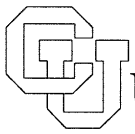
A System to Generate Test Data and Symbolically Execute Programs*

Lori Clarke

CU-CS-060-75 February 1975



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was supported in part by NSF Grant GJ 36461

ABSTRACT

This paper describes a system that is currently being implemented to generate test data for programs written in ANSI FORTRAN. Given a path, the system symbolically executes the program and creates a set of constraints on the program's input variables. If the set of constraints is linear, linear programming techniques are employed to solve the system of inequalities. A solution to the system of constraints is then a set of test data that will drive execution down the given path. If it can be determined that the set of constraints is inconsistent then the given path is shown to be nonexecutable. To increase the chance of detecting some of the more common programming errors, artificial constraints are temporarily created that simulate the error condition and an attempt is made to solve the augmented set of constraints. A symbolic representation of the program's output variables in terms of the program's input variables is also created. The symbolic representation is in a human readable form that facilitates error detection as well as being a possible aid in allegation generation and automatic program documentation.

1. Introduction

There is a growing interest in automated aids to program testing and verification. This is becoming more important as software development costs rise, particularly in the area of testing and debugging, and as more and more incorrect programs are disseminated.

The usual approach to program testing relies solely on the intuition of the programmer. The programmer generates data to test the program until satisfied that the program is correct. The success of this method depends on the expertise of the programmer and the complexity of the program. Experience has shown that this approach to program testing is inadequate and costly.[1] Consequently, several alternative approaches are being studied. Two of the most active approaches are program correctness and program validation.

The program correctness method challenges the saying that a program always has one more bug. In this method, assertions are inserted at transition points in a program and formal theorem proving techniques are employed to verify the correctness of the assertions at these points. It is a difficult task, however, to create the assertions. Because of this difficulty, program correctness methods are not usually applied to large programs where analysis is most needed.[1]

The program validation approach assures that the program has been analyzed and has passed a rigorous set of tests. Program validation guarantees that the subject program (program being analyzed) is without certain types of errors. Analysis of this type might include such things as extensive syntax checking, data flow analysis, graph analysis, and run time tests of all executable statements.

This paper describes a system that is currently being implemented to aid in program validation. This system has the following capabilities:

1) Generates test data to drive execution down a program path. Test data generation is a useful tool in program validation. Automatically generating input data for a comprehensive set of program paths and then executing the program with the generated input data, assures the user that the code has been well tested. Program analysis of this type should alleviate the problem of program errors occurring in running programs in segments of code that have never been tested before.
2) Detects nonexecutable program paths. Not all program paths are executable and, therefore, the system attempts to recognize nonexecutable paths. Detection of executable and nonexecutable paths is of value in analyzing programs as will be demonstrated below.
3) Creates symbolic representations of the programs output variables as functions of the programs input variables. Symbolic representations of the output variables aid in program validation by concisely representing a path's computations. The symbolic representation is in a human readable form that facilitates error detection as well as being a possible aid in allegation generation and automatic program documentation.
4) Detects certain types of program errors. To further aid in program validation, an attempt is made to generate data that will detect some of the more common run time errors, such as subscripts that are out of bounds.

The effectiveness of this system in validating that a subject program has been adequately tested depends on the selection of a comprehensive set of paths. In general, it is impossible to exercise all paths of a program since there may exist an infinite number. Therefore, a subset of paths must be selected. It has been suggested that a minimum comprehensive set of paths should execute every statement of the code at least once.[2] It has also been suggested that not only should every statement be executed but every branch condition as well.[3] In Figure 1, for example, all the statements could be executed without the true condition of statement 7 ever being tested. When all branch conditions are followed, then all loops are traversed at least once and, therefore, this seems to be a more thorough approach.

The test data generation system can also be used in conjunction with other validation projects. Often these projects are concerned with only a few paths that meet specific criteria. For example, the DAVE project, a data flow analysis system,[4] automatically detects paths that contain references to uninitialized variables. The test data generation system can be used to determine if any of these paths are executable.

2. Definitions

The flow of control in a program can be represented by a directed graph called the control flow graph. Γ is used to denote the successor operator. Thus, Γx denotes the set of nodes joined to the node x by edges directed from x. The nodes in the graph are called basic blocks. A basic block is defined as a maximal sequence of statements having the

```
 1              SUBROUTINE SUB(X,Y)
 2              X=X+1
 3              IF(X.GT.Y)GO TO 20
 4              X=Y-X
 5              GO TO 25
 6           20 X=X-Y
 7           25 IF(X.GT.-1)GO TO 30
 8              X=-X
 9           30 RETURN
10              END
```

Figure 1

property that whenever any one of the statements in the basic block is executed, every statement in the basic block is executed.

The control flow graph is assumed to have one entry point x_0 (a node with indegree zero) and one or more exit points (nodes with outdegree zero). A control path is a sequence of nodes $x_{i1}, x_{i2},\ldots,x_{in}$ where $x_{ij+1}\varepsilon\Gamma x_{ij}$ and $x_{i1}=x_0$. Not every control path can be executed. An execution path is a control path which can be executed. A non-executable or infeasible path is a control path which cannot be executed.

