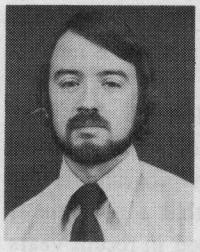
Proc. IEEE Symp. Computer Software Reliability, New York, NY, 1973.
[20] T. A. Thayer *et al.*, "Software reliability study," TRW Rep. 74-2260.1.9-29, June 1974.

William E. Howden was born in Vancouver, Canada, on December 8, 1940. He received the B.A. degree in mathematics from the University of California, Riverside, in 1963, the M.Sc. degree in mathematics from Rutgers University, New Brunswick, NJ, in 1965, the M.Sc. degree in computer science from Cambridge University, Cambridge, England, in 1970, and the Ph.D. degree in computer science from the University of California, Irvine, in 1973.



In 1965 and 1966 he was with Atomic Energy of Canada, Chalk River, Ont. From 1970 to 1974 he was a Lecturer in computer science at the University of California, Irvine. Since 1973 he has been a consultant to McDonnell Douglas, Huntington Beach, in software reliability. He is currently Assistant Professor of Information and Computer Science at the University of California, San Diego. His research interests are in software and system reliability and in interactive problem solving.

Dr. Howden is a member of the Association for Computing Machinery and the British Computing Society.

A System to Generate Test Data and Symbolically Execute Programs

LORI A. CLARKE

Abstract—**This paper describes a system that attempts to generate test data for programs written in ANSI Fortran. Given a path, the system symbolically executes the path 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 obtain a solution. A solution to the set of constraints is 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 error conditions and then an attempt is made to solve each 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 assertion generation and automatic program documentation.**

Index Terms—**Program validation, software reliability, symbolic execution, test data generation.**

I. Introduction

THERE is a growing awareness of the problems involved in testing programs and of the need for automated systems to aid in this process. This paper describes an implemented system that aids in the selection of test data and the detection of program errors.

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 testing programs is inadequate and costly [1]. Consequently, several alternative approaches have been proposed. These approaches can be categorized into two areas, program corrections (also called program verification or program proving) and program validation.

In the program correctness method formal mathematical proofs are used to demonstrate that a program terminates and satisfies the program's specifications. First, assertions about the program's variables are made at various points in the code and then theorem proving techniques are employed to verify the correctness of these assertions. In general, automated theorem proving techniques are used, though human assistance is still needed [2].

Program correctness has focused attention on the problems of program reliability. However, the state of the art is such that there are many drawbacks that prevent program correctness from being a practical tool, at least in the immediate future. Major difficulties are the creation of program assertions and the considerable human interaction frequently required in the theorem proving stages. Even after this rather complex process the results may be questionable. If the program cannot be proved correct this may be due to an error in the program but also may be due to a flaw in the assertions or limitation in the theorem prover, human or machine. Even if the program is proved correct, this process still may be questionable. In addition, proving programs cor-

Manuscript received October 6, 1975; revised May 14, 1976. This work was supported in part by the National Science Foundation under Grant GJ 36461.

The author is with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

rect is a complex and tedious task and, therefore, is not usually applied to large programs where such analysis is needed most [2], [3].

Program validation is a more vaguely defined area that encompasses a wide range of automated tools that analyze and evaluate programs. These tools aid in program testing though they do not necessarily guarantee that a program is correct. Various validation projects developed in recent years have analyzed several aspects of programs. For example, the DAVE system developed by Osterweil and Fosdick [4] analyzes the data flow of programs and detects data flow anomolies within and between subprograms. The PET system developed by Stucki [5] maintains relevent execution information about statements such as the execution count and minimum and maximum values. Ramamoorthy, Meeker, and Turner developed a system called ACES that detects unreliable program constructs [6]. The EFFIGY system developed by King algebraically represents a path's computations by symbolically executing a path [7]. The SELECT system developed at the Stanford Research Institute attempts to generate test data and verify assertions for program paths [3]. These are but a few of the validation tools that are now available.

