METHODOLOGY FOR THE GENERATION OF PROGRAM TEST DATA

William E. Howden

University of California at Irvine
2 February, 1974

Tech #41

The research described in this paper was carried out in cooperation with the McDonnell Douglas Astronautics program in the research and development of program validation software. The research was part of a project funded by the National Bureau of Standards.

Abstract

A methodology for generating program test data is described. The methodology is a model of the test data generation process and can be used to characterize the basic problems of test data generation. It is well defined and can be used to build an automatic test data generation system.

The methodology decomposes a program into a finite set of classes of paths in such a way that an intuitively complete set of test cases would cause the execution of one path in each class. The test data generation problem is theoretically unsolvable: there is no algorithm which, given any class of paths, will generate a test case that causes some path in that class to be followed. The methodology attempts to generate test data for as many of the classes of paths as possible. It operates by constructing descriptions of the input data subsets which cause the classes of paths to be followed. It transforms these descriptions into systems of predicates which it attempts to solve.

1. Introduction

The validation phase of the software production process has received increasing attention in the last few years (e.g. [1], [2] and [3]). The two most important approaches to validation which have been studied are program verification and program testing. In the program verification approach a program is mathematically proved to be correct over its entire input domain. In the testing approach a program is shown to be correct over a finite subset of its input domain by evaluating the program over that set. The verification approach is limited to small programs. The testing approach is generally applicable and is likely to remain the most important software validation tool.

Several programming tools have been built which automate parts of the program testing process. Stucki [4] and Brown [5] describe systems which automatically insert instrumentation statements into a program. The instrumentation statements keep a record of the branches and statements that are executed during the testing of a program. Reports can be generated which describe how thoroughly the statements and branches have been tested. The system described by Brown also contains features for manipulating a data base of test cases and for automatically checking test results. Krause [6] describes a system for extracting a sequence of paths from a program which "covers" each branch in the program. The system extracts skeletal descriptions of the paths, which the user examines in order to construct test cases which cause the paths to be followed. The skeletal descriptions which are generated by the system are similar to the "implicit descriptions" which are described below. A scheme has been devised by Paige and Balkovitch [7] for testing a program against its specifications. Ramamoorthy [8] has constructed a system which automatically checks a program for anomolous statements and constructions. Ramamoorthy's system also provides facilities for the automatic insertion of trace statements.

This paper describes a methodology for the generation of program test data. The methodology is a model of the test data generation process and can be used to characterize the basic problems of test data generation. It is well defined and can be used to build an automatic test data generation system. The methodology is general and can be applied to programs in different languages although it was designed with FORTRAN programs in mind. The basic problems of test data generation are described in context, along with the description of the methodology.

The methodology decomposes a program into a finite number of standard classes of program paths. It then attempts to generate a set of test cases which causes one path from each class to be tested. The general test data generation problem is undecidable. There is no algorithm which can examine any class of paths through a program and generate a test case for that class. A complete standard set of test cases contains one test case for each standard class of paths. The methodology attempts to generate a large subset of the complete set.

2. General Approach

The methodology consists of five phases. The first phase analyzes a program and constructs descriptions of the standard classes of paths. The input data which causes the paths in a class of paths to be followed can be characterized by a subset of the assignments, loops, function calls and branch predicates in the paths. The second phase of the methodology constructs descriptions of the sets of input data which cause the different standard classes of paths to be followed. The descriptions generated by the second phase are implicit descriptions in the sense that they do not explicitly describe a set of data in terms of predicates and relations. They contain assignments and loops and other program-like constructs. The third phase of the methodology attempts to transform the implicit descriptions into equivalent explicit descriptions. An explicit description consists entirely of predicates and relations. In general, it is not possible to transform any implicit description into an explicit description. The fourth phase constructs explicit descriptions of subsets of the input data sets for which the third phase was unable to construct explicit descriptions. The fifth phase of the methodology generates input values which satisfy explicit descriptions. Explicit descriptions of numeric input data consist of systems of equalities and inequalities. Values which satisfy numeric explicit descriptions can be obtained by the application of inequality solution techniques.

3. Generation of Class Descriptions

(a) Boundary-Interior Test Paths. There are a potentially infinite number of paths through a program which contains loops. Only a finite number of these can be tested. One approach is to test the K shortest paths, for some fixed constant K. The K shortest paths approach is redundant and unpredictable. It causes intuitively similar tests to be carried out. It tests the "important" paths through some programs but not through others.

Another approach is to group the paths into a finite number of standard classes and to test one path from each class. In the loop reduction method paths are grouped into classes by ignoring iterations of loops. This method has two obvious disadvantages. It does not distinguish between alternative paths through a loop and it does not distinguish between the boundary and interior tests of a loop. A boundary test of a loop is a test which causes the loop to be entered but not iterated. An interior test causes a loop to be entered and then iterated at least once. Experience indicates that both the boundary and interior conditions of a loop should be tested.

The boundary-interior method generates separate classes of paths for alternative paths through and for the boundary and interior tests of a loop. Paths which differ other than in traversals of loops are grouped in different classes. Paths which differ only in traversals of loops are classified as follows. Suppose P_1 and P_2 are two paths which enter and leave a loop L. P_1 and P_2 are placed in different classes if:

(i) P_1 is a boundary test and P_2 an interior test of L

(ii) P_1 and P_2 enter or leave L along different loop entrance or loop exit branches

(iii) P_1 and P_2 are boundary tests and they follow different paths through L

(iv) P_1 and P_2 are interior tests and they follow different paths through L on their first iteration of L.

