AUTOMATIC TEST DATA SELECTION TECHNIQUES †

Lori A. Clarke

COINS Technical Report 78-25
September 1978

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

† This paper appears in the Infotech State of the Art Report on Software Testing, Infotech International, 1979.

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

I. Introduction

In the last decade, there has been an increased awareness of the need for more reliable software. This concern has come about because of our demand for larger, more complex systems and because of our growing dependence on software. This dependence may be merely a convenience of modern society such as a department store billing system, or it may be of tremendous benefit, such as a cardiac monitoring system. While errors in the former system may be annoying, errors in the latter system may be catastrophic. The desired level of confidence and the amount of resources devoted to establishing that level of confidence vary from system to system and may even vary between parts of the same system.

Various methods of improving software reliability are being explored. These include work going on in language design, programming methodology, and program validation. Though there have been several proclamations that program correctness is just around the corner (about a five to ten year period), we now realize that, in general, absolute correctness is an unattainable goal. We have made some advances in improved program reliability and will probably continue to make progress. However, all aspects of the software development process are prone to errors. Since we cannot guarantee program correctness, we must expect that some errors will not be detected until after a system is released.

In this paper, we will discuss methods of program testing. Testing is defined here to be the execution of a program in its natural environment. Testing requires the selection of data to actually exercise the program. The results must then be analyzed and confirmed. If an erroneous result is uncovered, debugging in initiated.

Programmers have historically used testing to convince themselves and others that their software works. Though testing, like other reliability methods, has limitations, it has one major advantage: it is the only method in which the actual behavior of the software can be observed. Thus, errors or oversights in the supporting environment, which includes the available validation tools, translators, operating system and hardware, may be detected. Moreover, the actual performance can be evaluated for efficiency and usefulness.

Section II of this paper describes some of the limitations of program testing and outlines how a combination of methodologies and program analysis techniques may assist in the selection of test data. The remainder of the paper describes ATTEST, an automatic test enhancement system, that is under development. ATTEST aims to offer a more systematic approach to testing and attempts to relieve the user of some of the more tedious aspects of test data selection.

II. Testing Methodologies

Testing is usually given a low priority and often relegated to the least experienced programmers. Many software errors are not found until after the system is released, thus contributing to the high cost of maintenance. In fact, one study found that only one third of the statements were ever executed during the testing stage [1]. We need to replace this haphazard approach to testing with a systematic approach so that some quality assurance can be given for a system. Moreover, we need to employ experienced personnel familiar with the specifications, design, and implementation of the system. Personnel cognizant of decisions made during the software development stages can better test the ramifications of these decisions.

Only in some rare and usually uninteresting cases is it possible to efficiently enumerate and test a program on all possible input values for that program. Testing methodologies must therefore support techniques for selecting a subset of the data sets from the total input domain of a system. An inherent limitation of testing is that any subset of the input domain may not find all the errors in a program. A testing methodology should increase the likelihood of uncovering errors in the software. This section considers two testing methodologies: functional and structural. Each will be briefly described and an argument made as to why both forms of testing must be considered. The last part of this section outlines the benefits of combining these methodologies with a software testing tool.

Functional testing is sometimes referred to as black box testing or specification testing. In functional testing, only the functions of the software are considered and not the actual implementation. Data sets are selected to exercise all the functions of the software. These functions are referred to as subcases or special cases and include the general or typical cases as well as error detection cases.

In order to do functional testing well, the tester should be very familiar with the specifications. When a specification document is available, the various special cases may be explicitly listed. Goodenough and Gerhart [2] have shown how a decision table specification facilitates test data selection. Though decision tables are not applicable to many types of programming problems, when they are available, they are extremely useful for data selection.

Functional testing has several limitations. The main drawback is an inability to determine if the test set is complete. Just as we can not be sure a specification is complete, we can not be sure a functional test set exercises all the subcases. Another problem is that the number of test cases for functional testing may exceed the resources for testing. The example in [2] demonstrates that even a simple program may have an excessive number of test cases.

