain  $X$ . Suppose  $F = \{F_1, F_2, \dots, F_Q\}$ ,  $1 \leq Q \leq \infty$ , where each  $F_G$  is a partial function defined over  $D_G^F$  and undefined elsewhere in  $X$ . This subdomain  $D_G^F$  is called the *domain of definition* of  $F_G$ . Each of the partial functions  $F_G$  produces an  $N$ -tuple in the codomain  $Z$ ;  $F_G$  may thus be represented as a vector of  $N$  *component functions*,  $F_G = (f_{G1}, f_{G2}, \dots, f_{GN})$ , where the  $J$ th component produces the  $J$ th output value,  $z_J$ . Hence, for any  $x \in D_G^F$ ,  $F(x) = (f_{G1}(x), f_{G2}(x), \dots, f_{GN}(x)) \in Z$  and  $f_{GJ}(x) \in Z_J$ .

The program  $P$  that computes the function  $F$  has a construct similar to the

partial functions of  $F$ . Rather than performing the same computation on all elements of the input space,  $P$  may compute different functions along different *program paths*, which are executed for disjoint subsets of the domain  $X$ . A simplifying assumption that each path has  $M$  input values and  $N$  output values can be made without loss of generality; any program that does not satisfy this assumption can be transformed into an equivalent program that does, using  $\Lambda$  to represent an undefined value. Suppose  $P$  specifies a set of program paths,  $\{P_1, P_2, \dots, P_R\}$ ,  $1 \leq R \leq \infty$ , where each  $P_H$  is executed for a subset  $D_H^P$  of the program domain  $X$ . This subdomain  $D_H^P$  is called the *path domain* of  $P_H$ . Likewise, each path  $P_H$  specifies a vector of *path functions*,  $(p_{H1}, p_{H2}, \dots, p_{HN})$ , where  $J$ th component computes  $z_J$ .

Symbolic evaluation can be used to generate representations for the path domains and path functions of a program. In order to describe these representations and how each method of symbolic evaluation generates them, a few additional definitions must be presented.

Data flow analysis methods typically represent a program by a directed graph describing the possible flow of control through the program. The nodes in the graph,  $\{n_1, n_2, \dots, n_q\}$ , represent statements. Each edge is specified by an ordered pair of nodes,  $(n_i, n_j)$ , that indicates that a possible 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 which governs traversal of the edge  $(n_i, n_j)$  is denoted by  $bp(n_i, n_j)$ . For a binary condition following the node  $n_i$  and preceding nodes  $n_j$  and  $n_k$ , the branch predicate for one edge  $(n_i, n_k)$  is the complement of the branch predicate for the other edge  $(n_j, n_k)$  - thus,  $bp(n_i, n_k) = \sim bp(n_j, n_k)$ . Conditional statements, such as computed GO TO or CASE statements, may have more than two successor nodes and each branch predicate must be represented appropriately. For the

purposes of this paper, the *control flow graph* of a program is a directed graph which has a single entry point, the start node  $n_s$ , and a single exit point, the final node  $n_f$ . Both the start node and the final node are null nodes added to the graph when necessary to accomplish this single-entry, single-exit form without loss of generality.

The procedure in figure 1 calculates the time and distance at which a starship's velocity reduces to zero on its approach to dock on the star base station. The statements in DOCKING are annotated with their associated node numbers, and figure 2 shows the control flow graph for this procedure. This procedure is used throughout the paper to demonstrate the three methods of symbolic evaluation.

A *path* in a control flow graph is a sequence of statements,  $(n_{i_0}, n_{i_1}, \dots, n_{i_t})$ , where there exists a possible transfer of control from  $n_{i_j}$  to  $n_{i_{j+1}}$  for all  $n_{i_j}$ ,  $0 \leq j < t$ . A *partial program path*  $T_{ku}$  is a path which begins with the start node - that is,  $T_{ku} = (n_s, n_{k1}, n_{k2}, \dots, n_{ku})$ . Hence, for any partial program path  $T_{ku}$  with  $u \geq 1$ ,  $T_{ku} = (T_{ku-1}, n_{ku})$ , where  $T_{k0} = (n_s)$ . A *program path*  $P_H$  is a path which begins with the start node and ends with the final node - that is  $P_H = (n_s, n_{H1}, n_{H2}, \dots, n_{Hv}, n_f)$ . There is no guarantee that a sequence of statements representing a path is executable; some paths may be infeasible due to contradictory conditions governing the transfers of control along the path. The control flow graph is a representation of all possible paths through the corresponding program.

All three symbolic evaluation methods use the control flow graph to maintain a description of the program state at every point in the evaluation of the program. The state of the program includes some description of the path followed to reach the present point in the evaluation, as well as the values obtained for all program variables following the evaluation of that partial

```

n s procedure DOCKING(STATION, STARSHIP, THRUST, VELOCITY, DELTAT, TIME, DISTANCE, ERROR)

```

```

{ starship docking calculation (approximation) determines the time and }
{ the distance at which the starship's velocity reduces to zero on its }
{ approach to the star base station }

```

```

{ input variables: }

```

```

{ station - mass of the star base station (kg) }
{ starship - mass of the starship (kg) }
{ thrust - thrust force of the starship's engine (nt) }
{ velocity - initial velocity of the starship (m/s) }
{ deltat - change requested between iterations (a smaller value will }
{ make the calculation more exact) (s) }
{ time - initial time (s) }
{ distance - initial distance between the base and the starship (m) }

```

```

{ output variables: }

```

```

{ time - final time (s) }
{ distance - final distance between the base and the starship (m) }
{ error - nonzero if any input is invalid }

```

```

{ intermediate variables: }

```

```

{ gconst - universal gravitational constant (6.67E-11 nt-m2 /kg2) }
{ gravity - gravitational force of base station (m/s2) }
{ constacc - constant acceleration of starship (m/s2) }
{ currvel - current velocity of the starship (m/s) }
{ nextvel - velocity of the starship in next time interval (m/s) }

```

```

real STATION, STARSHIP
real THRUST, VELOCITY
real DELTAT, TIME, DISTANCE
real GCONST, GRAVITY, CONSTACC
real CURRVEL, NEXTVEL
integer ERROR

```

Figure 1  
Procedure DOCKING

```

{ all input values must be positive
{ set error flag if one is ≤ 0
}

if (STATION≤0.0 or STARSHIP≤0.0 or
THRUST≤0.0 or VELOCITY≤0.0 or
DELTAT≤0.0 or TIME≤0.0 or DISTANCE≤0.0)
then
n1     ERROR ← 1
else
{
{ input values are valid, continue computation
{ initialize the universal gravitational constant
n2     GCONST ← 6.67 * 10**(-11)
{ compute the gravitational force
{ first check for a zero divisor
{ if (DISTANCE**2 = 0.0)
{ then
n3     ERROR ← 1
{ else
n4     GRAVITY ← GCONST * STATION * STARSHIP / (DISTANCE**2)
{
{ gravity and thrust are assumed constant
{ throughout the computations to follow
{ the acceleration due to the starship's
{ engine is thrust / starship
{ (acceleration = force / mass)
{ compute the constant acceleration of
{ the starship which is the difference
{ of these two opposing accelerations
n5     CONSTACC ← GRAVITY - THRUST/STARSHIP
{
{ current velocity is the initial velocity
n6     CURRVEL ← VELOCITY
{
{ compute the velocity in the second time
{ interval (this initializes the loop)
{ nextvel ← vel + acceleration*deltat
n7     NEXTVEL ← CURRVEL + CONSTACC*DELTAT
{
{ determine when the velocity reduces
{ to zero by iteratively computing the
{ next velocity as a function of the
{ current velocity, the acceleration
{ (force/mass), and the change in time
{
{ repeat
n8     DISTANCE ← DISTANCE - CURRVEL*DELTAT
n9     CURRVEL ← NEXTVEL
n10    TIME ← TIME + DELTAT
n11    NEXTVEL ← CURRVEL + CONSTACC*DELTAT
until (NEXTVEL≤0.0)
ERROR ← 0
endif
endif
end {DOCKING}
$

```

Figure 1 (cont.)

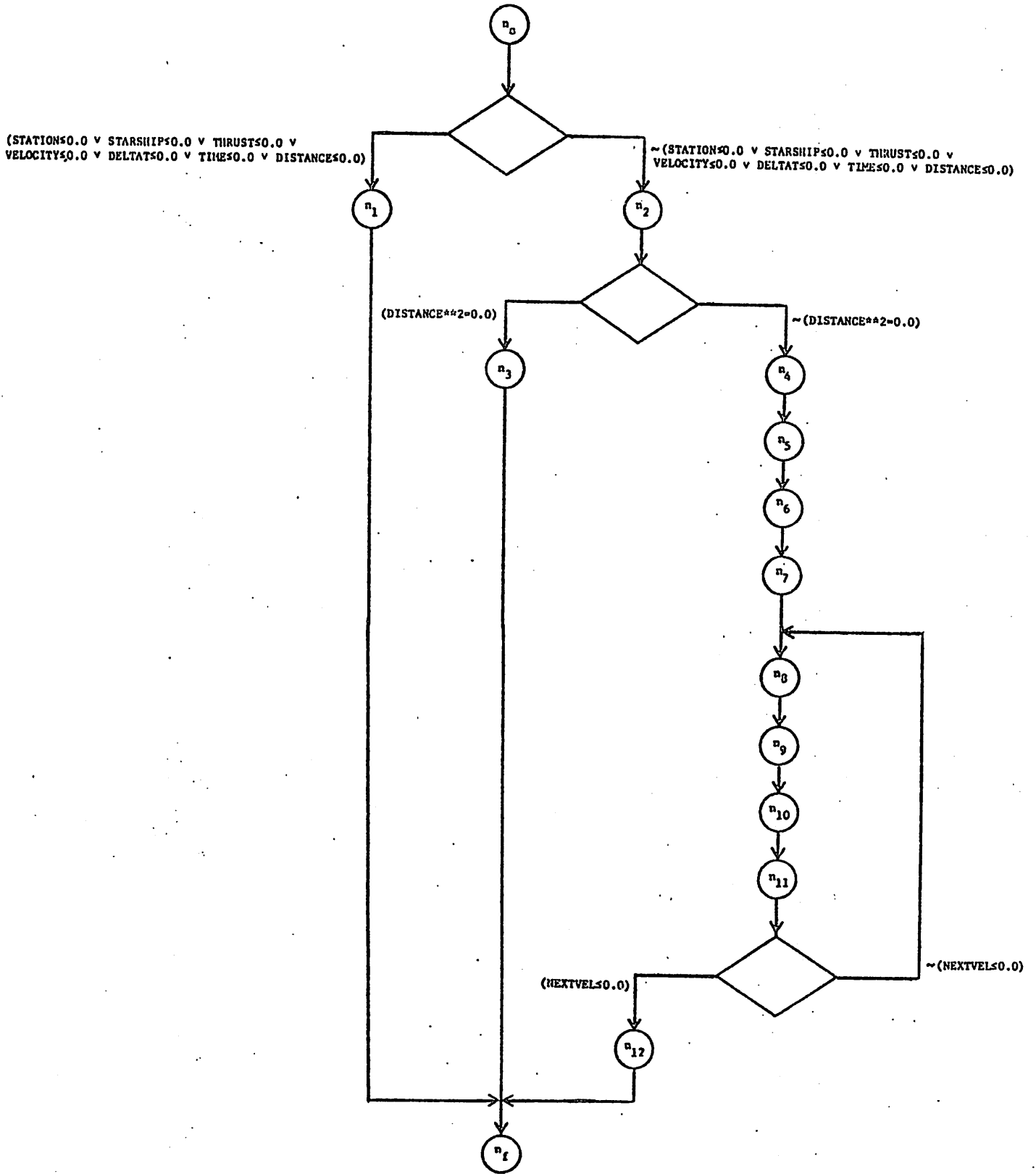


Figure 2  
Control Flow Graph for DOCKING



program path. The data descriptions generated in symbolic evaluation are symbolic representations of the program state. Given a partial program path,  $T_{ku} = (n_s, n_{k1}, n_{k2}, \dots, n_{ku})$ ,  $n_{ku}$  is the present point in the evaluation.  $VAL[T_{ku}]$  represents the values of all program variables  $(y_1, y_2, \dots, y_W)$  after evaluation of the partial program path  $T_{ku}$ .  $VAL[T_{ku}]$  is a vector containing an element for each program variable. Hence,  $VAL[T_{ku}] = (s(y_1[T_{ku}]), s(y_2[T_{ku}]), \dots, s(y_W[T_{ku}]))$ , where  $s(y_L[T_{ku}])$  denotes the symbolic value of program variable  $y_L$ , after evaluating  $T_{ku}$ , in terms of the symbolic names representing the input values. The path condition,  $PC[T_{ku}]$ , is the conjunct of the branch predicates evaluated along this particular partial program path.  $PC[T_{ku}] = s(bp(n_s, n_{k1})[T_{k0}]) \wedge s(bp(n_{k1}, n_{k2})[T_{k1}]) \wedge \dots \wedge s(bp(n_{k_{u-1}}, n_{ku})[T_{k_{u-1}}])$ , where  $s(bp(n_{k_{j-1}}, n_{kj})[T_{k_{j-1}}])$  denotes the symbolic value of the branch predicate when evaluated over the values of the program variables preceding traversal of the corresponding edge—that is, over  $VAL[T_{k_{j-1}}]$ . The path condition can be rewritten as  $PC[T_{ku}] = PC[T_{k_{u-1}}] \wedge s(bp(n_{k_{u-1}}, n_{ku})[T_{k_{u-1}}])$ . Finally,  $STATE[T_{ku}] = (T_{ku}, VAL[T_{ku}], PC[T_{ku}])$  represents the program state following symbolic evaluation of the partial program path  $T_{ku}$ . The partial program path will not be included in the notation when the path is obvious from the context.