(b) Class Descriptions. The first phase of the methodology uses the boundary-interior approach to decompose a program into a finite set of classes of paths. It constructs program-like descriptions of the classes. Class descriptions consist of branch predicates, assignment statements, I/O statements and "FOR-loops". Figure 2 contains the description of the class of paths which test the interior of (i.e., iterate at least once) the loop in the program in Figure 1.

```
1   READ N
2   IF N < 0 GO TO 10
3   M ← 1
4   IF N = 0  GO TO 8
5   M ← M * N
6   N ← N -1
7   GO TO 4
8   PRINT M
9   HALT
10  PRINT -1
11  HALT
```

Fig. 1. Factorial program.

The FOR-loop notation in Figure 2 is used for denoting traversals of loops. The use of the FOR-loop notation makes it possible to introduce a variable into a description which denotes the number of times a loop is traversed.

The class descriptions for a program will contain common subdescriptions. A complete set of class descriptions can be represented in the form of a "description tree". Figure 3 contains the description tree for the factorial program in Figure 1.

(c) <u>Class Description Generation Process</u>. Phase one of the methodology reads through a program and constructs its class description tree. The phase one process has the structure of a recursive finite state automaton. Each copy of the automaton is associated with the processing of a loop in the program. When a subloop is discovered during the

```
READ N
N ≥ 0
M ← 1
N ≠ 0
M ← M * N
N ← N - 1
K1 ≥ 0
FOR I1 = 1 TO K1
     N ≠ 0
     M ← M * N
     N ← N - 1
N = 0
PRINT M
HALT
```

Fig. 2. Class description.

processing of some loop a fresh copy of the automaton is created for the processing of the subloop. When the processing of a loop has been completed, control is returned to a previously interrupted copy of the automaton. The structure of the process is such that loops must be properly nested.

The structure of the phase one process is described by the state diagram in Figure 4. As it reads through a program the process adds assignment statements, branch predicates, I/O statements and FOR-loops to the description tree. The process begins in the MAIN state. It continues in this state until a branching statement or the entrance to a loop is encountered. When it reaches a branching statement it constructs

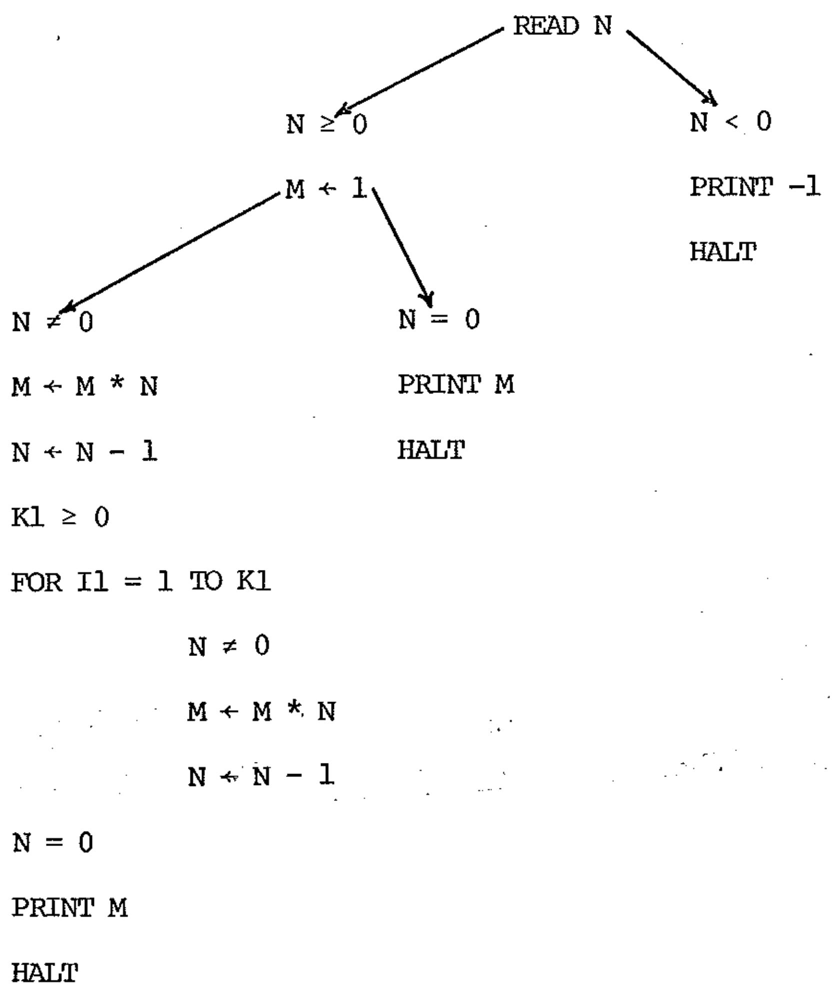

Fig. 3. Description tree.

branches in the description tree which correspond to statement branches. The BRANCH state chooses which branch to continue processing along. When a loop entrance is encountered the process enters the ENTRANCE state. It continues in the ENTRANCE state until a branching statement, the first