A program input variable is a variable that receives a value by means of some form of external communication. A program output variable returns a value by some form of external communication. External communication can occur in input and output statements and, if the calling program is not being analyzed in the parameter and common variables as well.

3. Overview

The system attempts to generate test data to execute the control paths of the subject program. Programs with loops may have an infinite number of control paths, and only a small set of these may be of interest to the user. Hence, this analysis program requires that the path to be analyzed be specified by the user. The user has a choice between two modes of operation, static or interactive. In the static mode, a path is specified initially by a sequence of basic blocks. The static mode is well suited to handle paths that are automatically generated. The interactive mode is more human oriented and queries the user at each conditional branch about which is to be the next basic block in the path.

In order to generate test data for a control path the variable relationships that affect the program flow must be determined. These variable relationships can be expressed as a set of constraints in terms of the program's input variables. To generate the constraints the path is symbolically executed. When a path is symbolically executed values are not assigned to variables as in a normal execution but instead expressions denoting the evolution of the variables are assigned. For example, in Figure 1, the variable representation of variable X after statements 1, 2, 3, and 4 are symbolically executed is X = I2-I1-1 where I1 and I2 denote the input values of parameters X and Y, respectively. The constraint created by the above control path is $I1 + 1 \leq I2$.

Whenever a conditional transfer of control is encountered one or more constraints, representing the branch in the conditional statement that is chosen, is generated. The constraint is then passed to an inequality solver to check if it is consistent with the existing constraints. If the constraint is inconsistent the path is infeasible. If the constraint is consistent the symbolic execution of the path continues. At the end of the path, the solution set found by the inequality solver is a data set that will force execution of the designated path.

For example in Figure 1, the path through statements 1, 2, 3, 4, 5, 7, and 9 (control paths are denoted here by statements instead of basic blocks for clarity) is associated with the following set of constraints.

$$I1 + 1 \leq I2$$

$$I2 - I1 - 1 > -1$$

One possible solution to the set of constraints is I1 = 0, I2 = 1. If the user were to call subroutine SUB with actual parameters 0 and 1 this path would be executed. The path through statements 1, 2, 3, 6, 8 and 9 is associated with the following set of constraints.

$$I1 + 1 > I2$$

$$I1 + 1 - I2 \leq -1$$

This set of constraints is inconsistent and the designated control path is therefore infeasible.

During the symbolic execution of the path whenever an output variable is used to communicate with the external environment the symbolic representation of the variable is returned to the user instead of a value. The symbolic representation can be used to detect errors in the program. By examining the symbolic representations of the output variables, computation errors can often be detected and can sometimes supply hints as to where the error in the computation occurred. Since the symbolic representation models the path's computations it can also be used for automatic program documentation and allegation generation.

There has been other current research in the area of symbolic execution and test data generation. Both Howden[3] and Huang[5] have described a system for generating path constraints. Both use an approach that traces backward through the code to determine how the conditional statement evolved. This approach does not allow early detection of infeasible paths and would require multiple passes to analyze each iteration of a loop. Huang has completely eliminated loops from his possible test paths.

The SELECT system [6] developed at Stanford Research Institute generates test data and creates a symbolic representation of the output

variables for programs written in a subset of LISP. SELECT has limited capabilities in handling procedure calls. Also, the path constraints and output variable representations are currently represented as a LISP list as opposed to a human recognizable form.

King has implemented a system called EFFIGY[7,8] that symbolically executes a program path and creates a symbolic representation of the output variables. EFFIGY uses theorem proving techniques to verify the feasibility of a path but does not generate test data. EFFIGY accepts programs written in a subset of PL/1 that allows only integer values.

An interactive system has also been developed at TRW that aids the user in generating test data.[2] This system is more user dependent than the other systems mentioned here.

Both SELECT and EFFIGY analyze programs written in a special dialect of a language. The test data generation system described here analyzes programs written in ANSI FORTRAN. It is felt that a popular user language poses a wider range of problems (such as communication between procedures and equivalencing of variable names). Also, analysis of programs written in a popular language should be a more realistic test for theoretical ideas. FORTRAN was chosen because it is a commonly used language and there is readily available a large source of programs in need of validation. The methods described in this paper, however, are applicable to other languages. In fact, the system translates the source language into an intermediate code before any program analysis is attempted.

4. Structure of the Analysis Program

Figure 2 depicts the overall flow of the analysis program. The data generation system is currently being used as an extension to a validation system, DAVE, which performs data flow analysis for the purpose of validation and error detection.[4] During a lexical analysis scan DAVE translates the subject program into a list of tokens. DAVE also creates a data base of information about each program unit. Each data base contains a symbol, common, and label table similar to tables usually constructed by compilers, a statement table which describes the input and output variables for each statement and a basic block table that represents the control flow graph. The token list and tables described above are also used in the data generation system. In addition, the program unit data bases and data generation tables are accessed using a data base package designed to facilitate data base restructuring.[9]

The data generation system can run independently of DAVE and in fact does not use the more sophisticated capabilities of that system. A preprocessor could be implemented that would simply build the needed tables and perform none of the analysis.