The system that will be described here also aids 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 powerful tool that can be employed to improve the current haphazard approach to testing programs. 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 never have been tested.

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.

3) Creates symbolic representations of the program's output variables as functions of the program's 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 assertion generation and automatic program documentation.

4) Detects certain types of program errors. Executing a program path and verifying the results does not guarantee that the path is correct for all possible input data. Therefore, 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. This incorporates some of the ideas for error checking suggested by Sites [8].

II. System Properties

Generating test data is a difficult task posing many complex problems. In fact, the general problem is unsolvable. Therefore, the system philosophy is to design and implement a system that will test the feasibility of generating test data as well as to clarify the major problem areas. The system is limited in that it requires each analysis path to be completely specified, it generates test data only when the path constraints are linear, and it ignores input and output specifications. These limitations are explained below.

The problem of generating test data to execute any specified statement in a program is analogous to the halting problem (assume the statement is STOP) and thus unsolvable. To circumvent this problem it is necessary to eliminate infinite paths. An approach that has been employed is to analyze paths that are restricted to a maximum-loop count or number of statements [3], [7]. This technique often creates a proliferation of program paths, especially in large programs. Furthermore, the user may only be concerned with a few paths or paths that satisfy a particular criterion. Therefore, it was decided that the system will not incorporate path selection. The system requires that the paths be completely specified but leaves the criteria of path selection to the user.

To generate test data that satisfies the conditional statements on the path requires that the system be capable of solving arbitrary systems of inequalities, an unsolvable problem [9]. Therefore, to generate test data for a specific program path is an unsolvable problem. A hypothesis of this work is that the inequalities will usually be relatively simple and often linear. This view is reinforced by cursory observations of programs and by Knuth's study of Fortran programs [10]. Should this assumption prove valid then linear programming techniques can be employed in most cases to solve the system of inequalities. If this premise fails then more powerful methods such as the conjugate gradient method [11] can be applied to solve the inequalities. The conjugate gradient method requires human interaction while the linear programming algorithm does not. For the prototype it was decided that human interaction will be kept to a minimum and introduced later only if experience indicates it to be necessary.

Another difficulty is array subscripts. A problem occurs when array subscripts depend on input data. For example, in the following segment of code assume the value of J is read as data.

```
A(1)=10.
A(2)=0.
IF(A(J). LT.5.)...
.
.
.
```

The value of J affects the conditional statement and therefore must be determined. This is a computable problem. J can only be assigned a value from a finite set of integers and, therefore, all possible values of J could be enumerated. Though enumeration is a possibility it may prove to be impractical. Therefore, the approach that was chosen marks whenever such a constraint occurs so that information may be obtained on how frequently the problem occurs. The test data generation process will only be stymied if undetermined array subscripts affect a conditional statement.

A decision was also made to ignore input and output statements except for the read and write variable lists. Thus, format specifications and other relevant information which constrain the size of the test data is ignored. Information of this type, though quite useful, adds another degree of complexity to the problem. It is felt that the main issues of test data generation should be addressed first and then, if the problem appears accessible, the system can be enhanced further.

Therefore, the general philosophy is to implement a prototype that can generate test data for some program paths. Human interaction will be kept to a minimum. The cause of all failures as well as relevant program constructs will be monitored so that the areas of greatest concern can be determined. It is expected that the actual test results will be informative about the feasibility of test data generation and will provide a good basis for future directions of research.

III. Overview

To describe the system, a few definitions are first needed. The subject program is represented by a directed graph call the *control flow graph*. The nodes in the graph are the executable statements of the program and the edges represent the program flow. The control flow graph is assumed to have one entry point n_0 (a node with indegree zero) and one or more exit points (nodes with outdegree zero). Γ is used to denote the successor operator. Thus, Γn denotes the set of nodes joined to the node n by edges directed from n. A *control path* is a sequence of nodes $n_{i1}, n_{i2}, \ldots, n_{im}$ where $n_{ij+1} \in \Gamma n_{ij}$ and $n_{i1} = n_0$. Not every control path can be executed. An *execution path* is a control path which cannot be executed.