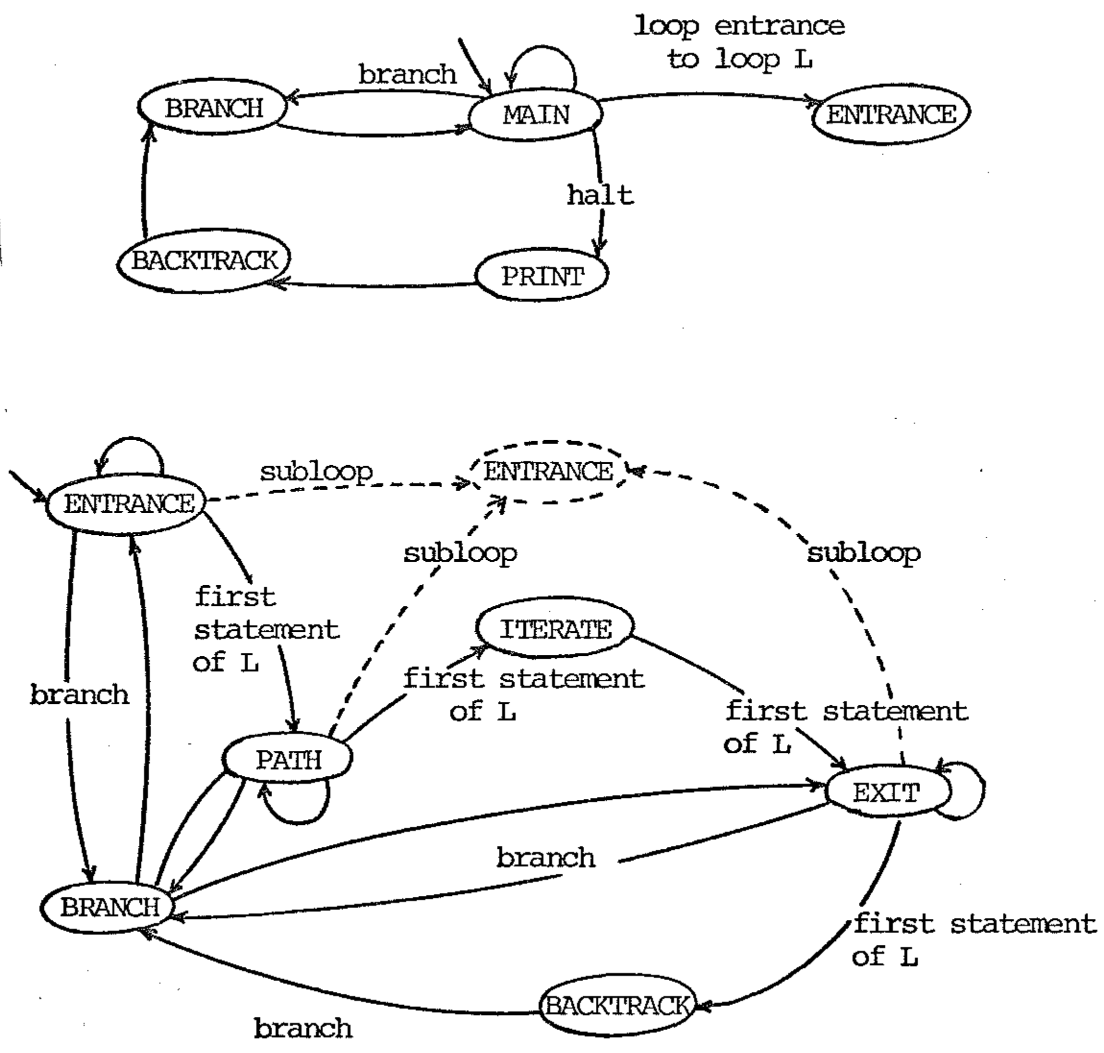

Fig. 4. State diagram for class description generation process.

statement in the loop or the entrance to a subloop is reached. If a branching statement is reached it sets up the appropriate branches in the description tree. If the first statement is reached the PATH state is entered. The PATH state sets up subpaths in the tree which correspond to the alternative paths which can be followed on the first iteration of the loop. If the entrance to a subloop is encountered a fresh copy of the automaton is created and entered at the ENTRANCE state. When the PATH state re-encounters the first statement of the loop being processed it passes control to the ITERATE state. The ITERATE state constructs a FOR-loop which describes all possible further iterations of the loop. The ITERATE state passes control to the EXIT state. The EXIT state "exits" from a loop. It constructs subpaths in the tree corresponding to the different paths through the loop from the first statement of the loop to some exit branch out of the loop. The BACKTRACK state causes control to return to some previously interrupted copy of the automaton. It does this by backtracking through the partially complete description tree until it encounters a node which has branches leading to uncompleted subpaths. It passes control to BRANCH to choose a branch along which to continue the tree construction process. The PRINT state prints out, or hands along to phase 2, a completed class description.

4. Implicit Input Data Descriptions

The input data which causes a path to be followed is the data which causes the predicates in the path to be satisfied. The predicates in a path, together with the input and computational statements wich affect the variables in the predicates, form an "implicit" description of the subset of the input domain which causes the path to be followed. Phase 2 of the methodology constructs implicit input data descriptions of the sets of data which cause classes of paths to be followed. It does this by extracting the predicates and predicate affecting statements from class descriptions. Figure 5 contains the implicit input data description for the class description in Figure 2.

```
N ← #1
N ≥ 0
N ≠ 0
N ← N - 1
K1 ≥ 0
FOR I1 = 1 TO K1
        N ≠ 0
        N ← N - 1
N = 0
```

Fig. 5. Implicit input data description.

Phase two can be described in two parts. The first part of phase two deletes the output statements from a class description and replaces all input statements by assignment statements. Each input statement in a program is assumed to read the next value in an input stream. The values in the input stream are represented by the dummy input variables #1, #2, ... Different notations can be developed for different kinds of program input. In Figure 5 the input statement "READ N" has been replaced by the assignment N ← #1. Special consideration must be given to input statements which occur inside loops. Similar techniques can be used for subroutine input.

The second part of phase two deletes all "unnecessary" assignment statements. It does this by reading backwards from each predicate. As it reads back it constructs lists of "predicate affecting" variables. It uses these lists to determine which assignment statements do not affect predicates and can be deleted from a description.