The symbolic evaluation of any element of the control flow graph - a statement or a transfer of control - changes the program state. Initially, the program state is defined as

$$\begin{aligned} T_{k0} &= (n_s) \\ VAL[T_{k0}] &= (\Lambda, \dots, \Lambda) \\ PC[T_{k0}] &= true \\ STATE[T_{k0}] &= (T_{k0}, VAL[T_{k0}], PC[T_{k0}]). \end{aligned}$$

All variables are initialized at the start node to the undefined value  $\Lambda$ , with the following exceptions: variables that are initialized by DATA statements

are assigned the corresponding constant value; variables that are parameters of the initial procedure of evaluation are assigned symbolic names. Symbolic names are assigned to input variables whenever input occurs on the program path. Throughout the symbolic evaluation, all symbolic representations of variable and branch predicate values are in terms of these symbolic names that represent the input values. This is accomplished by substituting the current symbolic value of a variable into an expression wherever that variable is referenced.

When evaluating a statement, other than an input statement, the program state will be updated in the *VAL* component only. For any variable that is assigned a new value within the statement, the corresponding component of the *VAL* vector is updated. For instance, if the assignment statement  $y_J = y_I * y_K$  occurs, then the  $y_J$  component of the *VAL* vector will change from its former value  $s(y_J)$  to the value of the algebraic expression  $s(y_I) * s(y_K)$ .

In the evaluation of a transfer of control, say edge  $(n_{kj}, n_{k j+1})$ , the program state is updated in the *PC* component only. The path condition will be augmented by the symbolic value of the branch predicate governing traversal of this edge - that is,  $PC[T_{k j+1}] = PC[T_{kj}] \wedge s(bp(n_{kj}, n_{k j+1})[T_{kj}])$ .

Following the evaluation of a complete program path, the symbolic representation of the program state defines the path functions and path domains of a program. Given a complete program path  $P_H$ , the program state after evaluation of the final node may be represented as

$$\begin{aligned}
 P_H &= (n_s, n_{H1}, n_{H2}, \dots, n_{Hv}, n_f) \\
 VAL[P_H] &= (s(y_1[P_H]), s(y_2[P_H]), \dots, s(y_w[P_H])) \\
 PC[P_H] &= s(bp(n_s, n_{H1})[T_{H0}]) \wedge \dots \wedge s(bp(n_{Hv}, n_f)[T_{Hv}]) \\
 STATE[P_H] &= (P_H, VAL[P_H], PC[P_H]).
 \end{aligned}$$

The path functions  $(p_{H1}, p_{H2}, \dots, p_{HN})$ , which compute the output values  $(z_1, z_2, \dots, z_N)$ , are provided by  $p_{HJ} = s(y_J[P_H])$ . Since all symbolic representations are in terms of the symbolic input values,  $p_{HJ}$  is a symbolic computational expression of the output value  $z_J$  in terms of the input values  $(x_1, x_2, \dots, x_M)$ . The path condition  $PC[P_H]$  provides a system of constraints on the program's input values and defines the path domain  $D_H^P$ . The subset of elements of the program domain that will cause execution of this program path is defined by  $D_H^P = \{x \in X \text{ such that } PC[P_H] \text{ is true}\}$ . These path functions and path domains can be generated for any program path that can be symbolically evaluated.

Each of the three methods of symbolic evaluation maintains a slightly different representation for the program state at any point in the evaluation. Furthermore, each method generates a slight variation of the path domains and path functions as final evaluation of the program.

Symbolic execution supports a program state which most resembles the *STATE* vector defined above. This method generates output for each path that is symbolically executed. For the most part, following symbolic execution of a particular path,  $P_H$ , the output produced consists of three things: the sequence of statements forming the path; a system of constraints on the program's input variables; and a vector containing a computational expression for each output variable. The constraints are analogous to the path condition ( $PC[P_H]$ ) and define the path domain. The vector corresponds to the output variable components of the symbolic value vector ( $VAL[P_H]$ ) given by the path functions  $(p_{H1}, p_{H2}, \dots, p_{HN})$  computed along this path. After symbolic execution of a program path, therefore, output similar to that shown in Figure 3 might be produced.

## STATEMENTS ON THIS PATH

$$n_s, n_{H1}, n_{H2}, \dots, n_{Hv}, n_f$$

## SYMBOLIC REPRESENTATION OF PATH CONDITION

$$s(bp(n_s, n_{H1})[T_{H0}]) \wedge s(bp(n_{H1}, n_{H2})[T_{H1}])$$

$$\wedge \dots \wedge s(bp(n_{Hv}, n_f)[T_{Hv}])$$

## SYMBOLIC REPRESENTATION OF OUTPUT VARIABLES

$$z_1 = p_{H1} = s(y_1[P_H])$$

$$z_2 = p_{H2} = s(y_2[P_H])$$

.

.

.

$$z_N = p_{HN} = s(y_N[P_H])$$

Figure 3.

## Final Results of Symbolic Execution

Rather than evaluate a program on a path-by-path basis, the method of global symbolic evaluation maintains a representation of the program state at a point in the evaluation as a conditional symbolic expression. This case-like construct encompasses the symbolic values of all program variables regardless of the partial program path followed to reach this point. The output generated following global symbolic evaluation of a program reflects this representation of the program state. Suppose the program has the paths  $\{P_1, P_2, \dots, P_R\}$ , then the final evaluation might have a form such as Figure 4.

case

$$PC[P_1]: z_1 = p_{11} = s(y_1[P_1])$$

$$z_2 = p_{12} = s(y_2[P_1])$$

·  
·  
·

$$z_N = p_{1N} = s(y_N[P_1])$$

·  
·  
·

$$PC[P_R]: z_1 = p_{R1} = s(y_1[P_R])$$

$$z_2 = p_{R2} = s(y_2[P_R])$$

·  
·  
·

$$z_N = p_{RN} = s(y_N[P_R])$$

endcase

Figure 4.

#### Final Results of Global Symbolic Evaluation

On the other end of the spectrum is the method of dynamic symbolic evaluation, which performs analysis on an input-by-input basis. A particular program path is evaluated while the program is actually executed for specific input data. Given an input vector  $x$ , a path, say  $P_H$ , is executed. This method traces the statements that are executed. In addition to supplying the output values that result from the execution, dynamic symbolic evaluation provides algebraic expressions for the output values. Following dynamic symbolic evaluation, output similar to figure 5 might result, where  $a(y_J[P_H])$  denotes the actual value  $z_J$  computed by the execution of the program on the data vector  $x$ .

## STATEMENTS EXECUTED

$$n_s, n_{H1}, \dots, n_{Hv}, n_f$$

## SYMBOLIC AND ACTUAL VALUES OF OUTPUT VARIABLES

$$z_1 = p_{H1} = s(y_1[P_H]) = a(y_1[P_H])$$

$$z_2 = p_{H2} = s(y_2[P_H]) = a(y_2[P_H])$$

$$\cdot$$

$$\cdot$$

$$\cdot$$

$$z_N = p_{HN} = s(y_N[P_H]) = a(y_N[P_H])$$

Figure 5.

## Final Results of Dynamic Symbolic Evaluation

These methods of symbolic evaluation will be explained in more detail in the sections which follow.

### 3. Symbolic Execution

Symbolic execution analyzes distinct program paths. In general, symbolic execution is attempted on only a subset of the paths in a program since a program containing a loop may contain an effectively infinite number of paths. Symbolic execution represents both the computations and the conditional statements on the selected path as algebraic expressions in terms of symbolic input values. The rest of this section describes symbolic execution in more detail as well as several implementation techniques and applications.

#### 3.1 General Method

Symbolic execution initiates its analysis by building the control flow graph of the program. As a path through the program is evaluated or "executed", the statements on the path are evaluated as if they were straight-line code. The branch predicates encountered along the path are also evaluated; the combination of those predicates dictates the input values for which this path can be executed. When an input statement is analyzed, the input values are represented by symbolic names. Throughout the analysis, the representations of all program variables are maintained as algebraic expressions in terms of these symbolic names. These algebraic expressions are formed by the evaluation of any assignment along the path; such an assignment causes an update to the *VAL* of the program state.

During symbolic execution of a path each branch predicate is evaluated over the symbolic values of the variables at that point on the path. The symbolic evaluation of a branch predicate results in a constraint, an equality or inequality condition, on the input data. Each constraint is then conjoined with all previously evaluated constraints for this path to form the path condition or *PC*. Not all paths in the program graph are executable. The path condition of a program path may be inconsistent, in which case no input data

exists that could cause execution of the path. Symbolic execution systems create the path condition and determine path condition consistency as well as create the path function.

### 3.2 Implementation Methods

Several symbolic execution systems have been developed [3, 6, 13, 16, 17, 19, 20] using one of two implementation techniques, either forward expansion or backward substitution. In addition, some of these systems try to determine path condition consistency [3, 6, 17, 20], and again two different approaches have successfully been tried. These approaches are referred to as the algebraic and axiomatic approaches. In this section, both methods of symbolic execution and path condition consistency determination will be described.

Forward expansion is the most intuitive approach to creating the algebraic expressions. Beginning with the start node, symbolic expressions are built as each statement in the path is encountered. The DOCKING procedure of figure 1 has undergone symbolic execution by forward expansion of two distinct program paths. Figure 6 shows how the *VAL* and *PC* evolve for an executable path, while figure 7 shows the evolution for a nonexecutable path.