Structural testing requires analyzing the program's data representations and control paths. A program can be represented as a directed graph in which the nodes represent the statements and edges represent the possible control transfers. During structural testing, data sets are selected to give good coverage of the program graph. For example, the coverage criterion may be the execution of all statements. While this coverage criterion has many drawbacks, it surpasses the program coverage that is typically achieved and increases the probability that many blatant errors will be caught before the program is released. Other structural testing criteria are discussed in Section VI.

Structural testing should also include testing special constructs in the code such as the boundary conditions of the internal data representations and of loops. For data representations, this involves at least testing for underflow and overflow. For loops, this generally involves exercising a loop the minimum and maximum number of times.

Even though structural testing guarantees some kind of minimal coverage of the program, it does not guarantee correctness of the tested components. A statement may be tested and still be in error. In addition, structural testing may not uncover a missing subcase. Finally, structural testing may also result in an excessive number of data sets. Depending on the application and resources, testing may have to be restricted to critical areas of the code.

The appendix contains a program that looks up an inventory number in a merchandise table. Functional testing would require data sets where inventory number is present in the table, is not present in the table, and is illegal. Structural testing would also detect these cases, and in addition, require testing the table representation. In the example, a linear search algorithm is used. Two additional test cases are needed with the inventory number set to the first element in the table and to the last element in the table. Another search algorithm would probably require different data sets.

Functional testing has the advantage of testing special cases that may have been overlooked or incorrectly implemented. Structural testing has the advantage of uncovering special cases that were included in the code but not in the specifications. It also has the advantage of concentrating on implementation problem areas. For best results in detecting errors, both methods should be combined.

Top down, bottom up, and unit testing are testing methodologies that are often proposed. The choice between these methods is often determined by the software development method that is imposed. Functional and structural testing can easily be used in conjunction with all three.

Functional testing can be facilitated by the use of formal specifications [3]. Structural testing can be assisted by a system that analyzes the program structures, monitors the test runs, and then aids in the selection of test data to execute the unexercised sections of the program. This can be particularly useful in large systems containing many special cases to handle erroneous data or erroneous computations. The actual program paths to these sections of code may be short but it still may be tedious to manually analyze each one. A more efficient method is to allow the human tester to concentrate on the critical areas of the code and use an automated testing tool to assist in data selection for the more mundane portions of the code. The following sections describe an automatic test selection system that attempts to generate data for unexercised sections of the code as well as data to exercise some of the implementation dependent special cases such as array indices out of bounds and division by zero errors.

III. ATTEST

ATTEST, an automatic test enhancement system, can either augment previously selected data sets or select all the test data for a program. Ideally, a testing group would first develop a set of test data using both the functional and structural methods. Thus, test cases based on the specifications and implementation would first be attempted. ATTEST would then determine the poorly tested areas of the program, if any, and generate data to exercise these areas.

In a large system, there are usually numerous checks for inconsistent data or results. Manually developing test cases for all these situations can be extremely tedious. However, not testing these situations can be costly since untested code frequently results in errors being discovered once a system is delivered. The compromise situation of combining manual and automatic test selection techniques is perhaps the best. The testing group can use their expertise to exercise the program and then a system like ATTEST can be used to guarantee a minimal level of coverage in the code and specifications.

ATTEST is composed of three major components: path selection, symbolic execution, and test data generation. The path selection component is concerned with selecting program paths that satisfy a testing criterion. Usually this involves choosing paths that contain segments of code in need of further testing. The symbolic execution component analyzes each chosen path. During symbolic execution, a path's computations are represented, the range of possible input values is constrained, and some error analysis is done. The test data generation component checks the input value constraints for consistency and, if they are consistent, generates test data that would drive execution down the selected path. It is then necessary to test the program with the generated test data to confirm the results. Figure 1 shows the interrelationships between the ATTEST components.