5. Transforming Implicit into Explicit Descriptions

(a) Explicit Input Data Descriptions. An explicit input data description for a FORTRAN program consists of a system of inequalities in input variables and constants. It is usually only possible to construct "partially" explicit descriptions. Partially explicit descriptions are simplified implicit descriptions. Like implicit descriptions, they contain assignments and FOR-loops as well as predicates and relations. Phase three of the methodology is a symbolic interpretation process which transforms implicit descriptions into explicit and partially explicit descriptions.

Phase three attempts to evaluate and delete the assignment statements and FOR-loops in an implicit description. An assignment statement is evaluated by substituting the current symbolic values of the independent variables in the statement into the statement. The expression on the right hand side of the resulting statement becomes the current symbolic value of the variable on the left hand side. Symbolic values are substituted for occurrences of variables in predicates and relations. Figure 6 contains a partially explicit description for the implicit description in Figure 5. The assignment $N \leftarrow \#1$ has been evaluated and the symbolic value #1 substituted for N in the predicates $N \geq 0$ and $N \neq 0$.

The assignment $N \leftarrow N - 1$ has also been evaluated.

```
N ← #1
#1 ≥ 0
#1 ≠ 0
N ← #1 - 1
K1 ≥ 0
FOR I1 = 1 TO K1
        N ≠ 0
        N ← N - 1
N = 0
```

Fig. 6. Partially explicit description.

The current symbolic value #1 - 1 of N cannot be substituted for occurrences of N in the FOR-loop because N is assigned a value in the loop. Phase three attempts to evaluate and delete FOR-loops by finding "closed forms" for iterative expressions. The FOR-loop in Figure 6 can be replaced by the closed form in Figure 7. Once the FOR-loops have been eliminated from a partially explicit description, further evaluation may be possible. The value of N computed in the assignment N ← #1 - 1 can be substituted for N in the closed form expression for the FOR-loop. A symbolic value for N can also be substituted into the predicate N = 0. Figure 8 contains the resulting description.

Some assignment statements can be deleted once they have been evaluated and others will remain in the resulting partially explicit description. Assignment statements which do not affect predicates in a partially explicit description can be deleted from the description. All of the assignment statements in Figure 8 can be deleted. Figure 9 contains the resulting description. In this case a completely explicit description is generated.

$N \leftarrow \#1$

$\#1 \geq 0$

$\#1 \neq 0$

$N \leftarrow \#1 - 1$

$K1 \geq 0$

$(N < 0 \vee N > K1 - 1)$

$N \leftarrow N - K1$

$N = 0$

Fig. 7. Description with closed form

$N \leftarrow \#1$

$\#1 \geq 0$

$N \leftarrow \#1 - 1$

$K1 \geq 0$

$(\#1 - 1 < 0 \vee \#1 - 1 > K1 - 1)$

$N \leftarrow \#1 - 1 - K1$

$\#1 - 1 - K1 = 0$

Fig. 8. Partially explicit description

$\#1 \geq 0$

$\#1 \neq 0$

$K1 \geq 0$

$(\#1 - 1 < 0 \vee \#1 - 1 > K1 - 1)$

$\#1 - 1 - K1 = 0$

Fig. 9. Explicit input data description

(b) Interpretation Problems. There are several problems which make interpretation a complicated process and which can prevent the construction of a completely explicit input data description. The problems result from the presence of array references and FOR-loops in an implicit description.

FORTRAN array references have the property that they may stand for different array elements, depending on the values of the indices in the reference. Suppose that the value of an index in a reference can only be determined at execution time. Then the symbolic interpreter may be unable to complete the evaluation of the statement in which the reference occurs. In the example in Figure 10 the interpreter can

```
A(K,2) ← 121
A(J,2) ← 144
X ← A(K,2)
```

Fig. 10. Indeterminate variable references problem.

evaluate the first two assignments and assign the values 121 and 144 to the variable symbols A(K,2) and A(J,2). If the interpreter is unable to determine whether or not K is equal to J at that point in the program it cannot assign the value 121 or 144 to X and cannot complete the evaluation of X ← A(K,2). The symbolic value of A(K,2) will be "indeterminate". The best the interpreter can do is to assign the symbolic values "A(K,2)" to X and refrain from deleting the assignments A(K,2) ← 121 and A(J,2) ← 144.

The interpretation process in phase three of the interpreter uses the concept of the "domain" and "range" of a variable symbol to determine when evaluation and deletion can take place. The domain of a variable symbol which occurs at some point in a program is the set of variables which that symbol may stand for at execution time. The range of the symbol is the set of possible values of the ranges of the variables in the domain of the symbol. The domain of an array reference is determined by the ranges of its indices.

There are three classes of FOR-loop interpretation problems. The first involves the substitution of values computed outside FOR-loops for variable references occurring inside FOR-loops. Suppose that a reference to a variable X occurs in a predicate or on the right hand side of an assignment inside a loop. Let X_0 be the value of X on entry to the loop. If X also appears on the left hand side of an assignment in the loop then the initial value X_0 of X cannot be "brought into" the loop and the assignment of X_0 to X outside the loop cannot be deleted from the description. This problem occurs in the example in Figure 5. The value #1 − 1 for N cannot be brought into the loop in the partially explicit description in Figure 6. In this particular example a closed form for the FOR-loop was discovered which allowed the substitution to be carried out later on in the evaluation process, and the assignment $N \leftarrow \#1 - 1$ to be deleted.

The second class of problems involves the substitution of values computed inside FOR-loops for variable references occurring outside FOR-loops. Suppose that a variable X is computed inside a loop and then referenced outside the loop. If there is no closed form for the iteratively computed value of X then the value of X cannot be "brought out" of the loop and the FOR-loop cannot be deleted. In the example in Figure 6 it was possible to construct a closed form for the iteratively computed value of N which could be "brought out" of the loop and substituted for the reference to N in the predicate N = 0. It will usually be difficult to construct closed forms. The construction of the closed form for even the simple

FOR-loop in Figure 6 requires the application of a relatively sophisticated closed form role.