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. A *program input variable* is a variable that receives a value by means of some form of external communication. Conversely, a *program output variable* returns a value by some form of external communications. Note that external communication can occur in input and output statements and, if the calling program is not being analyzed, in the parameter and COMMON variables.

To generate the constraints the path is symbolically executed. When a path is symbolically executed values are not assigned to variables as occurs during normal execution, rather, expressions denoting the evolution of the variables are assigned. For example, in Fig. 1, the variable representation of J after statements 1,2,3 and 4 are symbolically executed is J = I2 - I1 - 1 where I1 and I2 denote the input values of parameters J and K, respectively. The constraint created by the above control path is I1 + 1 $\leqslant$ I2.

Whenever a conditional transfer of control is encountered one or more constraints, representing the branch from the chosen conditional statement, are generated. Ideally, each constraint would be passed to an inequality solver to check its consistency with the existing constraints. If the constraint is inconsistent, it would be known that the path is infeasible. If the constraint is consistent, the symbolic execution of the path would continue. At the end of the path, the solution set found by the inequality solver would be a data set that would force execution of the designated path. Because of storage limitations and interface problems, however, the system does the symbolic execution as a separate job step. The constraints are written to a file and then during the next phases, simplified and solved. Although this is inefficient, since the symbolic execution may continue down an infeasible path, it is a reasonable alternative for an experimental system.

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

Fig. 1.

To demonstrate these ideas, consider the example in Fig. 1. The path through statements 1-5, 7, and 9 is associated with the following set of constraints:

$$I1 + 1 \leqslant I2$$

$$I2 - (I1 + 1) > -1.$$

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 1-3 and 6-9 is associated with the following set of constraints:

$$I1 + 1 > I2$$

$$I1 + 1 - I2 \leqslant -1.$$

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

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.

Symbolically executing programs in order to represent the path's computations was proposed by Balzer in the EXDAMS system, "extensible debugging and monitoring system," in 1969 [12]. Current research has also used symbolic execution to generate test data and validate programs.

The SELECT system [3] generates test data and creates a symbolic representation of the output variables for programs written in a subset of Lisp. SELECT has limited capabilities to handle procedure calls. Also, the path constraints and output variable representations are currently represented as a Lisp list rather than a more human recognizable form. EFFIGY accepts programs written in a subset of PL/1 that allows only integer values.

Howden has developed a system that recognizes classes of paths and generates path descriptions [13]. Miller and Melton have also developed a system that generates the path constraints [14]. These systems analyze Fortran programs. In both systems the path descriptions must be manually solved to generate test data and they do not handle procedure calls adequately. Huang has also briefly described a system for generating the path constraints [15]. Miller and Melton, Howden, and Huang do not use symbolic execution but use an approach that traces backwards through the code to determine how the conditional statements evolved. This approach does not allow early detection of nonexecutable paths and requires multiple passes to analyze each iteration of a loop.

An iteractive system developed at TRW recognizes some nonexecutable paths and aids the user in generating test data [16]. Goodenough and Gerhart have proposed a method of selecting test data and paths using decision tables [17]. Both these methods require more user participation 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 a large source of programs in need of validation is readily available. 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.