4.1 Intermediate Code Phase

Before the subject program is analyzed the token list is translated into an intermediate code similar to an assembly language. The intermediate code for the statements of a basic block is stored in a doubly linked list that is attached to the basic block node of the control flow

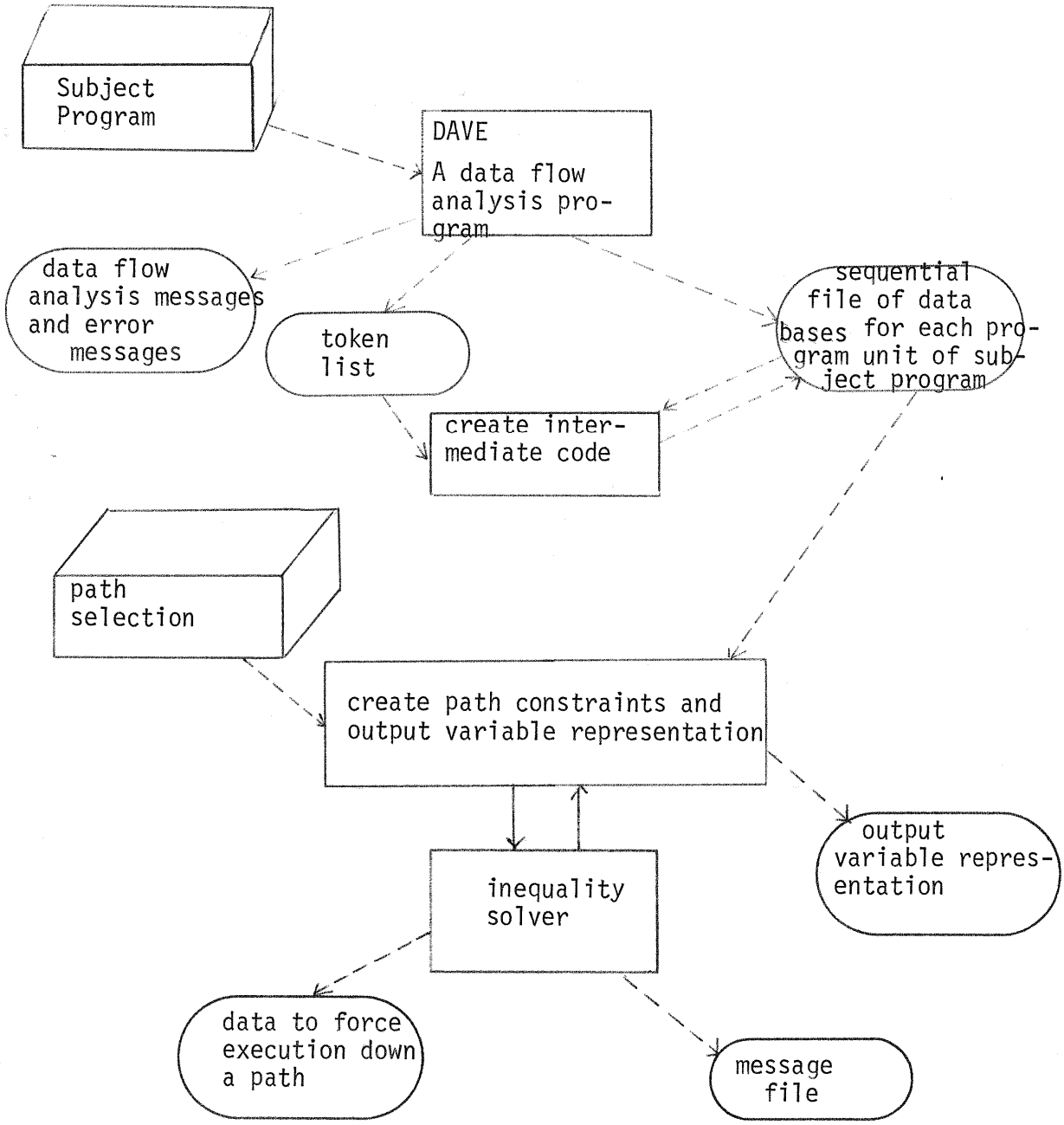

Subject Program
DAVE
A data flow analysis program
data flow analysis messages and error messages
token list
sequential file of data bases for each program unit of subject program
create intermediate code
path selection
create path constraints and output variable representation
output variable representation
inequality solver
data to force execution down a path
message file

Figure 2

graph (which is an entry in the basic block table of the program unit's data base). Intermediate code representing a conditional statement is attached to the corresponding edge of the graph. For example, in a logical if statement of the form IF(EXPRESSION)STATEMENT, the intermediate code representing the expression is attached to the edge that is followed if the evaluation of the expression is true. Code representing the complement of the expression is attached to the edge that is followed if the evaluation of the expression is false. An example of the code and control flow graph for the subroutine in Figure 1 is shown in Figure 3.