Before either symbolic execution technique is initiated, the source code is first translated into an intermediate form of binary expressions, an operator and two operands. During forward expansion, the binary expressions in each executed statement are then used to form an acyclic directed graph of a program's symbolic computations. Each variable that is assigned a value during execution of the path is actually assigned a pointer into this computational graph. The node of the graph that is pointed to by a variable can be treated as the root of a binary expression tree for this variable. Traversing the tree in inorder determines the symbolic expression for this variable. Figure 8 shows how the graph evolves during symbolic execution of the program path of the procedure DOCKING that was shown in figure 6.



statement	VAL	PC
n <sub>s</sub>	TIME: time DISTANCE: distance ERROR: $\Lambda$ STATION: station STARSHIP: starship THRUST: thrust VELOCITY: velocity DELTAT: deltat GCONST: $\Lambda$ GRAVITY: $\Lambda$ CONSTACC: $\Lambda$ CURRVEL: $\Lambda$ NEXTVEL: $\Lambda$	true
n <sub>2</sub>	VAL[n <sub>s</sub> ] updated by GCONST: $6.67 \cdot 10^{**(-11)}$	PC[n <sub>s</sub> ] $\wedge$ $\sim(\text{station} \leq 0.0 \vee \text{starship} \leq 0.0 \vee$ $\text{thrust} \leq 0.0 \vee \text{velocity} \leq 0.0 \vee$ $\text{deltat} \leq 0.0 \vee \text{time} \leq 0.0 \vee \text{distance} \leq 0.0)$
n <sub>4</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ] updated by GRAVITY: $6.67 \cdot 10^{**(-11)} \cdot \text{station} \cdot \text{starship}$ /distance**2	PC[n <sub>s</sub> ,n <sub>2</sub> ] $\wedge$ $\sim(\text{distance}^{**2} = 0.0)$
n <sub>5</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ] updated by CONSTACC: $6.67 \cdot 10^{**(-11)} \cdot \text{station} \cdot \text{starship}$ /distance**2 - thrust/starship	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ] $\wedge$ true
n <sub>6</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ] updated by CURRVEL: velocity	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ] $\wedge$ true
n <sub>7</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ] updated by NEXTVEL: velocity + $(6.67 \cdot 10^{**(-11)}$ $\cdot \text{station} \cdot \text{starship} / \text{distance}^{**2}$ - thrust/starship) * deltat	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ] $\wedge$ true

Figure 6

Symbolic Execution of an Executable Path Using Forward Expansion

n <sub>8</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ] updated by DISTANCE: distance - velocity*deltat	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ] ^ true
n <sub>9</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ] updated by CURRVEL: velocity + (6.67*10**(-11)) *station*starship/distance**2 - thrust/starship) * deltat	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ] ^ true
n <sub>10</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ,n <sub>9</sub> ] updated by TIME: time + deltat	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ,n <sub>9</sub> ] ^ true
n <sub>11</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ,n <sub>9</sub> ,n <sub>10</sub> ] updated by NEXTVEL: velocity + (6.67*10**(-11)) *station*starship/distance**2 - thrust/starship) * deltat + (6.67*10**(-11))*station *starship/distance**2 - thrust/starship) * deltat	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ,n <sub>9</sub> ,n <sub>10</sub> ] ^ true
n <sub>12</sub>	VAL[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ,n <sub>9</sub> ,n <sub>10</sub> ,n <sub>11</sub> ] updated by ERROR: 0	PC[n <sub>s</sub> ,n <sub>2</sub> ,n <sub>4</sub> ,n <sub>5</sub> ,n <sub>6</sub> ,n <sub>7</sub> ,n <sub>8</sub> ,n <sub>9</sub> ,n <sub>10</sub> ,n <sub>11</sub> ] ^ (velocity + (6.67*10**(-11))*station*starship/ distance**2 - thrust/starship) * deltat + (6.67*10**(-11))*station*starship/ distance**2 - thrust/starship) * deltat) ≤ 0.0

Figure 6 (cont.)

STATEMENTS ON THIS PATH

$n_s, n_2, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}, n_{11}, n_{12}, n_f$

SYMBOLIC REPRESENTATION OF PATH CONDITION

$\sim (\text{station} \leq 0.0 \vee \text{starship} \leq 0.0 \vee \text{thrust} \leq 0.0 \vee$   
 $\text{velocity} \leq 0.0 \vee \text{deltat} \leq 0.0 \vee \text{time} \leq 0.0 \vee \text{distance} \leq 0.0)$   
 $\wedge \sim (\text{distance}^{**2} = 0.0) \wedge ((\text{velocity} + (6.67 * 10^{**(-11)})$   
 $* \text{station} * \text{starship} / \text{distance}^{**2} - \text{thrust} / \text{starship})$   
 $* \text{deltat} + (6.67 * 10^{**(-11)}) * \text{station} * \text{starship} / \text{distance}^{**2}$   
 $- \text{thrust} / \text{starship}) * \text{deltat}) \leq 0.0)$

SIMPLIFIED SYMBOLIC REPRESENTATION OF PATH CONDITION

$\text{station} > 0.0 \wedge \text{starship} > 0.0 \wedge \text{thrust} > 0.0 \wedge$   
 $\text{velocity} > 0.0 \wedge \text{deltat} > 0.0 \wedge \text{time} > 0.0 \wedge \text{distance} > 0.0$   
 $\wedge (\text{velocity} * \text{distance}^{**2} * \text{starship} * \text{deltat} +$   
 $13.34 * 10^{**(-11)} * \text{station} * \text{starship}^{**2} * \text{deltat} -$   
 $2 * \text{thrust} * \text{distance}^{**2} * \text{deltat}) / \text{distance}^{**2} * \text{starship} \leq 0.0$

SYMBOLIC REPRESENTATION OF OUTPUT VARIABLES

TIME = time + deltat  
DISTANCE = distance + velocity\*deltat  
ERROR = 0

Figure 6 (cont.)

Final Output from Symbolic Execution of an Executable Path Using Forward Expansion

statement	VAL	PC
$n_s$	TIME: time DISTANCE: distance ERROR: $\Lambda$ STATION: station STARSHIP: starship THRUST: thrust VELOCITY: velocity DELTAT: deltat GCONST: $\Lambda$ GRAVITY: $\Lambda$ CONSTACC: $\Lambda$ CURRVEL: $\Lambda$ NEXTVEL: $\Lambda$	true
$n_2$	VAL[ $n_s$ ] updated by GCONST: $6.67 \cdot 10^{**(-11)}$	PC[ $n_s$ ] $\wedge$ $\sim(\text{station} \leq 0.0 \vee \text{starship} \leq 0.0 \vee$ $\text{thrust} \leq 0.0 \vee \text{velocity} \leq 0.0 \vee$ $\text{deltat} \leq 0.0 \vee \text{time} \leq 0.0 \vee \text{distance} \leq 0.0)$
$n_3$	VAL[ $n_s, n_2$ ] updated by ERROR: 1	PC[ $n_s, n_2$ ] $\wedge$ $\text{distance}^{**2} = 0.0$

Figure 7

Symbolic Execution of a Nonexecutable Path Using Forward Expansion

STATEMENTS ON THIS PATH

$n_s, n_2, n_3, n_f$

SYMBOLIC REPRESENTATION OF PATH CONDITION

$\sim (\text{station} \leq 0.0 \vee \text{starship} \leq 0.0 \vee \text{thrust} \leq 0.0 \vee$   
 $\text{velocity} \leq 0.0 \vee \text{deltat} \leq 0.0 \vee \text{time} \leq 0.0 \vee \text{distance} \leq 0.0)$   
 $\wedge \text{distance} ** 2 = 0.0$

SIMPLIFIED SYMBOLIC REPRESENTATION OF PATH CONDITION

$\text{station} > 0.0 \wedge \text{starship} > 0.0 \wedge \text{thrust} > 0.0 \wedge$   
 $\text{velocity} > 0.0 \wedge \text{deltat} > 0.0 \wedge \text{time} > 0.0 \wedge \text{distance} > 0.0$   
 $\wedge \text{distance} = 0.0$

\*\*\* NONEXECUTABLE PATH \*\*\*

Figure 7 (cont.)

Final Output from Symbolic Execution of a Nonexecutable Path Using Forward Expansion

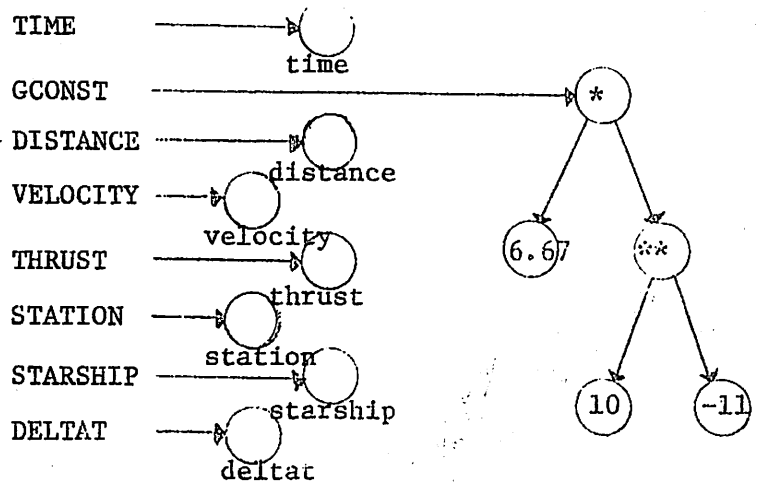


Figure 8a

Computational Graph After Partial Program Path  $(n_s, n_2)$

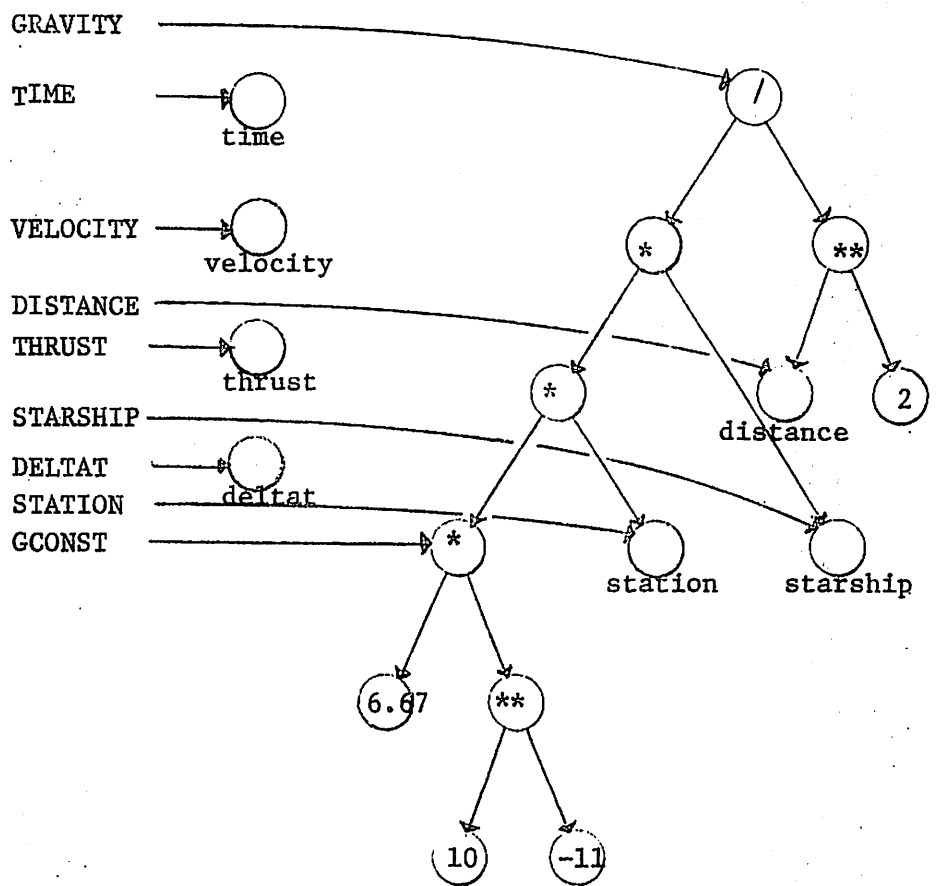


Figure 8b

Computational Graph After Partial Program Path  $(n_s, n_2, n_4)$

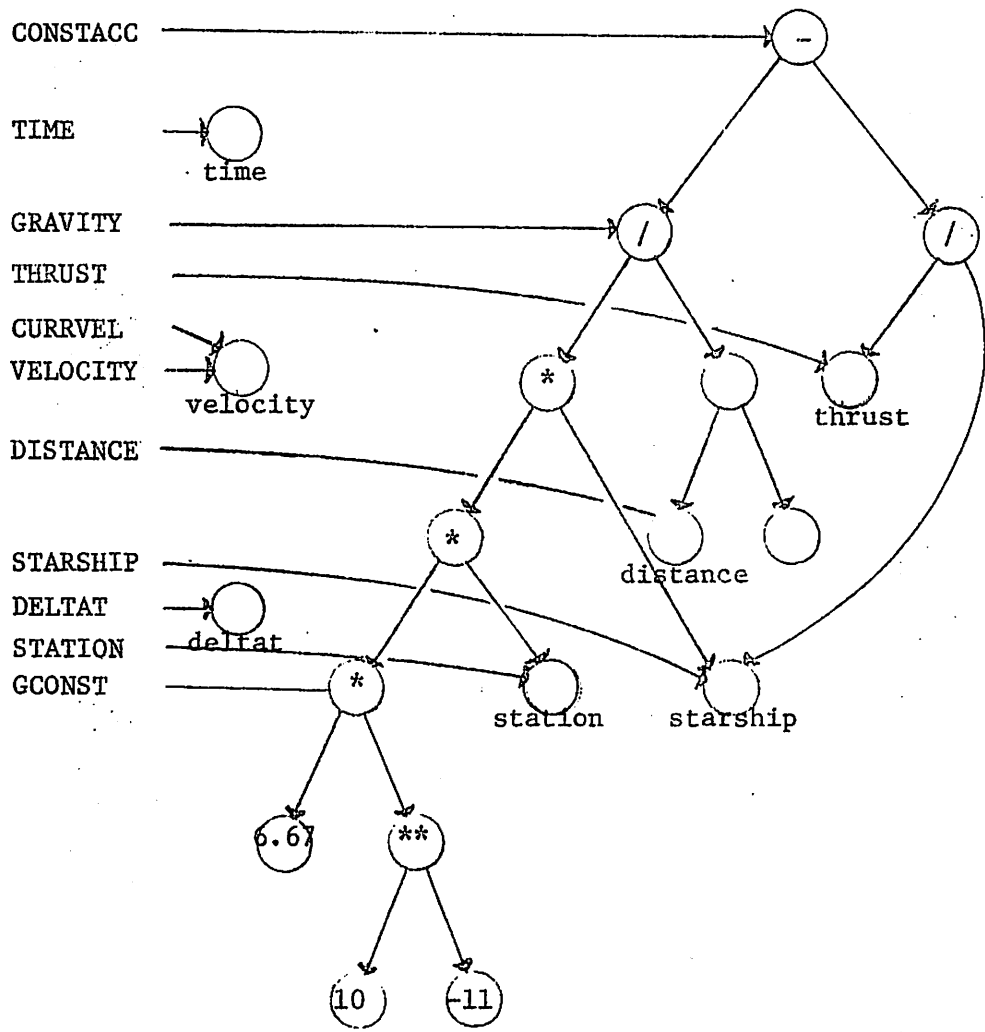


Figure 8c

Computational Graph After Partial Program Path  
( $n_s, n_2, n_4, n_5, n_6$ )