ATTEST analyzes programs written in ANSI FORTRAN but the general methods employed are relevant to any algebraic language. In a preprocessor stage, ATTEST translates the FORTRAN source statements into a machine

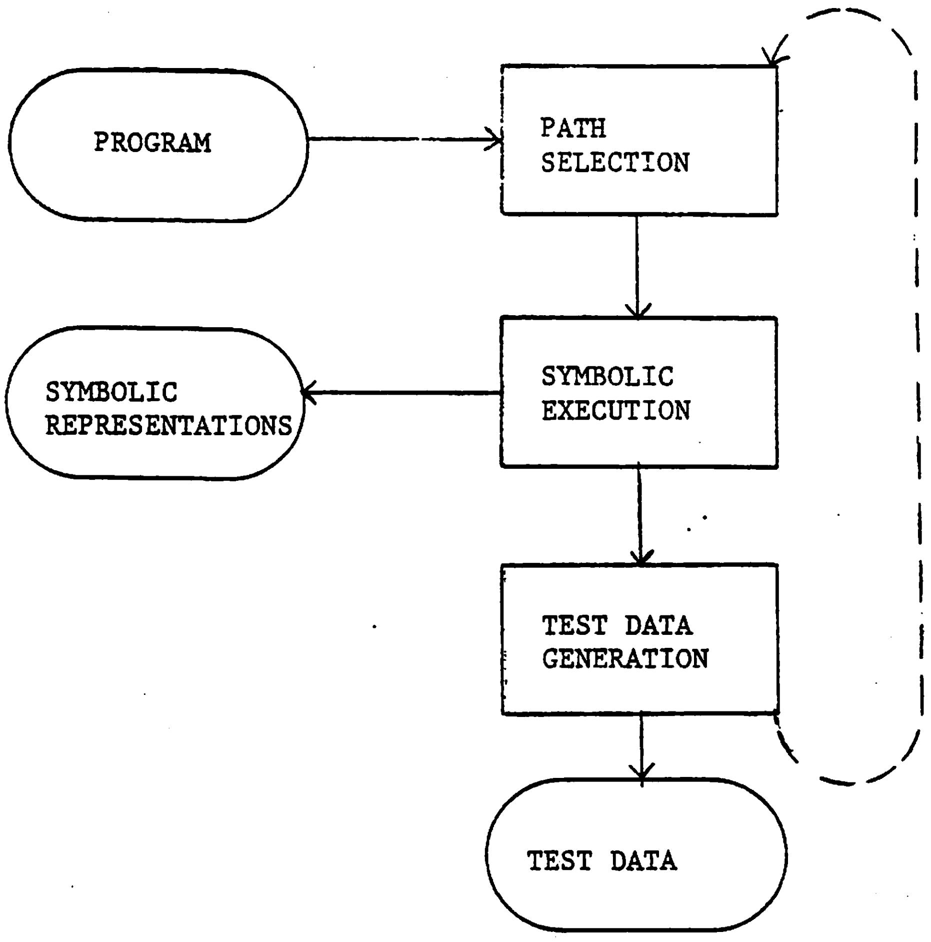

PROGRAM
PATH SELECTION
SYMBOLIC REPRESENTATIONS
SYMBOLIC EXECUTION
TEST DATA GENERATION
TEST DATA

Figure 1

independent assembly type language and represents the program structure as a directed graph. The analysis is then done on this translated form and not on the original code.

In the next three sections, each component of ATTEST will be described. Since path selection depends on the results of the other two components, it is described last.

IV. Symbolic Execution

Symbolic execution derives its name from the technique of representing input values by symbolic values instead of by actual numeric values. For example, after the statement READ A,B is symbolically executed, A and B would have symbolic values, say i_1 and i_2, associated with them. Any subsequent expression that references A or B would use the corresponding symbolic representation. For example, consider the following code segment:

```
1    READ RADIUS
2    PI=3.14
3    TWOPI=2.*PI
4    AREA=PI*RADIUS**2
5    CIRC=TWOPI*RADIUS
```

After symbolic execution of statements 1 to 5, the symbolic representations are PI=3.14, TWOPI=6.28, AREA=3.14*i_1**2, and CIRC=6.28*i_1, where i_1 represents the input value for RADIUS. Note that in general, a variable's symbolic representation changes whenever it is assigned a new value on a path and at any statement, a variable's symbolic representation depends on the path taken to arrive at that statement.