Representing the subject program in an intermediate form has several advantages. First, as was noted, it allows the analysis to be easily adapted to other languages. The analysis depends on the intermediate code. Any new language would have to be translated into the intermediate code but then few other modifications would be necessary to the test data generation system. Second, since all expressions are represented as a series of binary operations, it is easy to fold constants and simplify the variable representation during the analysis. Finally, the code is stored as a doubly linked list to enable future optimization and parallelization of the code.

4.2 Path Selection

The user has a choice between two methods of designating a path, static or interactive. The static method is designed to accept automatically generated paths while the interactive method is designed to aid a human user in selecting a path.

In the static mode, a path is designated by a sequence of subprogram names, basic block numbers, and loop counts. The syntax for describing a

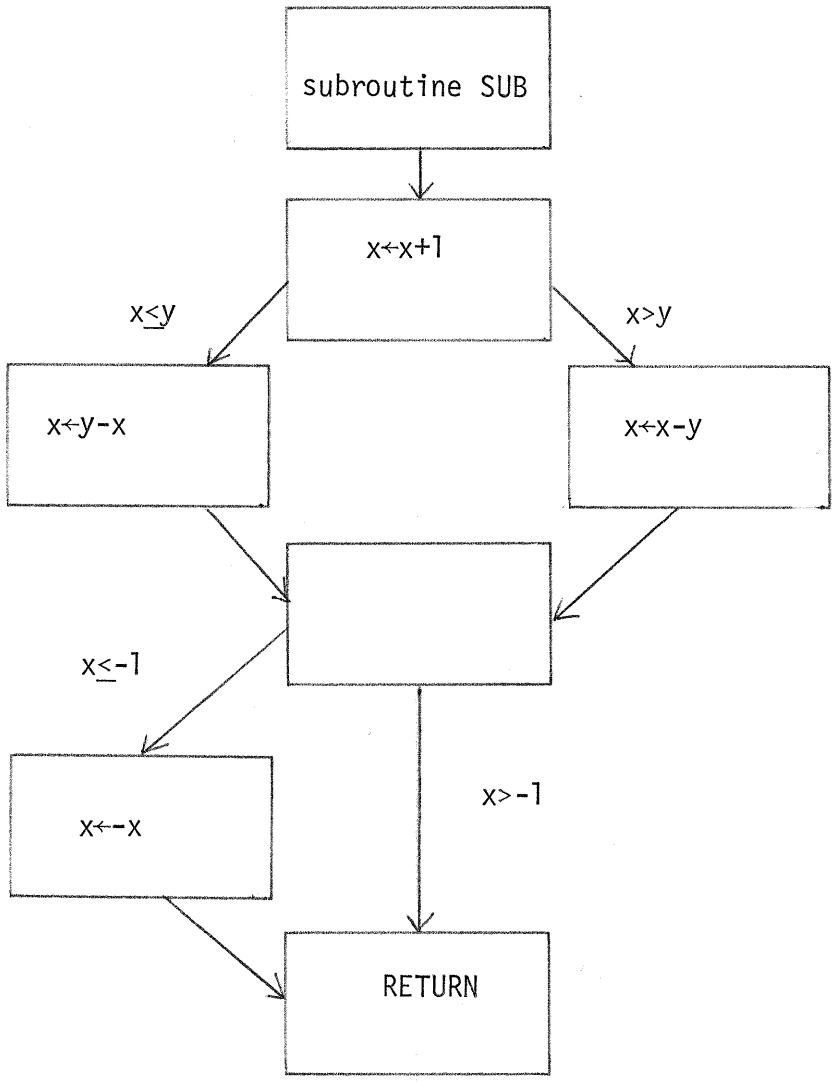


Figure 3

set of paths is given by the following BNF grammar.

<PATHSET>::=<PATH>,END

<PATH>::=<NAME>,<BBN>, EOP|

<NAME>,<BBN>, EOP,<PATH>

<BBN>::=<N>,<BBN>|<N>|(<BBN>)$<REPEAT>

<NAME>::= Name of subprogram unit

<N>::=basic block number

<REPEAT>::=loop repeat count ($\geq$1)

where <NAME> designates the entry or return to a program unit, EOP designates the end of a path, and END designates the end of the analysis. In addition, each path must satisfy the following conditions:

1) it must be a control path;
2) it can enter or return from a subprogram only when the corresponding code contains a procedure reference or return;
3) whenever a path enters a program unit the initial basic block must be the first executable basic block in the program unit.

For example, consider the path described by SUB1,1,2,5, SUB2,1,7,8, SUB1,(6,7)$2,EOP,END. In analyzing the path the system will start with subprogram SUB1 and symbolically execute basic blocks 1, 2, and 5. While executing basic block 5 there is a procedure call to subprogram SUB2. Subprogram SUB2's basic blocks 1, 7, and 8 are then executed. In basic block 8 there is a return statement and the analysis returns to subprogram SUB1, basic block 5. The remaining code in block 5 is executed. Then the loop formed by basic blocks 6 and 7 is executed twice.

If at any time during the analysis of the path, it can be determined that the path is infeasible, a message is returned to the user and the

analysis of that path is terminated. If the end of the path is encountered, the path is executable and the test data that is generated by the inequality solver is returned to the user.