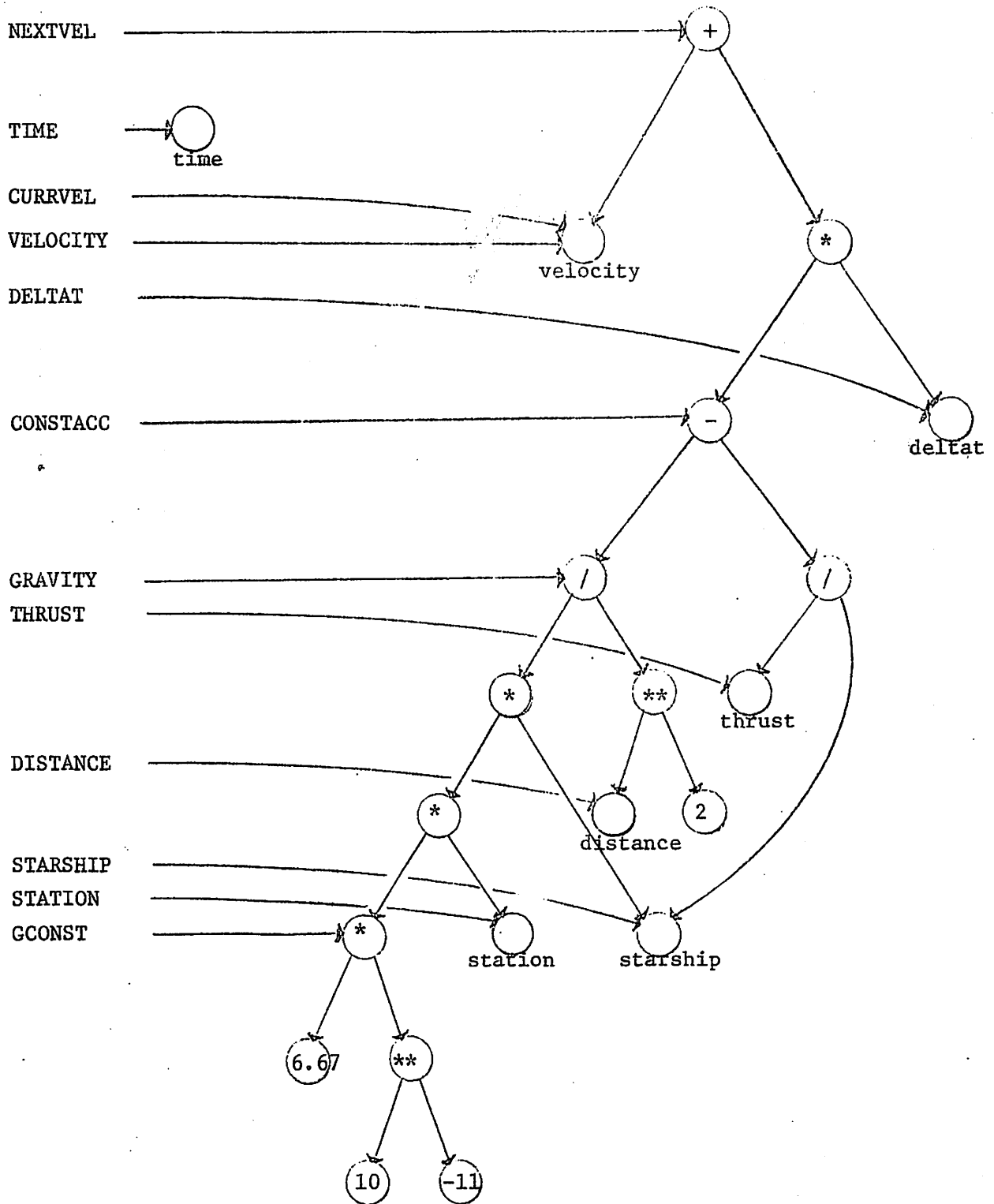


Figure 8d

Computational Graph After Partial Program Path  
 $(n_8, n_2, n_4, n_5, n_6, n_7)$



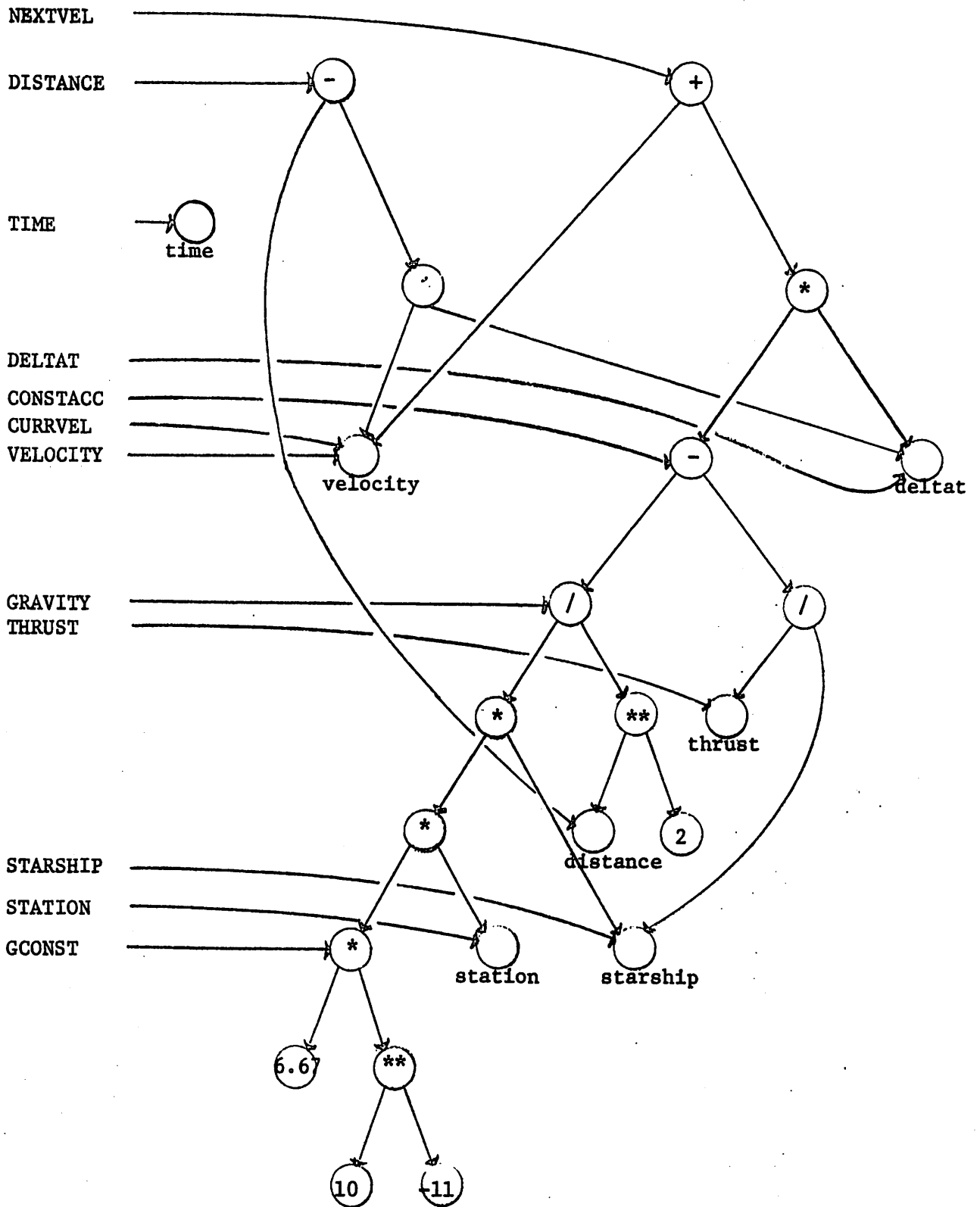


Figure 8e

Computational Graph After Partial Program Path  
( $n_8, n_2, n_4, n_5, n_6, n_7, n_8$ )

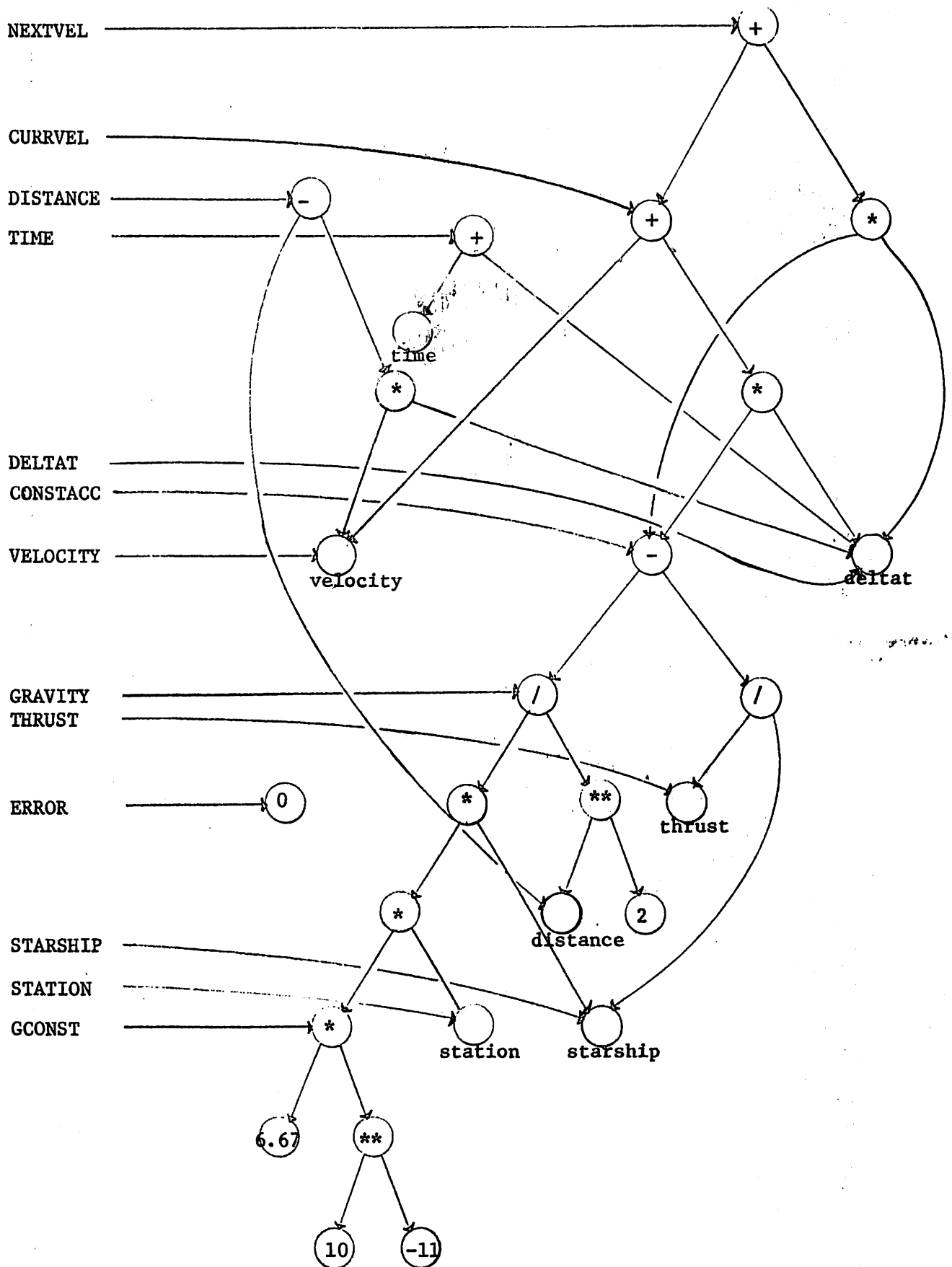


Figure 8f

Computational Graph After Partial Program Path

$(n_8, n_2, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}, n_{11}, n_{12}, n_f)$

Conditional statements can also be represented symbolically by an acyclic graph using the same form of binary expressions. The false branch of the first branch predicate is followed in the transfer from  $n_s$  to  $n_2$ , and this constraint is the *PC* for the partial program path  $(n_s, n_2)$ . Figure 9 shows the computational graph representation for this evaluated branch predicate.