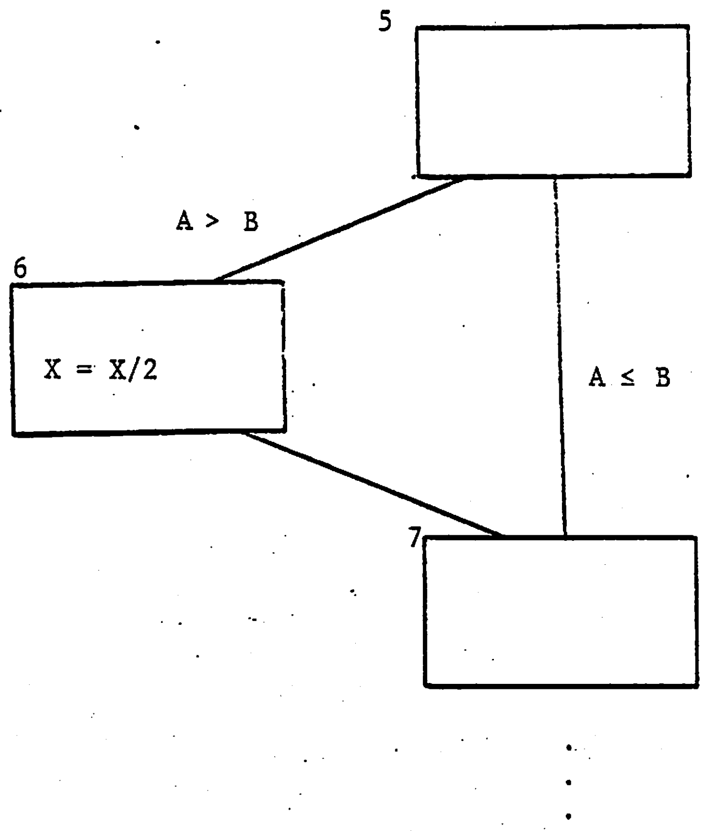
Symbolic execution is a powerful testing tool. After executing a series of statements, the resulting symbolic representations can be examined to determine the correctness of the path. While normal execution shows particular output values that result from particular input data, symbolic execution represents all computations for a path regardless of the input data. In other words, the resulting computations for a potentially infinite number of test runs with different input values may be represented by the symbolic representations for one program path. In addition to representing output for a class of input data, the symbolic representations are often more informative than numberic values. To use a trivial example, assume a program path computes A*A instead of 2*A. If the program is tested with data that always results in A having the value 0, then no error would be detected. However, if symbolic execution is used, the error is easily detected from the symbolic representation. In an experiment conducted by Howden, 68% of the errors in a set of programs were found using symbolic execution [4].

An unfavorable consequence of symbolic execution is that the symbolic representations are sometimes too long and complex to be meaningful. Even when a symbolic representation is short, the tester may not detect an error in a representation. Consider the case where the program should compute (A+B)/C and instead computes A+B/C. The missing parenthesis may go unnoticed, especially if the tester is the original coder who made the mistake. Another major drawback of symbolic execution is that it must be applied to completely specified program paths including the number of iterations for each loop. In general, a program may have an infinite number of program paths and only a subset will be chosen for analysis. The path selection section discusses several strategies for selecting a subset of the paths.