Another feature that could be easily added to the system is to allow the user to request that the symbolic representation of a variable be printed after the execution of any basic block. This facility would allow the user to investigate the evolution of a variable on a path and would be useful in isolating a program error. The user can presently cause the symbolic representation of a variable to be printed by inserting an output statement in the source code, however, this method is not as flexible since the source code cannot be modified during the data generation analysis.

The interactive mode is more human oriented. The user is aided by the system in selecting a control path. To initiate the analysis of a path in the interactive mode the user first designates the starting subprogram unit. If after a basic block x_j is executed there is more than one exit block (an exit block is any of the set of nodes x_i such that $x_i \varepsilon \Gamma x_j$) then the system lists all the exit blocks. The user chooses one of these exit blocks as the next node in the control path or ends the path. The analysis program informs the user when a path has entered or returned from a program unit.

If in attempting to analyze a block it is determined that the path is infeasible, a message is issued. The user may then end the analysis, end the analysis of that path and start a new path, or choose another exit block from the list and let the analysis continue. When the end of a path is encountered (either in the code or at the user's request) the data to drive execution down the path is printed and the user may initiate a new path if desired.

The system is designed so that it can easily be modified to save its current state. For example, if the user wants to follow more than one of the exit blocks from a basic block he may request that the current state be saved. He then chooses one of the exit blocks and continues with the analysis of the path. At a later time he may return to the saved state to choose another exit block and path.

4.3 Symbolic Execution

Symbolic execution involves assigning expressions to variables instead of values while following a program path. An expression represents the computation that would have evolved in order to compute each variable's value. An expression is represented internally as a tree. The trees are similar to expression trees that are often used in compilers for translating statements.[10] However, since the tree that is constructed here may represent several statements instead of one, it is called an evolution tree. The symbolic representation of a variable can be generated by traversing the variable's evolution tree.

An example of an evolution tree for a small segment of code is given in figure 4c. The method used to build the tree is similar to the code optimization techniques for eliminating common subexpressions described by Cocke and Schwartz[11] and is outlined below.

In order to build the tree, input variables and constants are assigned unique numbers called value numbers. All binary expressions are also assigned value numbers and entered into a computation table. The computation table contains the operator, the value number of the two operands, and the value number of the binary expression. In an assignment statement the variable being defined (on the LHS) is given

the same value number as the expression (on the RHS). A variable's value number is stored in the variable's symbol table entry.

To clarify the example to be given here, an input variable's value number will have a prefix of "I" and a constant's value number a prefix of "C." Now consider the statements in Figure 4a. B, C, and D are input variables and let their input value numbers be I1, I2, I3 respectively. The computation table entries for the two arithmetic expressions are shown in Figure 4b. After the first arithmetic statement is symbolically executed, variable A has the value number 2. When variable A is referenced in the second arithmetic statement, the value number 2 is used as the operand. After the second arithmetic statement is executed, C's original value number of I2 is replaced by its current value number 3.

In the fourth statement C is an output variable. The symbolic representation of C can be printed from the evolution tree which is contained in the computation table. The variable's value number is the pointer to the root of the variable's evolution tree. If the variable's value number is not an input or constant value number, it indexes an entry in the computation table. The operator in the computation table is the evolution tree node. The operands are the pointers to the left and right branches of the tree. Input and constant value numbers are the leaf nodes. The evolution tree and symbolic representation of variable C are depicted in Figure 4c and 4d.

In a similar manner path constraints can also be constructed. Conditional statements are entered into the computation table and assigned a value number. The evolution tree for the constraint can then be extracted from the computation table.

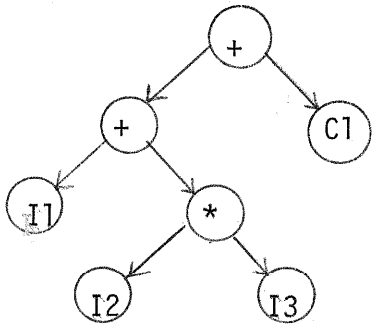
```
READ B, C, D
A = B + C * D
C = A + 5
WRITE C
```

Figure 4a

Operator	Operand 1 value #	Operand 2 value #	Expression value #
*	I 2	I 3	1
+	I 1	1	2
+	2	C 1	3

Computation table for the code in 4a

Figure 4b



Evolution tree for variable C

Figure 4c

C = (I 1) + (I 2) * (I 3) + 1

Symbolic representation of output variable C

Figure 4d

The above method allows communication between subprograms to be handled quite simply. In order to pass information to a subprogram, the actual parameter's value number is passed (even expressions have value numbers.) On returning to the calling program unit, the dummy parameter's value number is passed back. In order to pass an array, a list of value numbers must be passed between the program units. Subroutine linkage causes few problems since the computation table is a global table that reference value numbers and not local variable names.

An input variable receives a new input value every time the variable would receive an external value. For example, if a read statement is in a loop, the input variables in the statement receive new input values every time the loop is executed.

While symbolically executing the path, constant expressions are folded (computed) whenever possible. If the following statements were encountered on a path

```
A = 2
B = 3
C = A-B+1
```

the actual value of C would be computed and A, B, and C would have constant value numbers. Folding simplifies the evolution tree even though some of the history of the variable's evolution is lost. If experience shows that this is a hindrance to validating the program then folding can be suppressed.