Though the computational graph appears rather complicated to follow with the eye, it is easy to build and maintain. The graph can easily be maintained in a table with three fields: one for the operator and two for the operands. The tabular representation of the computational graph in figure 8f is shown in figure 10 where  $CG_i$  represents a pointer to the  $i$ th entry in the tabular computational graph.

A more detailed description of the forward expansion approach to symbolic execution can be found in [6]. There is a close similarity between the described forward expansion technique and the technique of common subexpression elimination used by many optimizing compilers, which is described in [9].

The backward substitution technique starts at the end of the path and develops each variables' symbolic expression by substituting the right hand side of an assignment statement for all occurrences of the left hand side variable [14, 16]. The backward substitution approach was proposed for systems concerned only with creating the path condition and not concerned with the symbolic expressions for intermediate or output variables. With this restriction, the backward substitution technique saves space by not maintaining extraneous expressions for the intermediate and output variables. An example of backward substitution is shown in figure 11, using the DOCKING procedure of figure 1 and again symbolically executing the program path of figure 6. An implementation technique that is just the reverse of the described forward

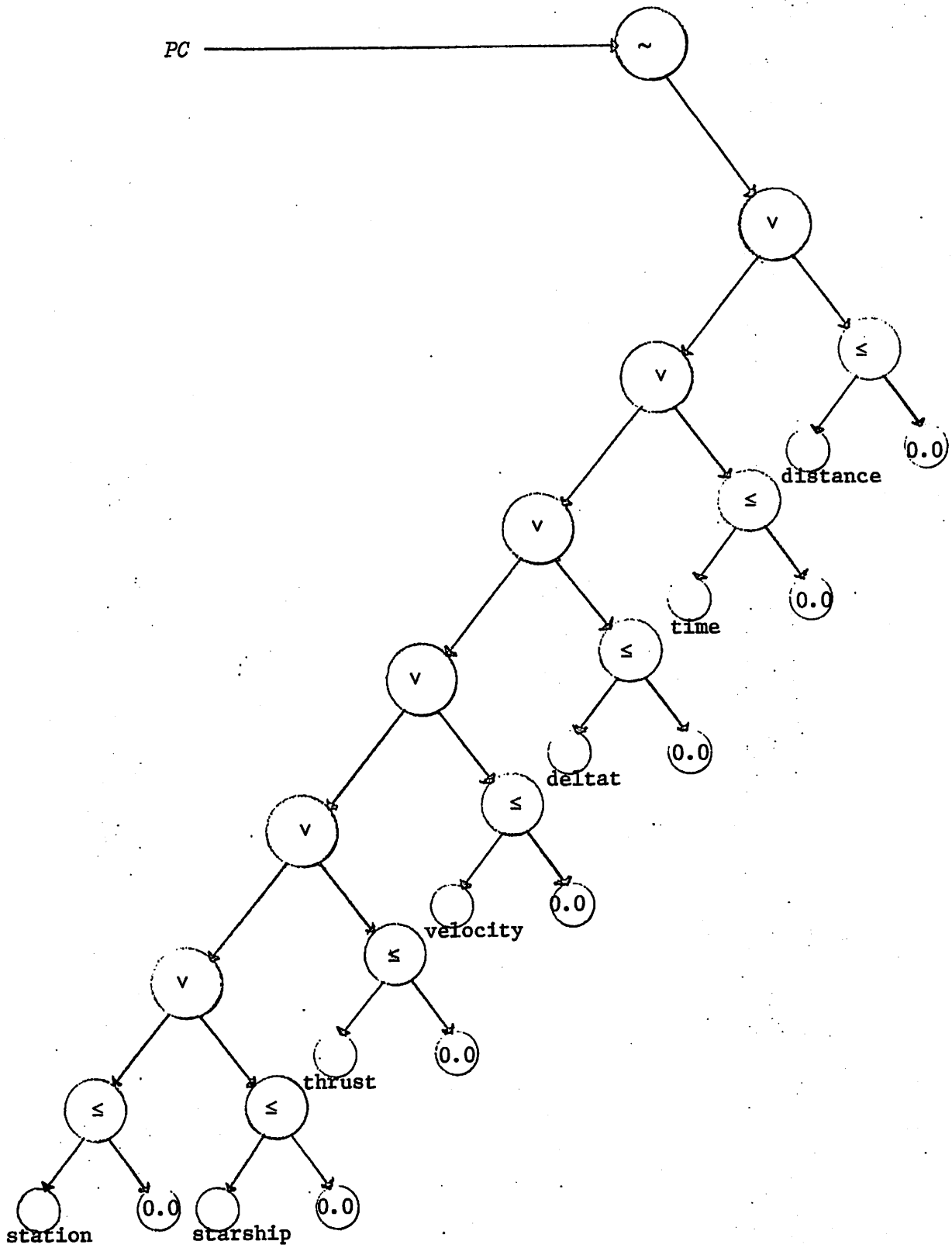


Figure 9

Computational Graph of Branch Predicate  $bp(n_s, n_2)[n_s, n_2]$

entry	operator	operand 1	operand 2
1	**	10	-11
2	*	6.67	CG1
3	*	CG2	station
4	*	CG3	starship
5	**	distance	2
6	/	CG4	CG5
7	/	thrust	starship
8	-	CG7	CG8
9	*	CG8	deltat
10	+	velocity	CG9
11	*	velocity	deltat
12	-	distance	CG11
13	+	time	deltat
14	*	CG8	deltat
15	+	CG10	CG14

Program Variables

TIME : CG13

GCONST : CG2

DISTANCE : CG12

GRAVITY : CG6

ERROR : 0

CONSTACC : CG8

STATION : station

CURRVEL : CG10

STARSHIP : starship

NEXTVEL : CG15

THRUST : thrust

VELOCITY : velocity

DELTAT : deltat

Figure 10

Tabular Representation of Computational Graph of figure 8f.

$n_f$	true
$bp(n_{11}, n_{12})$	$(NEXTVEL) \leq 0.0$
$n_{11}$	$(CURRVEL + CONSTACC*DELTAT) \leq 0.0$
$n_9$	$(NEXTVEL + CONSTACC*DELTAT) \leq 0.0$
$n_7$	$(CURRVEL + CONSTACC*DELTAT + CONSTACC*DELTAT) \leq 0.0$
$n_6$	$(VELOCITY + CONSTACC*DELTAT + CONSTACC*DELTAT) \leq 0.0$
$n_5$	$(VELOCITY + (GRAVITY-THRUST/STARSHIP)*DELTAT + (GRAVITY-THRUST/STARSHIP)*DELTAT) \leq 0.0$
$n_4$	$(VELOCITY + ((GCONST*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT + ((GCONST*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT) \leq 0.0$
$bp(n_2, n_4)$	$((VELOCITY + ((GCONST*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT + ((GCONST*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT) \leq 0.0) \wedge \sim(DISTANCE**2 = 0.0)$
$n_2$	$((VELOCITY + ((6.67*10**(-11))*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT + ((6.67*10**(-11))*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT) \leq 0.0) \wedge \sim(DISTANCE**2 = 0.0)$
$bp(n_s, n_2)$	$((VELOCITY + ((6.67*10**(-11))*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT + ((6.67*10**(-11))*STATION*STARSHIP/DISTANCE**2) - THRUST/STARSHIP)*DELTAT) \leq 0.0) \wedge \sim(DISTANCE**2 = 0.0) \wedge \sim(STATION \leq 0.0 \vee STARSHIP \leq 0.0 \vee THRUST \leq 0.0 \vee VELOCITY \leq 0.0 \vee DELTAT \leq 0.0 \vee TIME \leq 0.0 \vee DISTANCE \leq 0.0)$
$n_s$	$((velocity + ((6.67*10**(-11))*station*starship/distance**2) - thrust/starship)*deltat + ((6.67*10**(-11))*station*starship/distance**2) - thrust/starship)*deltat) \leq 0.0) \wedge \sim(distance**2 = 0.0) \wedge \sim(station \leq 0.0 \vee starship \leq 0.0 \vee thrust \leq 0.0 \vee velocity \leq 0.0 \vee deltat \leq 0.0 \vee time \leq 0.0 \vee distance \leq 0.0)$

NOTE: The final simplified PC would be the same as the final simplified PC shown in figure 6.

Figure 11  
Symbolic Execution Using Backwards Substitution

expansion technique can be used to create the symbolic expressions for backward substitution. Note that many of the statements, specifically those that do not modify variables for which values are input, can be ignored using backward substitution when only the path condition is desired. In the example of figure 11, statements 8, 10 and 12 are ignored. In a more general symbolic execution system where both the path condition and symbolic values are desired, the two approaches examine each statement and produce equivalent expressions for the *PC* and *VAL*. In systems which support early detection of nonexecutable paths, however, the forward expansion approach is more efficient. The rest of this section first describes several techniques for determining path condition consistency and then returns to the comparison between forward expansion and backward substitution.

In most cases, only a subset of the paths in a program are executable and, therefore, it is desirable to determine path condition consistency. 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 of a nonexecutable path. Moreover, it allows an alternative edge to be selected on a partial path whenever an inconsistent branch predicate is initially encountered. Thus, the partial path that has already been symbolically executed can usually be salvaged.

Using the notation introduced in section 1, whenever a partial path  $T_{ku}$  is augmented with a new node  $n_{k u+1}$ , the branch predicate  $s(bp(n_{ku}, n_{k u+1})[T_{ku}])$  is first simplified and then may be examined for consistency with the existing path condition  $PC[T_{ku}]$ . One of several algebraic manipulation systems [2, 4, 22] can be used to simplify the *PC* to a canonical form, so this aspect of the implementation will not be described further.

The branch predicate  $s(bp(n_{ku}, n_{ku+1})[T_{ku}])$  may either evaluate to a boolean constant (where the null branch predicate is considered to be the constant true), or it may be a symbolic expression in terms of the input variables. If the branch predicate is constant, consistency determination is immediate:  $PC \wedge \text{true} = PC$  and  $PC \wedge \text{false} = \text{false}$ . When the branch predicate is a symbolic expression over the input values (and the  $PC$  is not the constant true), it is necessary to use a more sophisticated technique for determining path condition consistency. One approach to this problem is to use standard theorem proving techniques. We refer to this as the axiomatic approach since it is based upon the axioms of predicate calculus. Another approach is to treat each conjunct in the  $PC$  as a constraint and to use one of several algebraic methods - such as a gradient hill climbing algorithm [11], linear programming [18] or a more brute force approach [20] - to solve the system of constraints. Both the axiomatic and algebraic approaches work well on the simple constraints that are generally created during symbolic execution [7]. No method, however, can solve all arbitrary systems of constraints [10]. In some instances, path consistency can not be determined. The symbolic execution of such a path can continue, but whether or not the path can be executed is suspect.

Whenever the last node  $n_{ku}$  in the partial path  $T_{ku}$  has only one successor node, the branch predicate is null and is represented by the constant true, which is always consistent with the existing  $PC$ . When there is more than one successor node, each successor node and its respective branch predicate are considered as alternatives to the current path. There are three cases to be considered: 1) none of the alternative branch predicates are consistent with the  $PC$ ; 2) only one of the alternative branch predicates is consistent with the  $PC$ ; and 3) more than one of the alternatives are consistent with the  $PC$ . The path shown in figure 12 demonstrates all three cases.