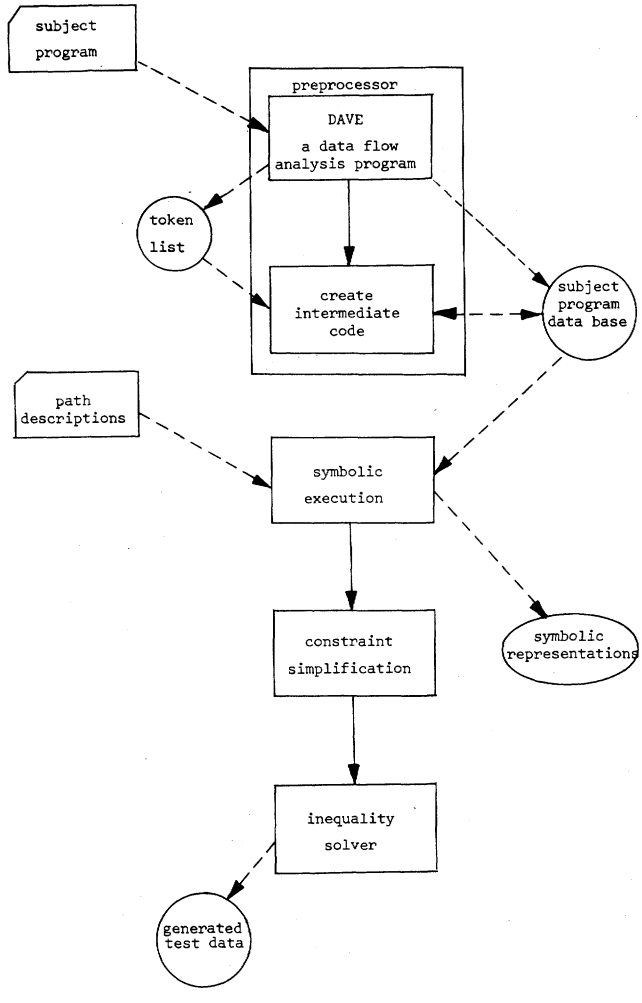


Fig. 2.

IV. Structure of the Analysis Program

Fig. 2 depicts the overall flow of the analysis program. The system consists of a preprocessor and three phases; symbolic execution, constraint simplification, and inequality solver. Instead of building the preprocessor from scratch the DAVE system, a data flow analysis program, is used. 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 table, COMMON table, and label table similar to tables usually constructed by compilers, and a statement flow table that represents the control flow graph. The token list and tables described above are also used in the data generation system.

The data generation system can run independently of DAVE and in fact does not use the more sophisticated capabilities of that system. A different preprocessor could be implemented that would simply build the needed tables.

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 each statement is stored in a doubly linked list that is attached to the corresponding node of the control flow graph. 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 Fig. 1 is shown in Fig. 3.

Representing the subject program in an intermediate form has several advantages. First, as was noted, it allows the analysis to be more easily adapted to other languages. A new language would have to be translated into the intermediate code and then, depending on the new language, modifications would be necessary to the test data generation system. Second, since all expressions are represented as a series of binary and array 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 detection of parallelism in the code.

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

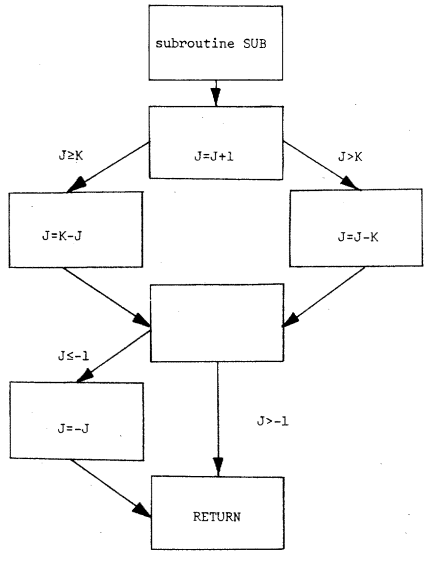


Fig. 3.

method is designed to aid a human user in selecting a path. Both methods will be described below.

In the static mode, a path is designated by a sequence of subprogram names, statement numbers, and loop counts. EOP designates the end of a path, END designates the end of the analysis, and $N designates a loop count of n. 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 statement must be the first executable statement in the program unit.

For example, consider the path described by

SUB1, 1,2,3,5, SUB2, 1,2,7,8, SUB1,(6,7)$2,EOP,END.

In analyzing this path the system will start with subprogram SUB1 and symbolically execute statements 1,2,3 and 5. While executing statement 5 there is a procedure call to subprogram SUB2. Subprogram SUB2's statements 1,2,7, and 8 are then executed. In statement 8 there is a return statement and the analysis returns to subprogram SUB1, statement 5. The remaining code in statement 5 is executed. Then the loop formed by statements 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. Since the inequality solver is called only after the symbolic execution phase is complete, infeasible paths are only detected at this stage when a predicate folds to the value false.