The third class of FOR-loop interpretation problems involves the interpretation of "disjunctive" and "recurrence" statements. Suppose that a program contains a loop L and that the conditional statement "IF P THEN $X \leftarrow Y$" occurs inside L. Phase one of the methodology will distinguish between the paths through L for which P is true and those for which $\sim P$ is true. Each class description which is constructed by phase one will contain a description of a particular path through L and a FOR-loop which describes all possible iterations of L. The FOR-loop will contain the disjunctive statement $(P \wedge X \leftarrow V) \vee \sim P$. A disjunctive statement consists of a number of terms connected by the $\vee$ symbol. An assignment which occurs as part of a term in a disjunctive statement cannot be symbolically evaluated unless the interpreter is able to determine the truth values of the predicates in the statement. When the phase three interpreter encounters a disjunctive statement is marks the values of the assigned variables in the statement as "indeterminate".

In a recurrence assignment, the variable on the left hand side also occurs on the right hand side. The description in Figure 6 contains the recurrence assignment $N \leftarrow N - 1$. Recurrence assignments can occur both inside and outside FOR-loops and can be evaluated in the normal way when they occur outside a loop. They cannot be symbolically evaluated when they occur inside a loop. The phase three interpreter marks the values of assigned variables in recurrence assignments as indeterminate.

(c) Interpretation Process. The phase three interpretation process consists of two parts. In the first part the assignment statements and FOR-loops in a description are evaluated. Values of variables are substituted into predicates and relations. In the second part the unnecessary statements and FOR-loops in the evaluated description are deleted. The separation of evaluation and statement deletion simplifies the problem of determining if an assignment can be deleted from a partially explicit description.

The evaluation part of the interpretation process (the evaluator), has the structure of the recursive automaton in Figure 11. A new copy of the automaton is created whenever a sub FOR-loop is discovered during the processing of a description. Each of the states in the automaton is associated with a subprocess which makes a "symbol pass", "evaluation pass" or "closed-form pass" over an implicit or partially explicit description.

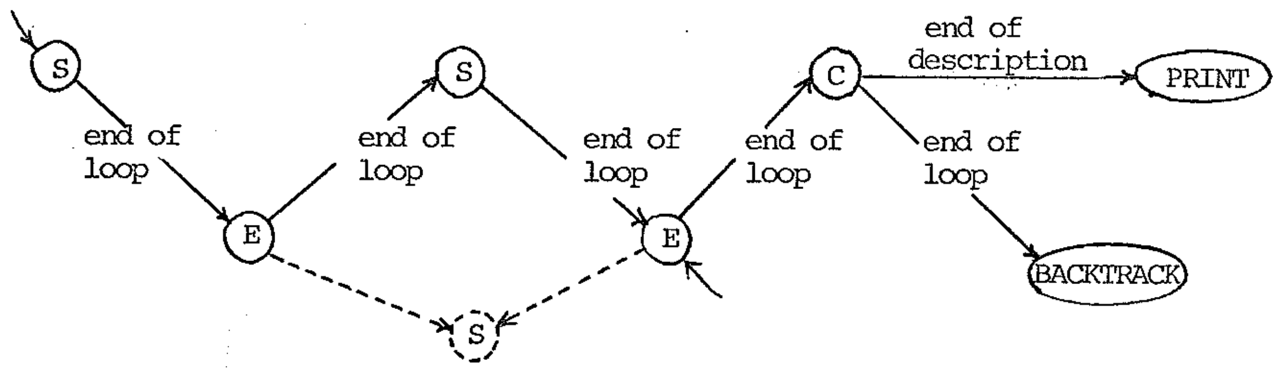

Fig. 11. State diagram for evaluation process

The evaluator begins in the E state at the first statement of a description. The E state uses an E-list to symbolically evaluate assignment statements and substitute values of variables into predicates and relation. A E-list is an ordered list of ordered pairs. The first element of each pair is a variable symbol and the second a symbolic value.

Each time a new symbolic value is computed for a variable an ordered pair is added to the end of the E-list. When the E-state encounters a subloop the evaluator creates a new copy of the automaton and enters it at the first S-state. It saves the interrupted copy of the automaton, together with its partially completed E-list.

The S-state subprocess creates "FOR-loop symbol lists" or S-lists. S-lists are lists of ordered pairs of variable symbols and statement numbers. An S-list for a loop contains an ordered pair for each variable symbol which occurs on the left hand side of an assignment in the loop. The evaluator uses the S-list for a loop to determine when a value of a variable can be "brought inside" a loop. Consider the example in Figure 12.

```
X ← 10
FOR I = 1 to N
        Y ← X
          .
          .
          .
X ← 20
```

Fig. 12. FOR-Loop

Suppose that the S-list for the loop has be constructed and that the evaluator is attempting to evaluate the assignment Y ← X. If the S-list does not contain an entry for X then the value 10 can be brought into the loop and substituted for the occurrence of X in the assignment. If the S-list does contain an entry for X then the value of X is indeterminate in the loop.

When the S-state of an automaton reaches the end of a FOR-loop control passes to the following E-state of the automaton. The E-state uses the computed S-list for the loop, an E-list which it constructs as it goes along, and the partially completed E-lists of the outer loops to evaluate the assignments in the loop. It uses the S-list to determine when it is possible to bring a value from an outer loop E-list into the loop which is being processed.