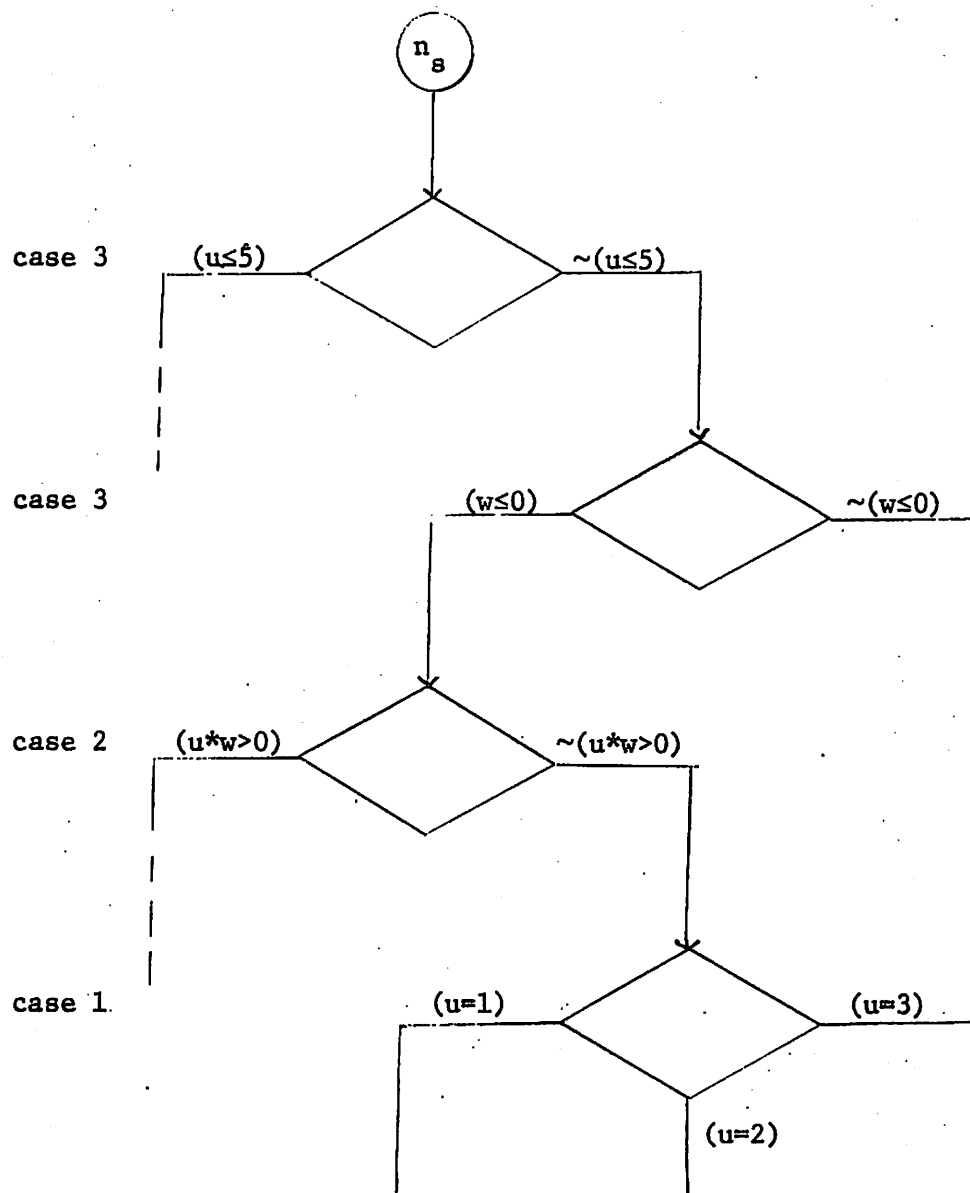


Figure 12

Path Demonstrating the 3 Cases  
for Branch Predicates

The first case only occurs when evaluating multi-conditional predicates like those that occur for computed GO TO statements or CASE statements without an otherwise clause. This case implies a program error.

The alternative branch predicates for a set of successor nodes either all evaluate to constant boolean values, or all evaluate to symbolic expressions involving the input variables. Cases 1 and 2 can occur in either of these situations. When all of the branch predicates evaluate to boolean constants, at most one of the alternatives can evaluate to true. Hence, case 3 can occur only when the branch predicates are symbolic expressions. Some symbolic execution systems will select the successor node in the first two cases and will give the user an option in selecting the path when the third case occurs.

Now to return to the comparison between forward expansion and backward substitution. Forward expansion is a more efficient technique of symbolic execution than backward substitution when *PC* consistency is determined at each branch point. Using forward expansion, the branch predicates maintain their original form once they are created and conjoined to the *PC*. The *PC* is, therefore, only modified by the conjunction of a new, simplified constraint. In backward substitution the *PC* is likewise modified by a new, simplified constraint but, in addition, the *PC* may be modified by any assignment statement on the path that changes the value of any variable referenced in the *PC*. In other words, the *PC* that is created using forward expansion only contains expressions in terms of the input variables while the *PC* that is created using backward substitution may reference intermediate variables that are later modified. The additional *PC* modification during backward substitution is costly since it also requires resimplification of the modified constraints and consistency must be checked after each simplification.

### 3.3 Applications

Symbolic execution systems have several interesting applications. This section will consider three applications: validation and documentation, error detection, and test data generation.

The symbolic expressions that are generated for a program path can quite naturally be used for validation and documentation. The expressions often provide a concise representation of the output produced along a path. These expressions can be used to document the program or can be examined for errors. The symbolic expressions describe the path functions  $(P_{H1}, P_{H2}, \dots, P_{HN})$  for the entire path domain  $D_H^P$ . Normal execution, on the otherhand, only provides particular output values  $(z_1, \dots, z_N)$  for particular input values  $(x_1, \dots, x_M)$ . It is possible for the output data to be correct while the path functions are incorrect. To use a trivial example, assume the intended function of a program path with one input value and one output value is  $F_G(v) = 2*v+3$  but the computed function of the path is  $P_G(v) = 3*v+3$ . If the program path is executed with  $v = 0$ , then the actual resulting value and intended value agree. Examination of the path function would quickly uncover the error. While not all errors would be this glaring or all symbolic expressions this short, examining the symbolic path functions is often useful in uncovering program errors [15]. This is a particularly beneficial feature for examining programs for scientific applications, where it is often extremely difficult to manually compute the intended result accurately due to the complexity of the computations and domain of the input data.

The path functions created during symbolic execution could be evaluated for particular data values. The result would be the same as if the path had been executed. (In cases where round off errors, overflow, or underflow could occur, there may be discrepancies. We do not address these types of problems

here.) The benefit of evaluation at this point is that the symbolic expressions for the path functions  $(p_{H1}, p_{H2}, \dots, p_{HN})$  and path domain  $D_H^P$  can be used to guide in the selection of input values. For example, boundary points of the path domain may be selected to check the correctness of the branch predicates [25]. Also, if a path function is a polynomial, examination of its degree can be used in selecting the number of test data points needed to determine the correctness of this function.

As work in the area of program specification advances, it may also be possible to compare the intended program function  $F$  with the program  $P$  using symbolic execution. Work on this approach to program testing is currently in progress [21].

Symbolic execution can also be actively applied to the detection of program errors. At appropriate points in the program, boolean conditions can be generated for certain predefined error conditions. These conditions can be evaluated and checked for consistency with the  $PC$  just as branch predicates are evaluated. Consistency implies the existence of input data in the path domain  $D_H^P$  that would cause the described error. Inconsistency implies that the error condition could not occur for any element in the input domain. This demonstrates another advantage of symbolic execution over normal program execution. Normal execution of a path may not uncover a run time error while symbolic execution of a path can detect the presence or guarantee the absence of some errors.

The ATTEST system [8] automatically generates constraints for certain error conditions whenever it encounters certain program constructs. For example, whenever a nonconstant divisor is encountered, a constraint is created comparing the symbolic value of the divisor to zero. This constraint is then temporarily conjoined to the  $PC$ . If the augmented  $PC$  is consistent,

then input data exists that would cause a division by zero error; an error report is issued. If the augmented  $PC$  is inconsistent, then this run time error could not occur for this division on this path. The division constraint is removed before symbolic execution continues.

Path verification of program assertions is another method of error detection. Instead of predefining the error condition, user created assertions define the error conditions. In general, the user asserts that a boolean predicate is true at a particular point in the program. An error exists if the predicate is not true. When the assertion is encountered during symbolic execution, the complement of the boolean predicate is evaluated and conjoined to the  $PC$ . The rest of the analysis is then identical to the implicit error detection described above.

Test data generation is another natural application of symbolic execution. The path condition is examined to determine a solution — that is, test data to execute the program path. Symbolic execution, like other methods of program validation, does not test the program in its natural environment. Evaluation of the path functions for particular input values return 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 program demonstrates its actual performance characteristics. Select [3] and ATTEST [8] 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, an algebraic method is used to solve the system of constraints in the  $PC$ . Additional work [25] is being done to further refine methods of selecting data within a path domain to increase the probability of detecting errors.

#### 4. Global Symbolic Execution

The goal of global symbolic evaluation [5] is the derivation of a global representation of the program- a representation of all program variables, for all the paths rather than along a specific path through the program. In other words, global symbolic evaluation results in a closed form representation of an entire program, independent of any particular path execution. In this section, we will describe the general method of global symbolic evaluation and explain in some detail the technique used in evaluating loops within a program.

##### 4.1 General Method

Global symbolic evaluation, like symbolic execution, analyzes the control flow graph of the program. The nodes in the graph are numbered such that if node  $n_i$  is a predecessor of node  $n_j$ , then  $i < j$ . To maintain this node ordering and since loops are handled separately by loop analysis, all backward branches are disregarded. The control flow graph in figure 2 has the appropriate node numbering for global symbolic evaluation of the procedure DOCKING. The numbering of the nodes in the control flow graph provides the order in which the statements will be symbolically evaluated.

As in symbolic execution, the input values are represented by symbolic names, and all program variables are represented as expressions in terms of those symbolic names throughout the analysis. The actual evaluation of a statement is performed by the same technique as that used in symbolic execution. Furthermore, the computations themselves are maintained in a form analogous to the computational graph created in symbolic execution. The two methods differ in the way in which conditional branching is analyzed.

In evaluating a particular node, symbolic execution only considers the program state of the one partial program path preceding the current node, whereas global symbolic evaluation considers the program state of all immediate predecessor nodes in the control flow graph. At any node in the graph, global symbolic evaluation maintains a representation of the state that describes the conditions and computations of all partial program paths reaching that node. This results in a conditional representation, or CASE statement, where each component of the CASE statement represents such a path. Furthermore, a partial program path may represent a class of paths which differ by the number of iterations of any loop on the path. Loop analysis develops these classes and will be explained in the next section. We will therefore refer to the program state of node  $n_i$  as  $STATE[n_i]$ , where the state may have several *PCs* associated with it and each *PC* has a corresponding *VAL*. The representation of the program state for global symbolic evaluation is shown in figure 4.

To see how a node is evaluated, consider a particular node  $n_k$ , with predecessor nodes  $n_i$  and  $n_j$  (which have been previously evaluated). Control may reach  $n_k$  via either of the edges  $(n_i, n_k)$  or  $(n_j, n_k)$ , and the transfer from either predecessor node occurs under the conditions of the corresponding branch predicate  $bp(n_i, n_k)$  or  $bp(n_j, n_k)$ . Thus, when  $n_k$  is evaluated, there are two possible symbolic *STATES* that are effective. The program state at node  $n_k$  is then a conditional symbolic expression provided by updating the *STATE* in the context of either possible transfer to the node. In the context of the transfer from predecessor node  $n_i$  to  $n_k$ ,  $STATE[n_k]$  is obtained by updating the *STATE* of node  $n_i$  in much the same way as the update is performed in symbolic execution. The branch predicate  $bp(n_i, n_k)$  is conjoined

to all the *PCs* associated with  $n_i$  and the conjunctions are checked for consistency. If any of the augmented *PCs* are inconsistent, the case is discarded from the updated *STATE*. Each remaining *PC's* *VAL* is updated in all components whose variables are modified by node  $n_k$ . The same procedure is followed for the transfer from  $n_j$  to  $n_k$  and these two *STATE* vectors form the conditional representation of the program state at node  $n_k$ .