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 statement n_j is executed there is more than one exit node (an exit node is any of the set of nodes n_i such that $n_i \in \Gamma n_j$) then the system lists all the exit nodes. The user chooses one of these exit nodes 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 path 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 node from the list and let the analysis continue.

Symbolic Execution

Symbolic execution involves assigning expressions instead of values to variables 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 directed graph. The graphs are similar to expression trees that are often used in compliers for translating statements. However, the graph that is constructed here is called an evolution graph and may represent several statements and variables instead of just one statement and variable. The symbolic representation of a variable is generated by traversing the variable's evolution graph.

An example of an evolution graph for a small segment of code is given in Fig. 4(c). The method used to build the graph is similar to the code optimization techniques for eliminating common subexpressions described by Cocke and Schwartz [18] and is outlined below.

In order to build the graph, input variables and constants are assigned unique symbols called input and constant value numbers. All unary and binary expressions are assigned computation value numbers and entered into a computation table. The computation table contains the operator, the value numbers of the two operands, and the computation value number of the binary expression. In an assignment statement the variable being defined on the left-hand side (LHS) is given the same value number as the expression on the right-hand side (RHS). A variable's value number is stored in the variable's symbol table entry in the subprogram's data base.

To clarify the example to be given here, an input variable's

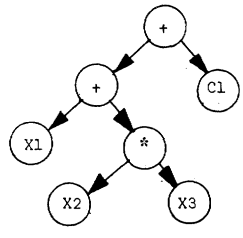
```
READ(UNIT) B, C, D
A = B + C * D
C = A + 5
WRITE C
```

(a)

Operator	Operand 1 value #	Operand 2 value #	Expression value #
*	X 2	X 3	T 1
+	X 1	T 1	T 2
+	T 2	C 1	T 3

Computation table for the code in 3.8a

(b)



Evolution graph for variable C

(c)

name	value number
A	T 2
B	X 1
C	T 3
D	X 3

Symbol table after execution of the code in 3.8a

(d)

```
C = (X1) + (X2) * (X3) + 5
```

Symbolic representation of output variable C

(e)

Fig. 4.

value number will have a prefix of "X," a constant's value number a prefix of "C," and a computation value number a prefix of "T." Now consider the statements in Fig. 4(a). B, C, and D are input variables; let their input value numbers be X1, X2, and X3, respectively. The computation table entries for the two arithmetic expressions are shown in Fig. 4(b). After the first arithmetic statement is symbolically executed, variable A has the value number T2. When variable A is referenced in the second arithmetic statement, the value number T2 is used as the operand. After the second arithmetic statement is executed, C's original value number of X2 is replaced by its current value number T3. Fig. 4(c) shows the symbol table entires for A, B, C, and D at this point.

In the fourth statement C is an output variable. The symbolic representation of C can be printed from the evolution graph which is contained in the computation table. The variable's value number is the pointer to the root of the variable's evolution graph. If the variable's value number is not an input or constant value number, it indexes an entry in the computation table. The operands are pointers to the left and right edges of the graph. Input and constant value numbers are the terminal nodes. The symbolic representation of a variable is found by a preorder traversal of the subgraph pointed to by the variable's value number. The evolution graph and symbolic representation of variable C are depicted in Fig. 4(c) and (d).

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

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.

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 graph even though some of the variable's evolution is lost. If experience shows that this is a hindrance to validating the program then folding can be suppressed.

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 $S(I) > 100$ and $S(I) < 1$ are created where S(I) represents the symbolic representation of variable I. 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.

Simplification Phase

The constraints that have been generated during the symbolic execution phase may be long and in an unsimplified form. Therefore, the constraints are first simplified before an attempt is made to solve the inequalities. For example, the constraint I1 - I2 $\leq$ 3 * I2 would be simplified to I1 - 4 * I2 $\leq$ 0.

Many techniques could have been used to simplify the constraints. Because of ease of applicability and availability however, Altran, a language designed for algebraic manipulations, was chosen [19].