4.4 Error Checking

Generating data to force execution down a path can assure that the code has been tested but cannot assure that all errors have been detected.

To increase the chance of detecting some of the more common programming errors, artificial constraints are temporarily created to simulate error conditions. An attempt is then made to solve the augmented set of constraints. If there exists a solution set to the augmented constraints then errors may occur when executing the code and a message is therefore issued.

Subscripts that are out of bounds are a common and often elusive programming error and will be used to illustrate the error detection capabilities. Assume the allowable subscript range of an array is declared to be between 1 and 100. When array element X(I) is referenced on the path, the two constraints $I \leq 100$ and $I \geq 1$ are created. If either of these constraints is consistent with the existing constraints an error message is returned to the user. If both are inconsistent with the existing constraints, they are removed from the set of constraints and the symbolic execution of the path continues. The current implementation also tries to detect division by zero in a similar way.

4.5 Inequality Solver

When a conditional branch is encountered during the symbolic execution of a path, a constraint is generated. If the constraint can be evaluated to a true or false value then the path analysis continues or an infeasible path message is returned to the user, respectively. More often, however, the constraint depends on an input variable and therefore, a true or false value cannot be assigned to the constraint. The constraint is then passed to an inequality solver. The inequality solver attempts to find a solution to the set of constraints and therefore confirm that the set of constraints is consistent. If a solution exists, the sym-

bolic execution of the path continues. If the constraints are inconsistent or cannot be handled by the inequality solver, a message is returned to the user.

If the set of constraints is linear then linear programming techniques can be employed to solve the system. The general form of a linear programming problem is

$$\text{Max } O(x)$$
$$\text{Subject to } Ax \leq b$$
$$x \geq 0$$

where O is a linear function called the objective function, x is an m vector, b is an n vector, and A is an nxm matrix with m>n. The vector x is the set of input variables, and the constraints generated by the system are transformed into the form $Ax \leq b$. A few examples of some of the possible transformation techniques are outlined below. These are intended to demonstrate the applicability of linear programming to solving the path constraints.

The constraint $a_{ij}x_i \geq b_i$ can be easily transformed into the $\leq$ form by multiplying the constraint by -1. The constraint $a_{ij}x_i = b_j$ can be replaced by the two constraints $a_{ij}x_j \leq b_j$ and $-a_{ij}x_j \leq -b_j$. The constraint $a_{ij}x_i < b_j$ can be transformed using a technique called the big M method.[12] Using this technique the constraint becomes $a_{ij}x_i + y \leq b_j$ where y>0. To cause y to be greater than zero, My is added to the objective function where M is a large value. The modified problem is then

$$\text{Max } O'(x) = O(x) + My$$
$$\text{Subject to}$$
$$a_{ij}x_j + y \leq b_i$$

A feasible solution exists only when y>0.

The constraint $a_{ij}x_j \neq b_i$ can be replaced by either of the two inequalities $a_{ij}x_i < b_i$ or $a_{ij}x_j > b_i$ which can then be transformed by the methods outlined above. If it is then determined at some point in the analysis that the set of inequalities is inconsistent, the alternative constraint will then be attempted. If there are N distinct constraints on the given path then there are 2^N possible sets of constraints. Solutions to all 2^N sets of constraints would have to be unsuccessfully attempted before it could be determined that the path is infeasible.

To avoid having to use the trial and error approach described above, another technique may be employed, assuming there is an upper and lower bound on the constraint. Assume $a_{ij}x_j \neq b_i$ is the constraint and $L \leq a_{ij}x_j \leq U$. Then the follow constraints are added to the system of inequalities.

$$(b-L)y + a_{ij}x_j > b$$
$$(b-U)(1-y) + a_{ij}x_j < b$$
$$y \leq 1$$

where U and -L are chosen to be large numbers and $y(0 \leq y \leq 1)$ is an integer variable. This same technique can be employed to handle disjunctive constraints.

The input variables will not necessarily be restricted to non-negative values as required by the linear programming problem. If variable x_j is not constrained to be non-negative then this can be handled by substituting $(x_p - x_q)$ for x_j where $x_p, x_q \geq 0$. The linear programming system solves for x_p and x_q which will then be used to compute x_j.

Each time a new constraint is generated during the symbolic execution of a path, the inequality solver is called. The inequality

solver first checks to see if the new constraint is consistent with the last solution. If it is, the inequality solver is done. If it is not consistent, the constraint is added to the linear programming problem. A nice property of some linear programming algorithms is that when a constraint is added to a previously solved problem, the previous solution can be used to find a solution to the augmented problem. This is more efficient then solving the augmented problem as a totally new linear programming problem.[12] By exploiting this property, the speed of the data generation system will be significantly increased.