#### 4.2 Loop Analysis

The described representation of the state of a node in terms of all partial program paths into the node is only possible because of the form in which global symbolic evaluation represents loops. Loop analysis attempts to represent a program loop with a closed form expression describing the effects of that loop. By doing this, paths which differ only by the number of iterations of a loop are represented by one path.

Given a loop, global symbolic evaluation develops expressions for the values of all variables modified within the body of the loop in terms of the symbolic input values and a symbolic iteration count for the loop. In addition, a conditional expression is obtained representing the actual number of iterations of the loop that will be performed for any arbitrary execution of the program, for any arbitrary assignment to the input values.

Loop analysis begins by associating with the loop an iteration counter  $k$ . For each variable  $v$  whose value may change within the loop, a special symbolic value  $v_k$  is used to represent the value of the variable  $v$  at the beginning of the  $k$ th iteration of the loop. Symbolic evaluation of the loop body is then performed in much the same manner as the forward expansion technique of symbolic execution. This "execution" provides the symbolic value of the variable  $v$  at the end of the  $k$ th iteration, under the assumption of another



iteration of the loop. This symbolic value is, alternatively, the variable's value at the beginning of the  $k+1$ st iteration of the loop,  $v_{k+1}$ . The global symbolic evaluation outside the loop determines the initial value,  $v_1$ , of the variable just prior to the first iteration of the loop. The symbolic expressions  $v_k$  and  $v_{k+1}$  provide a recurrence relation with the boundary value  $v_1$ . The solution to the recurrence relation, which will be represented by  $v(k)$ , is the value of the variable  $v$  upon exit from the  $k$ th iteration of the loop.

In addition to determining this representation for the variables modified within the loop, the closed form representation of a loop must contain a conditional expression for the number of times the loop is performed. Each condition under which the loop will be exited is singled out; these are the branch predicates that control any transfer to a point outside of the loop body. Each condition is, in general, some constraint on variables modified within the loop (otherwise it would not control exit from the loop). These branch predicates can, therefore, be evaluated over the values of the modified variables at the beginning of the  $k$ th iteration - that is, over the solutions to the recurrence relations,  $v(k)$ . This produces a symbolic representation for each exit condition as a function of the general iteration number,  $k$ . The number of the iteration before which exit occurs, call it  $k_L$ , is the minimum  $k$ ,  $k \geq 0$ , such that one of the exit conditions is true.

The loop may then be represented in its closed form by:  $k_L$ , the conditional expression for the number of times the loop will be executed; and  $v(k_L)$ , the symbolic value of variable  $v$  after  $k_L$  iterations of the loop, for each variable  $v$  which is modified within the loop. Figure 13 shows the analysis performed for the loop in the procedure DOCKING of figure 1.

$$\begin{aligned} \text{DISTANCE}_{k+1} &= \text{DISTANCE}_k - \text{CURRVEL}_k * \text{deltat} \\ \text{CURRVEL}_{k+1} &= \text{NEXTVEL}_k \\ \text{TIME}_{k+1} &= \text{TIME}_k + \text{deltat} \\ \text{NEXTVEL}_{k+1} &= \text{CURRVEL}_{k+1} + (6.67*10^{**(-11)}* \text{station}* \text{starship} / \text{distance}^{**2} \\ &\quad - \text{thrust} / \text{starship}) * \text{deltat} \end{aligned}$$

Initial values

$$\begin{aligned} \text{DISTANCE}_1 &= \text{distance} \\ \text{CURRVEL}_1 &= \text{velocity} \\ \text{TIME}_1 &= \text{time} \\ \text{NEXTVEL}_1 &= \text{velocity} + (6.67*10^{**(-11)}* \text{station}* \text{starship} / \text{distance}^{**2} \\ &\quad - \text{thrust} / \text{starship}) * \text{deltat} \end{aligned}$$

Solutions to recurrence relations

$$\begin{aligned} \text{TIME}(k) &= \text{time} + \sum_{i=2}^k \text{deltat} \\ \text{NEXTVEL}(k) &= \text{velocity} + \sum_{i=1}^k [(6.67*10^{**(-11)}* \text{station}* \text{starship} / \text{distance}^{**2} \\ &\quad - \text{thrust} / \text{starship}) * \text{deltat}] \\ \text{CURRVEL}(k) &= \text{velocity} + \sum_{i=2}^k [(6.67*10^{**(-11)}* \text{station}* \text{starship} / \text{distance}^{**2} \\ &\quad - \text{thrust} / \text{starship}) * \text{deltat}] \\ \text{DISTANCE}(k) &= \text{distance} - \sum_{j=2}^k [(\text{velocity} + \sum_{i=2}^j [(6.67*10^{**(-11)}* \text{station}* \text{starship} \\ &\quad / \text{distance}^{**2} - \text{thrust} / \text{starship}) * \text{deltat}]) * \text{deltat}] \end{aligned}$$

Simplified solutions

$$\begin{aligned} \text{TIME}(k) &= \text{time} + (k-1) * \text{deltat} \\ \text{NEXTVEL}(k) &= \text{velocity} + (k) * (6.67*10^{**(-11)}* \text{station}* \text{starship}^{**2} * \text{deltat} \\ &\quad - \text{thrust} * \text{distance}^{**2} * \text{deltat}) / \text{distance}^{**2} * \text{starship} \\ \text{CURRVEL}(k) &= \text{velocity} + (k-1) * (6.67*10^{**(-11)}* \text{station}* \text{starship}^{**2} * \text{deltat} \\ &\quad - \text{thrust} * \text{distance}^{**2} * \text{deltat}) / \text{distance}^{**2} * \text{starship} \\ \text{DISTANCE}(k) &= \text{distance} + \sum_{j=2}^k [\text{velocity} * \text{deltat} + (j-1) * (6.67*10^{**(-11)}* \\ &\quad \text{station}* \text{starship}^{**2} * \text{deltat}^{**2} - \text{thrust} * \text{distance}^{**2} * \text{deltat}^{**2}) \\ &\quad / \text{distance}^{**2} * \text{starship}] \end{aligned}$$

Exit condition

$$\text{NEXTVEL}(k) \leq 0.0$$

Evaluated exit condition

$$(\text{velocity} + (k) * (6.67*10^{**(-11)}* \text{station}* \text{starship}^{**2} * \text{deltat} \\ - \text{thrust} * \text{distance}^{**2} * \text{deltat}) / \text{distance}^{**2} * \text{starship}) \leq 0.0$$

Number of iterations of loop,  $k_L$ 

$$\begin{aligned} k_L &= \text{minimum } k, \text{ such that } k \geq 0 \text{ and } (\text{velocity} + (k) * (6.67*10^{**(-11)}* \\ &\quad * \text{station}* \text{starship}^{**2} * \text{deltat} - \text{thrust} * \text{distance}^{**2} * \text{deltat}) / \text{distance}^{**2} \\ &\quad * \text{starship}) \leq 0.0 \end{aligned}$$

Figure 13

Obtaining the recurrence relation  $v(k)$  is not always straightforward. Complications arise in several situations. When there are simultaneous recurrence relations, several variables, which may be dependent, are modified within the loop. In particular, the dependence may be cyclic;  $v$  may depend on  $w$ , which depends on  $v$ . Problems are also caused when the recurrence relations are conditional, in which case the closed form solution becomes quite complicated, provided it can be solved at all.

When a closed form representation of a loop can be found by this analysis technique, it provides a more general evaluation of a loop than the technique employed by symbolic execution systems - evaluating the loop for a specific number of iterations. There is no reason, however, that this loop analysis technique could not also be incorporated into symbolic execution.

After the loop has been analyzed, the closed form representation becomes part of the program state at the point where the loop is exited, and evaluation continues. Figure 14 shows the program state following global symbolic evaluation of the procedure DOCKING, where the conditions and functions have been simplified.

#### 4.3 Applications

Global symbolic evaluation has several possible applications, many of which are similar to those of symbolic execution. Test data generation could conceivably be performed by solving for the  $PC$ s in the CASE statement. New methods for solving a  $PC$  must be explored since the  $PC$  may contain recurrence relations as well as constraints. The closed form representation of a program could be compared with some types of program specifications to determine consistency. User-provided assertions can be checked for

case

```

station ≤ 0.0 ∨ starship ≤ 0.0 ∨ thrust ≤ 0.0 ∨
velocity ≤ 0.0 ∨ deltat ≤ 0.0 ∨ time ≤ 0.0 ∨
distance ≤ 0.0 :

```

```

    TIME = time
    DISTANCE = distance
    ERROR = 1
    STATION = station
    STARSHIP = starship
    THRUST = thrust
    VELOCITY = velocity
    DELTAT = deltat
    GCONST = Λ
    GRAVITY = Λ
    CONSTACC = Λ
    CURRVEL = Λ
    NEXTVEL = Λ

```

```

station > 0.0 ∧ starship > 0.0 ∧ thrust > 0.0 ∧
velocity > 0.0 ∧ deltat > 0.0 ∧ time > 0.0 ∧
distance > 0.0 :

```

```

    TIME = time + (kL-1)*deltat
    DISTANCE = distance - Σj=2kL [(velocity*deltat + (j-1) * (6.67*10**(-11))
        *station*starship**2*deltat**2 - thrust*distance**2*deltat**2)
        / distance**2*starship]
    ERROR = 0
    STATION = station
    STARSHIP = starship
    THRUST = thrust
    VELOCITY = velocity
    DELTAT = deltat
    GCONST = 6.67*10**(-11)
    GRAVITY = 6.67*10**(-11)*station*starship/distance**2
    CONSTACC = (6.67*10**(-11)*station*starship**2 - thrust*distance**2)/
        distance**2*starship
    CURRVEL = velocity + (kL-1)*(6.67*10**(-11)*station*starship**2*deltat
        - thrust*distance**2*deltat) / distance**2*starship
    NEXTVEL = velocity + (kL) * (6.67*10**(-11)*station*starship**2*deltat
        - thrust*distance**2*deltat) / distance**2*starship

```

endcase

Figure 14

Program State Following Global Symbolic Evaluation of DOCKING

validity; with global symbolic evaluation, the truth of these assertions can be checked for all paths rather than a specific path. In addition, since the program state is maintained at all points in the program, assertions could be provided by the user after completion of global symbolic evaluation without requiring reevaluation of the program. Similarly, global symbolic evaluation can be used to automatically generate and check error conditions as it analyzes a program.

Global symbolic evaluation also has applications in program optimization [24]. As in optimizing compilers, the existence of the computational graph [9] makes common subexpression elimination and constant folding relatively straightforward. In addition, several types of loop optimizations may often be performed when the closed form representations of loops are obtainable. Loop-invariant computations may be easily detected since they are independent of the iteration count of the loop; these may thus be moved outside of the loop. Loop fusion can sometimes be performed when the number of iterations performed by two loops can be determined to be the same, and variables manipulated in the second loop are not computed in a later iteration of the first loop. When variables modified within the loop have values that form arithmetic progressions - that is, they are incremented by the same amount each time through the loop - these computations can sometimes be moved out of the loop and replaced by expressions in terms of the iteration count. Optimizations that perform in-line substitution of a procedure may also be benefited by global symbolic evaluation, since the closed form representation of the procedure may enable better determination of when such substitution is useful.

## 5. Dynamic Symbolic Execution

Dynamic symbolic evaluation is just one of the features provided in dynamic testing systems [1,12]. Using test data to determine the path, dynamic symbolic evaluation systems provide symbolic representations of the executed path's computations. This section gives a brief overview of dynamic testing systems and then describes dynamic symbolic evaluation, its implementation techniques and its applications.

### 5.1 General Method

Dynamic testing systems monitor program behavior during execution. This is implemented by instrumenting the program - that is, by inserting calls to analysis procedures in appropriate places in the code. This is generally done by a preprocessor and may double the number of statements in the source program. The user then supplies input data to execute the instrumented program.

Dynamic testing systems may provide a profile of each execution run as well as an accumulated profile of all execution runs. Some of the types of information in a profile include the number of times each statement was executed, the minimum and maximum number of times each loop was traversed, the minimum and maximum values assigned to variables, and the paths that were executed. In addition, the system may check the validity of user assertions at run time [12,23]. Unlike the assertion checking done by symbolic execution systems, dynamic assertion checking is done just for the supplied input data and not for the whole path. Either the assertion is true and thus valid for the input data, or the assertion is false, and thus invalid for the program.

The dynamic symbolic evaluation component of dynamic testing provides a symbolic representation of the computations of each executed path. For input data that executes path  $P_H$ ,  $VAL[P_H]$  is provided.  $VAL[P_H]$  can be represented internally as a computational graph. This computational graph would be similar to the computational graph for symbolic execution but it would be augmented to include the value produced at each node. The computational graph can be created using the forward expansion method described in section 3.2.