After two S and E passes have been applied in succession to a FOR-loop, the C-state of the automaton is entered. The C-state attempts to find a closed form for the evaluated FOR-loop produced by the two S-E passes. It attempts to find closed forms for all of the iteratively computed values and predicates in the loop.

When a C-pass reaches the end of a loop control passes to the BACKTRACK state. The BACKTRACK state returns control to the interrupted automaton associated with the next enter FOR-loop. If the loop is at the top level of the description control returns to the "top level" automaton. The process halts when the C-state in the top level automaton reaches the end of the implicit description.

One of the interesting features of implicit description evaluation is the necessity for repeated applications of S and E passes to a FOR-loop. The evaluator described in Figure 12 carries out two S-E passes in succession to each loop each time the loop is encountered. The reason why more than one S-E pass is necessary is that processing on some earlier portion of a FOR-loop may be blocked by the presence of unevaluatable statements later in the loop. Once the statements which come later in the loop have been evaluated it may be possible to carry out the blocked processing on a second S-E pass of the loop. Consider the example in Figure 13. On the first S-pass over the loop an S-list containing the variable symbols T and A(X) will be constructed. On the first E-pass the evaluator will attempt to evaluate the assignment T ← A(4). The evaluator will not be able to bring the value A(4) = 3 into the loop since A(X) will be in the S-list and it will not be able to determine at that point that A(X) ≢ A(4). When the assignment A(X) ← 6 is encountered the evaluator will be able to bring the value X = 2 into the loop since X will not be in the loop's S-list. The first S-E pass will generate the partially explicit description in Figure 14.

```
X ← 2
A(4) ← 3
FOR I = 1 to N
        T ← A(4)
        A(X) ← 6
X ← 10
```

Fig. 13. FOR-Loop.

```
X ← 2
A(4) ← 3
FOR I = 1 TO N
        T ← A(4)
        A(2) ← 6
X ← 10
```

Fig. 14. FOR-Loop after first S-E pass.

The S-list constructed during the second S-E pass will contain the symbols T and A(2). The value A(4) = 3 will be brought into the loop since the symbol A(4) cannot be equivalent to any of the variable symbols that will be in the S-list. The second S-E pass will generate the partially explicit path in Figure 15.

```
X ← 2
A(4) ← 3
FOR I = 1 TO N
        T ← 3
        A(2) ← 6
X ← 10
```

Fig. 15. FOR-Loop after second S-E pass.

Although it has not been proved, experience indicates that two S-E passes are sufficient to complete the S-E processing of any FORTRAN implicit path description. More passes may be necessary for programming languages containing other kinds of program and data structures.

The deletion part of the interpretation process deletes assignment statements which no longer affect predicates. The deletion part of the interpreter works in the same way as the deletion subprocess in phase two of the methodology. The deletion process in phase two deletes assignment statements which do not affect predicates in class descriptions. The phase three deletion process deletes assignment statements which do not affect predicates in evaluated partially explicit descriptions. In both cases the deletion process constructs lists of predicate-affecting variables by reading backwards from each predicate in a description. The process uses these lists to determine which assignment statements must be retained.

6. Explicit Subset Descriptions

Phase three of the methodology transforms implicit descriptions into explicit and partially explicit descriptions. Implicit descriptions which contain FOR-loops will usually be transformed into partially explicit rather than explicit descriptions. It is usually very difficult to find closed forms for and to eliminate FOR-loops from implicit descriptions. Implicit descriptions which do not contain FOR-loops can almost always be transformed into explicit descriptions.

Each of the explicit and partially explicit descriptions which are generated by phase three of the methodology describes a set of input data. Phase four of the methodology constructs explicit descriptions of subsets of the sets which are described by partially explicit descriptions. It constructs subset descriptions by traversing the FOR-loops in partially explicit descriptions. Suppose that a partially explicit description D contains a FOR-loop whose loop upperbound is a variable $K \geq 0$. D describes the set of all input data which satisfies the predicates in D for all feasible choices of K. Loop-free descriptions of subsets of D can be constructed by choosing particular values of K. If D contains disjunctive statements, subset descriptions consisting of simple sequences of predicates and assignments can be constructed by choosing a particular term in each disjunctive expression. Phase four constructs explicit subset descriptions by choosing particular values of loop bounds and particular terms in disjunctive statements.

Figure 7 contains a closed form for the FOR-loop in the example in Figure 6. Suppose that the evaluator had been unable to construct the closed form for the FOR-loop. Then phase three would have generated the partially explicit description in Figure 16. Each choice of a non-negative

```
#1 ≥ 0
#1 ≠ 0
N ← #1 - 1
K1 ≥ 0
FOR I1 = 1 TO K1
        N ≠ 0
        N ← N - 1
N = 0
```

Fig. 16. Partially explicit description containing FOR-Loop.

integer value for K1 corresponds to a different subset of the set described by the partially explicit description. Figure 17 contains the subset description corresponding to the choice K1 = 0. The description in Figure 17 contains no FOR-loops and can be evaluated to produce the explicit subset description in Figure 18.

A description is feasible if there are values in the input domain which satisfy the descriptions. Infeasible descriptions describe the empty subset of the input domain.

```
#1 ≥ 0
#1 ≠ 0
N ← #1 - 1
N = 0
```

Fig. 17. Partially explicit subset description.

$$\#1 \geq 0$$

$$\#1 \neq 0$$

$$\#1 - 1 = 0$$

Fig. 18. Explicit subset description.

Particular choices of values for loop bounds and terms in disjunctive statements can result in the generation of infeasible subset descriptions. If a partially explicit description is itself infeasible then all choices of loop bounds and disjunctive terms will result in infeasible subset descriptions. Phase four of the methodology attempts to choose loop bounds and disjunctive terms in such a way that the resulting subset description is feasible whenever the original partially explicit description is feasible.