The inequality solver that will be described in the next section requires that the constraints be linear. Therefore, the Altran program also recognizes and flags nonlinear constraints and constraints that reference the intrinsic and built-in Fortran functions. The Altran program must often manipulate the expression to obtain a linear form. For example, I1/I2 $\leq$ 7 is recognized as equivalent to a linear expression and is therefore transformed into I1 - 7 * I2 $\leq$ 0.

The inequality solver also requires that the inequalities be in a specific form that will be described in the next section. The Altran program easily performs the necessary manipulations.

Inequality Solver

The inequality solver attempts to solve the constraints that have been generated during the symbolic execution phase and simplified during the simplification phase. As was mentioned, ideally the inequality solver would be called after each constraint is generated. If the new constraint is consistent with the previous constraints then the symbolic execution would continue. If the new constraint is inconsistent then the path is infeasible and the symbolic execution of that path would end. However, due to limitiations of the current operating system, it was decided that for this experimental system the symbolic execution would be done as a separate job step.

In order to determine when a path becomes infeasible, each constraint is added to the system of constraints one at a time. When a constraint is first added to the system of constraints a check is made to determine if the previous solution satisfies the current constraint. If so, the new constraint is consistent and the next constraint may be added. If not, a new solution is attempted. If the inequality solver finds that the new constraint is inconsistent with the previous constraints, then the path is infeasible. A message is returned to the user and then the next path is attempted. If all the constraints are consistent, then the final solution is a test data set that would cause execution of the path.

When a temporary constraint is encountered, it is added to the set of previous constraints and a solution is attempted. If a solution can be found, then an error message is returned to the user. In any case, the temporary constraint is removed from the set of constraints and the next constraint is examined.

As was previously shown, solving a general system of inequalities is an unsolvable problem. Therefore, any choice of an algorithm will not always be successful. The algorithm that was chosen is a linear programming algorithm for integer and real variables due to Glover [20]. Of course, linear programming algorithms can solve only systems of linear constraints. The general linear programming problem and its applicability to test data generation is described below.

The general form of a linear programming problem is

$$\text{MAX } 0(X).$$

Subject to

$$AX \leq B$$

$$X \geq 0.$$

Where 0 is a linear function called the objective function, X is an N-vector of unknowns, B is an M-vector of constants, and A is an $M \times N$ matrix with $(N > M)$ [21]. The vector X represents the input variables. The constraints must be transformed into the form $AX \leq B$. A few examples of the transformation techniques will be described below.

The constraint $\Sigma_j a_{ij} x_j \geq b_i$ can easily be transformed into the $\leq$ form by multiplying the constraint by -1. The constraint $\Sigma_j a_{ij} x_j = b_i$ can be replaced by the two constraints $\Sigma_j a_{ij} x_j \leq b_i$ and $\Sigma_j - a_{ij} x_j \leq -b_i$. The constraint $\Sigma_j a_{ij} x_j < b_i$ can be changed to $\Sigma_j a_{ij} x_j + \xi \leq b_i$ where ξ is a small constant. The constraint $\Sigma_j a_{ij} x_j \neq b_i$ can be replaced by one of the two inequalities $\Sigma_j a_{ij} x_j > b_i$ or $\Sigma_j a_{ij} x_j < b_i$ which is then transformed by the methods described above. If the inequality chosen is determined by the inequality solver to be inconsistent with the system of constraints then the alternative constraint will be attempted. Similarly, any constraint containing the OR operator is treated as alternative constraints. AND operators between constraints are removed and each conjunctive term is a separate constraint.

The symbolic representation of the input variables will not necessarily be restricted to nonnegative values as required by the linear programming problem. If variable X_{ij} is not constrained to be nonnegative, then this can be handled by substituting $(X_{ip} - X_{iq})$ for X_{ij} where $X_{ip}, X_{iq} \geq 0$. The linear programming system solves for X_{ip} and X_{iq} and then substitutes back to compute X_{ij}.