The vector x of input variables may be of various data types; real, integer, logical, hollerith, or complex. There exist linear programming systems that handle mixed integer and real data types so this causes no problems.[12] Logical data types can be handled by converting true and false to the values 1 and 0 respectively. When a logical input variable, say L, is first encountered on the path, it is designated as an integer variable and the constraints $0 \leq L$ and $L \leq 1$ are generated. Hollerith variables are treated as integer variables. Complex variables, however, cannot be handled by the linear programming inequality solver.

Linear programming techniques are restricted to systems of linear constraints. Therefore, not all paths can be completely analyzed. It is felt that the research will show that a large proportion of programs have linear constraints and will not be limited by this restriction. Cursory observations of programs as well as Knuth's analysis of FORTRAN programs[13] reinforce this view. When the constraints are nonlinear and the inequality solver fails, the constraints themselves should still be an aid to humans in selecting test data and in validating the program.

5. Problems and Future Research

The test data generation system that has been described above is meant to be an experimental system. There are various areas that need to be investigated further. In addition, there are some extensions that would enhance the system's efficiency and effectiveness that should be explored. A brief description of some of these areas will be given below.

5.1 Intermediate Code Translation

The current system could be made more efficient if optimization and parallelization techniques were applied to the intermediate code. Optimization such as common subexpression elimination, folding, and removing code from loops as well as parallelization of array expressions would reduce execution time and storage requirements. Since there exists a wealth of information on code optimization, it will not be discussed here. A brief description of parallelization that would benefit the test data generation system will now described, however.

Consider the code below.

```
      DO 10 I = 1,1000
   10 A(I) = 0
```

The only feasible path requires that the loop be symbolically executed 1000 times, creating an array element entry for each array element defined in the loop. If parallelization were employed a series of array elements could be denoted by an interval such as $[n_1,n_2,n_3,n_4]$ where n_1 is the start of the array subscript, n_2 is greater or equal to the final array subscript, n_3 is the increment value used to compute the next subscript

in the series, and n_4 is the nested loop level. This notation is similar to the DO notation in FORTRAN. Using this notation, the loop above could be written as A[1,1000,1,1]=0. Only one array entry would be needed to store this information in the computation stack and the path would be traversed only once during the symbolic execution. Procedures for implementing the parallelization will not be given here.

It is interesting to note that the information required in program validation, code optimization and parallelization are all quite similar. Perhaps future compilers will build a standard data base of information about the program which can be used for validation, optimization, and parallelization.

5.2 Path Selection

If parallelization techniques are not employed, a problem arises in analyzing loops when the designated number of loop traversals does not agree with the number of traversals dictated by the code. This problem arises when a loop is closed (i.e. has only 1 entry and 1 exit point) and the number of traversals does not depend on input variables. This appears to be a common occurrence, especially during the initialization of arrays. For example, again consider the code

```
      DO 10 I = 1,1000
10    A(I)=0
```

If the user requests that the loop be traversed any number of times other than 1000, an infeasible path is detected. Loops similar to the above are probably not a major concern to the user since they have no affect on the input variables. However, it is a burden on the user that the exact number of loop traversals be known since it may not always be as

obvious as in the above example.

The system can be extended to return the required number of loop traversals for closed loops not dependent on input variables. It may also be desirable to override the user's request in these circumstances and (after issuing the appropriate message) execute the loop the required number of times.

There are various other circumstances where information on the structure of the program's executable paths can be recognized. Problems of this type need to be explored further.

5.3 Symbolic Execution

A major problem in analyzing programs is the inability to recognize distinct array elements. If an array's subscript depends on an input variable, it is impossible to tell which element is being referenced. To illustrate this problem, consider the following statements with input variable J.

```
DIMENSION A(10)
  .
  .
  .
A(5) = 1
A(I) = A(J)+2
```

A(J) may or may not be the same array element as A(5). There are several alternative methods of handling this problem; however, none are completely satisfactory.

One such approach is to mark any variable that is ambiguously defined as undetermined. If both variables I and J in the above code depend on input variables then array A is marked as undetermined. If only variable J depends on an input variable, then only array element

A(I) is marked as undetermined. Any variable that subsequently depends on an undetermined variable is also marked as undetermined. This method hinders the generation of test data only if a constraint depends on an undetermined variable.

A second alternative is the approach used by the SELECT system to analyze LISP type programs.[6] A value for the ambiguous subscript is chosen by the system. If an infeasible path is subsequently detected, the analysis program then backtracks and tries another subscript value. The path is marked as infeasible only after all possible subscripts have been attempted and failed. The problem with this method is that it may prove too costly, especially for FORTRAN programs with large arrays. For the above example with both I and J as input variables, there would be 100 possible subscript combinations for that one statement alone.

A third alternative attempts to classify nonfolding subscripts in terms of an input variable. Array subscripts are allowed that reduce to the form I+C where I is an input variable and C is a constant. In this method, if an array is subscripted in terms of an input variable it must always be subscripted in terms of that input variable or the array element is ambiguous. Ambiguous array references will be marked as undetermined as outlined in the first method described above.

This method seems to be a reasonable approach since cursory observations of programs have shown that nonfolding subscripts often reduce to the form I+C. The major drawback of this method is its inability to handle arrays passed as parameters between program units. Fortran allows segments of arrays to be passed and array dimensions to be redefined between one program unit and another. Neither of these program

constructs can readily be accommodated using nonconstant subscripts.