Two general techniques can be used to help ensure the generation of feasible subset descriptions. The predicates which constrain the loop bounds in a partially explicit description form a loop bound subdescription. The first technique is to only choose loop bound values which satisfy loop bound subdescriptions. These subdescriptions will often be simple loop-free systems of predicates from which loop bound values can be easily generated. The subdescription constraining the loop bound in Figure 16 consists of the single predicate $K1 \geq 0$. If a subdescription's minimal solution is always chosen then the resulting subset description will be as short as possible. The second technique is heuristic search. If a subset description is infeasible then a new subset description can be generated by choosing new loop bound values or disjunctive terms. Different heuristics can be used to guide the search through the set of possible choices. Suppose, for example, that a variable in a subset description is constrained by contradictory predicates and that in the original partially explicit description the value of the variable is set inside a FOR-loop. If the FOR-loop was iterated K times in forming the subset description then a new subset description can be generated by iterating the loop K+1 times. Other rules can be developed for other kinds of feasibility problems.

7. Generation of Test Cases

Phase five is the test data generation phase of the methodology. It divides standard classes of paths into three sets: those for which it can generate test data, those for which it can determine infeasibility, and those for which it can neither generate test data nor determine infeasibility. In general, since the test data generation problem is unsolvable, this is the best that can be expected from a test data generation methodology.

Phase five is an integrated collection of inequality solution techniques that can be applied to all of or parts of explicit descriptions for FORTRAN programs. The techniques are applied to complete descriptions to generate test cases and to subdescriptions to check feasibility. If a subdescription is infeasible then the description is infeasible. The phase five techniques are applied to both the explicit descriptions which are generated by phase three of the methodology and to the subset descriptions generated by phase four.

There are several well defined methods for solving classes of inequalities which do not contain function calls, subroutine calls, and array references with variable indices. Some of the methods are effective in the sense that they always produce solutions. Others are partially effective, they produce solutions to some systems of inequalities but not to others. Phase five includes both effective and partially effective methods. It uses a straightforward effective method for solving linear systems in one variable. Kuhn's method [9] is used to produce solutions for linear real valued systems in several variables. A method developed by Singhania and described in [10] is used to produce solutions for non-linear systems in one variable of degree less than five. Both Kuhn's and Singhania's method are effective. A large number of the descriptions which are generated by phases three and four of the methodology can be solved using these and other effective techniques. The explicit subset description in Figure 18 can be easily solved using the method for linear systems in one variable.

Phase five uses partially effective methods for solving general non-linear system and integer valued systems. The basic method is backtrack search. The backtrack search method constructs a sequence of partial solutions to a system. A partial solution to a system is a set of values for some subset of the variables in the system which does not contradict the relations constraining those variables. The method orders the variables in a system into a sequence. It begins by finding a partial solution for the first variable in the sequence. It then attempts to extend this partial solution by choosing a value for the next variable in the sequence. It continues until a complete solution has been constructed If it is unable to extend a partial solution at some stage it "backs up" and attempts to change the previous partial solution. If it backs up to the initial single variable system and exhausts the set of all possible values which satisfy the constraints on that variable then the system is unsolvable.

The backtrack approach can be applied directly to the solution of linear integer valued systems. Kuhn's method is used to generate a sequence of linear systems, each of which contains one less variable than its predecessor in the sequence. Any solution to a member of the sequence is a partial solution to its predecessor. The first member of the sequence is the original system. The last member is a single variable system. The method begins by choosing an integer solution to the single variable system. It uses this solution to reduce the second to last system to a single variable system. It then attempts to construct integer solutions to this system. The integer solution to the two variable system can be used to reduce and solve the three variable system, and so on. If some reduced system has no integer solution the method backs up and attempts to construct a different integer solution to the previous reduced system in the sequence.

The backtrack method can also be applied to non-linear systems. The non-linear system is replaced with a linear system by substituting dummy linear variables for occurrences of variables raised to powers greater than one. Kuhn's method is used to generate a sequence of partial solutions to the linear system. At each stage the method checks to see that the partial solution to the linear system does not contradict the substitution relationships. If the substitution relationships are not contradicted then the next partial solution in the sequence is constructed. If some relationship is contradicted then the method backs up and attempts to construct an alternative linear partial solution. The backtrack method is illustrated for the non-linear integer valued system in Figure 19.

$$\begin{aligned} x - 4y + 6y^2 &\geq 2 \\ -x + 2y - 2y^2 &\geq 0 \\ -x + 5y - 5y^2 &\geq -1 \end{aligned}$$

Fig. 19. Non-linear integer valued system.

The substitution $z = y^2$ reduces the non-linear system to the linear system in Figure 20.

$$\begin{aligned} x - 4y + 6z &\geq 2 \\ -x - 2y - 2z &\geq 0 \\ -x + 5y - 5z &\geq -1 \end{aligned}$$

Fig. 20. Linear system.

Kuhn's method can be applied to the system in Figure 20 to produce the sequence of systems in Figure 21.

$$\begin{aligned} x - 4y + 6z &\geq 2 \\ -x - 2y - 2z &\geq 0 \\ -x + 5y - 5z &\geq -1 \end{aligned} \qquad \begin{aligned} -2y + 4z &\geq 2 \\ y + z &\geq 1 \end{aligned}$$

Fig. 21. Sequence of partial solution systems.

An initial linear partial solution can be constructed by choosing a value for z which satisfies the last system in the sequence. Suppose z=1 is chosen. This can be used to reduce the preceeding system to the single variable system in Figure 22.

$$\begin{aligned} y &\geq 1 \\ y &\geq 0 \end{aligned}$$

Fig. 22. Reduced system.