The vector X of input variables may be of various data types: real, integer, logical, hollerith, double precision, or complex. As noted, the real and integer data types can be handled by mixed linear programming algorithms. Logical data types are handled by converting true and false values to 1 and 0, respectively. For example, the expression (L.OR. .NOT.M) is represented by (L.EQ.1).OR.(M.EQ.0). Fortran restricts the use of logical operators so that no additional constraints are necessary. Hollerith variables are treated as integer variables. Complex and double precision variables cannot be handled by the linear programming inequality solver.

V. 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 is gained about the program from the path constraints and symbolic representations of the output variables.

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] W. C. Hetzel, Ed, *Program Test Method*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[2] R. London, "A view of program verification," in *Proc. Int. Conf. Reliable Software*, Apr. 1975, pp. 534–545.

[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*, Apr. 1975, pp. 234–244.

[4] L. J. Osterweil, and L. D. Fosdick, "Data flow analysis as an aid in documentation, assertion generation, validation, and error detection," Dep. Comput. Sci., Univ. Colorado, Boulder, Rep. 55, Sept. 1974.

[5] L. G. Stucki, "Automatic generation of self-metric software," in *Rec. 1973 IEEE Symp. Software Reliability*, pp. 94–100.

[6] C. V. Ramamoorthy, R. E. Meeker, and J. Turner, "Design and construction of an automated software evaluation system," in *Rec. 1973 IEEE Symp. Software Reliability*, pp. 28–37.

[7] J. C. King, "A new approach to program testing," in *Proc. Int. Conf. Reliable Software*, Apr. 1975, pp. 228–233.

[8] S. L. Sites, "Proving that computer programs terminate clearly," Dept. Comput. Sci., Stanford Univ., Stanford, CA.

[9] M. Davis, "Hilbert's tenth problem is unsolvable," *Amer. Math. Mon.*, vol. 80, pp. 233–269, Mar. 1973.

[10] D. C. Knuth, "An empirical study of FORTRAN programs," *Software–Practice and Experience*, vol. 1, pp. 105–133, 1971.

[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. M. Balzer, "EXDAMS–Extendable debugging and monitoring system," in *1969 Spring Joint Computer Conf., AFIPS Conf. Proc.* vol. 34. Montvale, NJ: AFIPS Press, 1969, pp. 567–580.

[13] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554–559, May 1975.

[14] E. F. Miller, and R. A. Melton, "Automated generation of test case datasets," in *Proc. Int. Conf. Reliable Software*, Apr. 1975, pp. 51–58.

[15] J. C. Huang, "Program testing," Dep. Comput. Sci., Univ. Houston, Houston, TX, May 1974.

[16] K. W. Krause, R. W. Smith, and M. A. Goodwin, "Optimal software test planning through automated network analysis," in *Rec. 1973 IEEE Symp. Software Reliability*, pp. 18–22.

[17] J. B. Goodenough, and S. L. Gerhart, "Toward a theory of test data selection," *Proc. Int. Conf. Reliable Software*, Apr. 1975, pp. 493–510.

[18] J. Cocke, and J. T. Schwartz, *Programming Languages and Their Compilers*, New York Univ., Courant Inst. Math. Sci.

[19] W. S. Brown, *Altran User's Manual*, Bell Telephone Lab., vol. 1, 1973.

[20] F. Glover, private communications.

[21] G. B. Dantzig, *Linear Programming and Extensions*. Princeton, NJ: Princeton Univ. Press, 1963.

Lori A. Clarke was born in New York City, NY, on February 11, 1947. She received the B.A. degree in mathematics from the University of Rochester, Rochester, NY, in 1969 and the Ph.D. degree in computer science from the University of Colorado, Boulder, in 1976.

She worked for Preventive Psychiatry, University of Rochester, from 1969 to 1970 developing a data base management system. She was an Applications Programmer for the National Center for Atmospheric Research from 1970 to 1974. She is currently an Assistant Professor in the Department of Computer and Information Science, University of Massachusetts, Amherst. Her interests include software reliability and compiler design.

Dr. Clarke is a member of the Association for Computing Machinery.