At the end of the path, the expression for each output variable is shown. Generally, dynamic evaluation systems display the expressions as trees instead of mathematical expressions, though both or either form could be displayed. Using the DOCKING procedure and the same path as that in figure 6, the computational trees are shown in figure 15, while figure 16 shows the output that might be produced by dynamic symbolic evaluation using the format of figure 5.

Existing dynamic symbolic evaluation systems are only concerned with the  $VAL$  component of the program state. Since the input values are known, each branch predicate evaluates to the constant value true (or a run time error is encountered). The  $PC$  is, therefore, equal to true. It would be easy to extend dynamic symbolic evaluation to include symbolic representations of the  $PC$ . Note, the path functions are represented symbolically though all computations evaluate to a numeric value. The  $PC$  provides valuable information by defining the path domain even though it is not necessary to check for path consistency. Examination of the  $PC$ , like examination of the  $VAL$ , may uncover program errors. An erroneous  $PC$  would imply an erroneous branch predicate or erroneous calculation affecting the branch predicate.

Input Values

STATION =  $4.0 \times 10^{14}$     DELTAT = 10.0  
 STARSHIP = 100.0    TIME = 120530.0  
 THRUST = 4500.0    DISTANCE = 2000.0  
 VELOCITY = 75.0

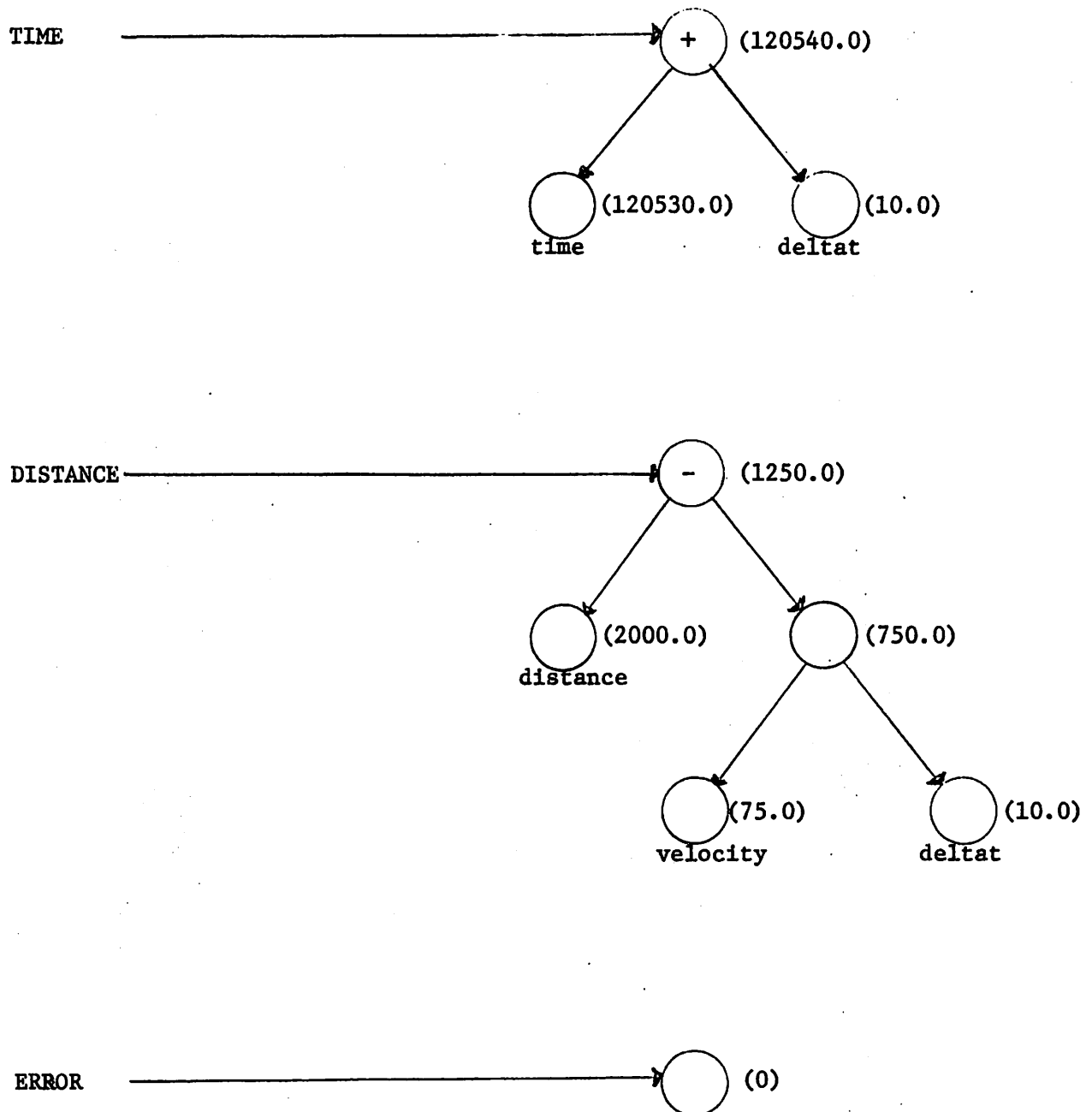
Output VariablesComputational Trees

Figure 15

Computational Trees for  
 Dynamic Symbolic Evaluation



## STATEMENTS EXECUTED

 $n_s, n_2, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}, n_{11}, n_{12}, n_f$ 

## SYMBOLIC AND ACTUAL VALUES OF OUTPUT VARIABLES

TIME = time + deltat = 120540.

DISTANCE = distance - velocity \* deltat = 1250.

ERROR = 0.

Figure 16

Final Output Produced for Dynamic Symbolic Evaluation

## 5.2 Applications

The primary application of dynamic symbolic evaluation is program debugging. When an error is uncovered in a program, dynamic symbolic evaluation provides a picture of the resulting computations. Examination of symbolic representations of the path functions and path condition often helps to isolate the cause of an error. To assist in debugging, these systems provide a capability for examining the computational graph while it is being constructed. Program execution can be followed statement by statement. These systems also allow the user to back up execution. In other words, the user can direct the system to execute the path backwards and thus undo the computational graph to help the user isolate the error. Experiments with ISMS have shown that both forward and backward execution are beneficial for debugging [12]. Note that backward execution is not the same as the backward substitution technique described in section 3.2. In backward execution at least part of the path has already been executed and the corresponding part of the computational graph has been built. Dynamic symbolic evaluation requires an implementation method somewhat similar to forward expansion since input data is used to determine the path, thus implying a forward analysis approach.

## 6. Conclusion

In this paper, three methods of symbolic program analysis have been described. All three methods represent a program's computations and input domains by symbolic expressions in terms of the input values, though the methods differ in their scope of representation.

Dynamic symbolic evaluation is the most restrictive method of the three. Using input data to determine a path, dynamic symbolic evaluation represents the path functions. Since input data is used to select the path, the path condition evaluates to true. Though the path condition can be described symbolically, there is no need to test for path consistency. With no simplification of the path conditions nor path consistency determination necessary, the implementation of dynamic symbolic evaluation is straightforward. The major application of this method is program debugging.

Symbolic execution systems are not dependent on input data to determine the path as dynamic symbolic evaluation is, but rather can analyze any specified program path. Symbolic execution systems represent the path functions and path condition. Since many program paths are not executable, symbolic execution tries to determine path condition consistency. The most efficient symbolic execution systems use a forward expansion technique of implementation and determine path condition consistency whenever a new branch predicate is conjoined to the existing path condition. In general, path condition consistency can not always be determined. In practice, consistency can often be determined using one of several existing techniques. In addition, there is work currently being done on improving methods of solving arbitrary systems of constraints.

There are several interesting applications of symbolic execution in the area of program validation including automatic error detection, test data generation, and determination of consistency with program specifications.

Global symbolic evaluation has the widest scope of analysis; it attempts to represent the total program function by a symbolic expression. Since there may be an effectively infinite number of paths in a program, this method requires more sophisticated analysis than the mere conjunction of the symbolic expressions for each path. Instead a technique of loop analysis is used that attempts to represent each program loop in a closed form dependent on an arbitrary loop iteration count. While this approach can successfully analyze several types of loops, additional work is needed in this area. By using a closed form representation for each loop, the computations for a set of paths and their respective domains can be represented. Each such representation is one case in the conditional program representation provided by global symbolic evaluation. Path condition consistency still must be determined for each case in this conditional representation, but now this process is even further complicated by the presence of recurrence relationships describing each loop. This is another area in need of further research.

Dynamic symbolic evaluation is a well understood process that has been implemented in two dynamic testing systems. Symbolic execution has also been successfully implemented though there are still several implementation problems to be examined as well as several areas of research to be explored. Global symbolic evaluation is a relatively new method with prospective applications in the areas of program validation

and program optimization. Its applicability in the future will most likely depend on its success in loop analysis and consistency determination.

## REFERENCES

- [1] R.M. Balzer, "EXDAMS--Extendable Debugging and Monitoring System," in 1969 Spring Joint Computer Conf., AFIPS Conf. Proc., Vol. 34. Montvale, New Jersey, AFIPS Press, 1969, pp. 576-580.
- [2] R. Bogen, "MACSYMA Reference Manual," The Mathlab Group, Project MAC, Massachusetts Institute of Technology, 1975.
- [3] R.S. Boyer, B. Elspas, and K.N. Levitt, "SELECT-A Formal System for Testing and Debugging Programs by Symbolic Execution," in Proc. Int. Conf. Reliable Software, April 1975, pp. 234-244.
- [4] W.S. Brown, Altran User's Manual, Bell Telephone Lab., Vol. 1, 1973.
- [5] T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs," to appear in IEEE Trans. on Software Engineering.
- [6] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. on Software Engineering, September 1976, pp. 215-222.
- [7] L.A. Clarke, "Test Data Generation and Symbolic Execution of Programs as an Aid to Program Validation," Thesis, University of Colorado, 1976.
- [8] L.A. Clarke, "Automatic Test Data Selection Techniques," Infotech State of the Art Report on Software Testing, September 1978.
- [9] J. Cocke and J.T. Schwartz, Programming Languages and Their Compilers, New York University, Courant Inst. Math. Sci.
- [10] M. Davis, "Hilbert's Tenth Problem is Unsolvable," American Math. Mon., Vol. 80, March 1973, pp. 233-269.
- [11] B. Elspas, M. Green, A. Korsak, and P. Wong, "Solving Non Linear Inequalities Associated with Computer Program Paths," Stanford Res. Inst., preliminary draft.
- [12] R.E. Fairley, "An Experimental Program-Testing Facility," IEEE Trans. on Software Engineering, December 1975, pp. 350-357.
- [13] W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Trans. on Software Engineering, July 1977, pp. 266-278.
- [14] W.E. Howden, "Methodology for the Generation of Program Test Data," IEEE Trans. Comput., Vol. C-24, May 1975, pp. 554-559.
- [15] W.E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Trans. on Software Engineering, September 1976, pp. 208-215.

- [16] J.C. Huang, "An Approach to Program Testing," ACM Computing Surveys, September 1975, pp. 113-128.
- [17] J.C. King, "Symbolic Execution and Program Testing," CACM, July 1976, pp. 385-394.
- [18] A.H. Land and S. Powell, FORTTRAN Codes for Mathematical Programming, John Wiley & Sons, 1973.
- [19] E.F. Miller and R.A. Melton, "Automated Generation of Test Case Datasets," in Proc. Int. Conf. Reliable Software, April 1975, pp. 51-58.
- [20] C.V. Ramamoorthy, S.F. Ho and W.T. Chen, "On the Automated Generation of Program Test Data," IEEE Trans. on Software Engineering, December 1976, pp. 293-300.
- [21] D.J. Richardson, "Theoretical Considerations in Testing Programs by Demonstrating Consistency with Specifications," Workshop on Software Testing and Test Documentation, Florida, December 1978, pp. 19-56.
- [22] 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, TR-78-16.
- [23] L.G. Stucki, "Automatic Generation of Self-Metric Software," in Rec. 1973 IEEE Symp. Software Reliability, pp. 94-100.
- [24] J.A. Townley, "The Harvard Program Manipulation System," Center for Research in Computing Technology, Harvard University, TR-23-76.
- [25] L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," Workshop on Software Testing and Test Documentation, Florida, December 1978, pp. 335-354.