The initial partial solution can be extended to a two variable partial solution by choosing a value for y which satisfies the reduced system. Suppose $y=0$ is chosen. Although $(z=1, y=0)$ is a partial solution to the linear system it contradicts the substitution relationship $z=y^2$.. The method will backtrack at this point and choose a new value for y. If $y=1$ is chosen then a complete solution to the original system will be generated.

In order to be able to generate test data for other than the lowest level functions and subroutines in a program, phase five of the methodology must be capable of solving systems containing functions and subroutine calls. In certain special cases a function call can be conveniently replaced with an equivalent subsystem of inequalities which does not contain the call. The only general technique which has been developed is a variation of the backtrack search method. Suppose that a system contains a single subroutine call. The system is first partially solved to find a set of bounds on the input to the subroutine and a set of bounds on its output. A set of input values which satisfies the input bounds is chosen and the subroutine is evaluated. If the resulting output satisfies the output bounds then a solution to the part of the system which affects and is affected by the subroutine has been discovered. If the output does not satisfy the bounds the method backtracks and a new set of input values is chosen. The method can be extended to systems containing more than one subroutine or function call. It cannot be easily applied to systems in which there are complicated interactions between subroutine calls.

8. Conclusions and Future Research

The methodology which is described in the preceeding sections can be used to generate data for programs which must be completely tested.

The boundary interior approach to the classification of program paths which is used in phase one permits the selection of a finite yet intuitively complete set of test cases. The methodology can be used to build a system which will automatically generate test data for some classes of paths, determine the infeasibility of other classes and print out partially explicit descriptions of the input data which causes the remaining classes of paths to be followed.

Parts of the methodology can be used for purposes other than the automatic generation of test data. Phases one and two can be used to print out descriptions of the standard classes of paths through a program and implicit descriptions of the input data which causes the paths to be followed. These two phases could be used as part of an interactive testing system.

The first four phases of the methodology could be used as part of an automatic partial program correctness system. In the boundary-interior approach to partial program correctness the correctness of one path from each standard class of paths is proved. Phases one through four can be used to generate simple loop-free path descriptions. The correctness of a loop-free program path is considerably easier to prove than the correctness of an entire program.

Many of the basic problems of test data generation were discovered during the design of the methodology. The most important involve the loop structure, subroutine calls and complexity of arithmetic expressions in programs. It might be interesting to characterize the set of testable program schemata. A schema is testable is all interpretations of the schema yield testable programs. A testable program is a program for which it is possible to automatically generate a complete set of test data.

Present research plans include the implementation of phases one and two of the methodology and an extensive investigation of its general applicability. Further study of the subroutine problem is also planned. General techniques must be developed for solving predicate systems which contain calls to user defined subroutines and functions.

Acknowledgements

Several students at the University of California at Irvine participated in the research which is described in this paper. Ram Singhania carried out a survey of inequality solution techniques. His work was fundamental to the design of phase five of the methodology. Donald Mies patiently and carefully compiled statistics on the structure of FORTRAN programs. The statistics were used to guide the direction of the research. Steven Williams experimented with the boundary-interior classification scheme.

The author would like to acknowledge the many helpful comments of Rona Stillman and Selden Stewart of the National Bureau of Standards. Dr. Stillman suggested the use of a regular expression notation for the representation of class descriptions. This suggestion influenced the design of the algorithm for generating class descriptions.

The research described in this paper was associated with a McDonnell Douglas Astronautics project in program testing which was funded by the National Bureau of Standards. The project was carried out under the supervision of Z. Jelinski and Leon Stucki of the Information Sciences Division of McDonnell Douglas. The author wishes to express his gratitude to Mr. Stucki and Mr. Jelinski for their encouragement and advice throughout the research.

References

[1] B. Elspas, K.N. Levitt, R.J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness", Computing Surveys, vol. 4, pp. 97-147, June 1972.

[2] W. C. Hetzel, ed. Program Test Methods. New York: Prentice Hall, 1972.

[3] 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE, 1973.

[4] L.G. Stucki, "Automatic Generation of Self-Metric Software", in 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE, 1973, pp. 94-100.

[5] J.R. Brown, A.J. DeSalvio, D.E. Heine, and J.G. Purdy, "Automated Software Quality Assurance", in Program Test Methods. New York: Prentice Hall, 1972, pp 76-92.

[6] K.W. Krause, R.W. Smith, and M.A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis", in 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE, 1973, pp. 18-22.

[7] M.R. Paige and E.E. Bolkovitch, "On Testing Programs", in 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE, 1973, pp. 23-27.

[8] C.V. Ramamoorthy, R.J. Meeker, and J. Turner, "Design and Construction of an Automated Software Evaluation System", in 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE, 1973, pp. 28-37.

[9] H.W. Kuhn, "Solvability and Consistency for Linear Equations and Inequalities", American Mathematical Monthly, vol. 63, pp. 27-38, April 1956.

[10] W.E. Howden, L.G. Stucki, and Z. Jelinski, "Final Report: Methodology for the Effective Test Cast Selection, Part I", McDonnell Douglas Astronautics Report MDC G5301, Jan., 1974.

Affiliation of Author

Department of Information and Computer Science
University of California
Irvine, California 92664

Acknowledgement of Financial Support

The research described in this paper was carried out in cooperation with the McDonnell Douglas Astronautics program in the research and development of program validation software. The research was funded by the National Bureau of Standards.

Figure Captions

<u>Address for Correspondence</u>

Dr. William E. Howden

Department of Information & Computer Science

University of California

Irvine, California 92664

Index Terms

Software validation, program testing, automatic generation of test cases, flowchart analysis, analysis of programs, program paths, systems of predicates, inequality solution techniques.