The current implementation employs the first method outlined above for handling arrays. Using this method, information can be obtained on the frequency of ambiguous array subscripts and analysis failure due to this problem. The second method of automatically selecting subscripts and backtracking when necessary can easily be implemented at a later time if desired. To implement the third approach would require extensive modifications to the current implementation.

Another difficulty that arises in FORTRAN is in analyzing the REWIND and BACKSPACE statements correctly. To analyze these statements a history of the input and output variables for each file must be maintained. The number of items read or written on a file may depend on an input variable and therefore may not be determinable. The current implementation cannot handle REWIND or BACKSPACE statements and ignores these statements. This is not considered a serious problem, however, since these statements occur relatively infrequently.

5.4 Inequality Solver

The path constraints and symbolic representation of the output variables may be quite unwieldy. As was noted, constant expressions are folded during the symbolic execution of the path, but this is only done within a binary expression. The inequality solver, however, requires that the constraints be in a simplified form. The simplification can be easily accomplished by passing the expression to a system that is designed to do symbolic algebraic manipulations such as ALTRAN.[14] Therefore, both the path constraints and output variable expressions are simplified by ALTRAN immediately after they are generated.

It may also be desirable to allow the user to suppress the simplification of expressions. The unsimplified expression is in a form that represents the variable's evolution while the simplified expression, though concise, loses some of the historical information. For example, the expression A = (2*I1) + (I1+2) contains more information than the simplified expression A = 3*I1+2. The unsimplified form may be desired, especially when the user suspects a program error.

A major drawback of the inequality solver described in section 4.5 is that only linear constraints can be accepted by linear programming algorithms. Though this may not affect many programs, the system would be enhanced with a more general inequality solver. There exist various techniques for handling special nonlinear cases.[12,15] These techniques must be examined further. The modularity of the test data generation system allows various inequality solvers to be tested.

6. Conclusion

The test data generation system described in this paper symbolically executes a path and provides a symbolic representation of the output variables and path constraints in terms of the program's input variables. Common run time errors such as subscripts that are out of bounds and division by zero may also be detected. Using the path constraints an attempt is made to generate test data that would cause execution of the selected path or to determine that the path is infeasible. The capabilities provided by this system should aid in program testing and validation.

The system is limited in its ability to handle all constructs of FORTRAN, particularly array references that depend on input variables. In addition, test data generation is currently confined to paths that

can be described by a set of linear path constraints. Even in program paths where the analysis is incomplete due to the system's limitations, knowledge will be gained about the program from the path constraints and symbolic representation of the output variables.

This system is currently being implemented. It is hoped that from this experimental system more can be learned about the structure of programs such as the types of array usage and the complexity of path constraints. This information should aid future research in program validation and test data generation.

References

1. Hetzel, William C. (editor) Program Test Methods. Prentice-Hall (1973).

2. Brown, J. R.; De Salvia, A. J.; Heine, D. E.; Purdy, J. G., "Automated Software Quality Assurance," Program Test Methods, pp. 181-203.

3. Howden, William; Stucki, Leon G., "Methodology for the Effective Test Case Selection," McDonnell Douglas MDC G5307, January 1974.

4. Osterweil, Leon J and Fosdick, L. D., "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection." Report 55 (September 1974) Dept. of Computer Science, University of Colorado, Boulder, CO.

5. Huang, J. C. "Program Testing." (May, 1974) Dept. of Computer Science, University of Houston, Houston, Texas.

6. Boyer, Robert S.; Elspas, Bernard; Levtt, Karl N. "Select--A System for Testing and Debugging Programs by Symbolic Execution." (Preliminary copy) Stanford Research Institute.

7. King, James C. "A New Approach to Program Testing." RC 5037 (September 1974) Computer Science Department, IBM Thomas J. Research Center, Yorktown Heights, N. Y.

8. King, James C. "Symbolic Execution and Program Testing." RC 5082 (October, 1974) Computer Science Department, IBM Thomas J. Research Center, Yorktown Heights, N. Y.

9. Osterweil, Leon; Clarke, Lori; Smith, David W. "A Fortran System for Flexible Creation and Accessing of Data Bases." Report 52 (August 1974) Dept. of Computer Science, University of Colorado, Boulder, CO.

10. Gries, David, Compiler Construction for Digital Computers, Wiley (1971).

11. Cocke, J. and Schwartz, J. T., Programming Languages and Their Compilers. New York University, Courant Institute of Mathematical Sciences.

12. Dantzig, G. B. Linear Programming and Extensions, Princeton University Press (1963).

13. Knuth, D. C., "An Empirical Study of FORTRAN Programs," Software--Practice and Experience, Vol. 1, 1971, pp. 105-133.

14. Brown, W. S. Altran User's Manual, Bell Telephone Laboratories, Inc. Vol. 1, 1973.

15. Elspas, Bernard; Green, Milton; Korsak, Andrew; and Wong, Peter. "Solving Nonlinear Inequalities Associated with Computer Program Paths", (preliminary draft). Stanford Research Institute.