V. Test Data Generation

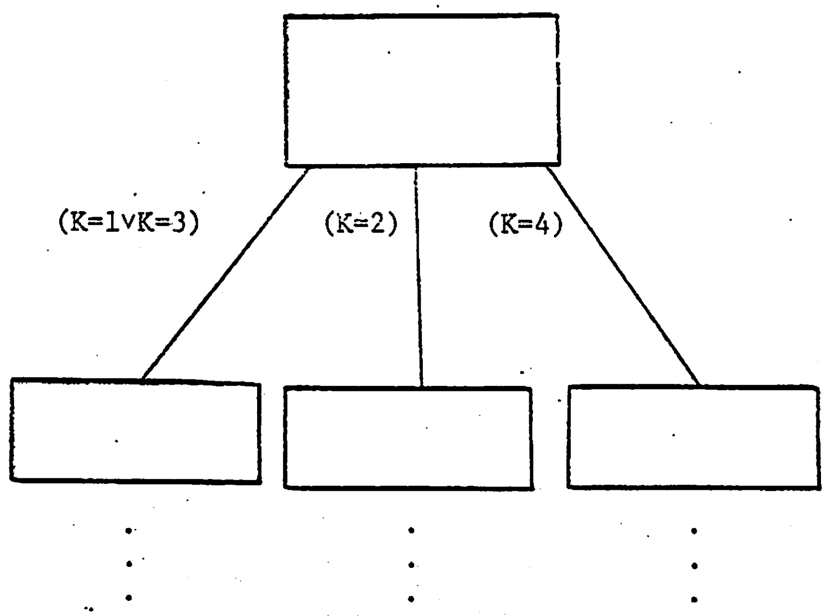
In the test data generation component of ATTEST, we are concerned with selecting data that would cause the execution of the selected path. The generated data must therefore satisfy all the conditional branches on the path. We use symbolic execution to represent the conditional branches on the path and then, in the test data generation component, we attempt to generate a satisfactory data set. In this section, we will first discuss the symbolic representations of the conditional branches and then describe the test data generation method.

Conditional statements are represented in the program graph by relational and logical expressions that annotate the graph edges according to their appropriate interpretations. Two examples using a FORTRAN logical IF and computed GO TO statements are given in Figures 2 and 3. In Figure 2, the nodes in the graph are numbered to easily distinguish the two paths 5,6,7 and 5,7. The symbolic execution of the path 5,6,7 results in the predicate $S(A) > S(B)$ while the symbolic execution of the path 5,7 results in the predicate $S(A) \leq S(B)$, where $S(A)$ and $S(B)$ denote the symbolic representations of A and B after execution of node 5 on the path. A predicate is associated with each conditional statement in a complete program path. We denote the symbolic representation of the ith conditional statement in a path by P_i. In order to



IF(A.GT.B) X=X/2

Figure 2



IF(10,20,10,30),K

Figure 3

actually execute a path, each new predicate P_{i+1} must be consistent with the conjunction of all predicates previously encountered on the path (P_j, $1 \le j \le i$). If the new predicate is inconsistent, the path is infeasible or nonexecutable. The symbolic representation of a predicate may evaluate to the value true or false, just as some variables have constant symbolic representations such as the variable TWOPI in the previous code segment. A predicate that evaluates to the constant true is consistent with all previous predicates on the path while a predicate that evaluates to the value false is inconsistent.

When the symbolic representation of a predicate does not evaluate to a constant, the test data generation component attempts to determine if the new predicate is consistent with the existing set of predicates. Two techniques, axiomatic and algebraic, have successfully been used to determine predicate consistency.

The axiomatic approach uses first order predicate calculus to determine if the most recently encountered predicate is consistent with the conjunction of the existing set of predicates. A theorem prover is used to determine consistency. This method is subject to the limitations of automatic theorem proving [5].

The algebraic approach represents each predicate as an equality or inequality expression. The set of predicates forms a set of constraints over the input domain. If any solution exists to the set of constraints, then the new constraint is consistent with the existing set of constraints. Since ATTEST is concerned with actually generating input data and not just consistency, the algebraic approach is used. When the end of a path is encountered, any solution to the set of constraints is a data set that would cause execution of the path. Figure 4 shows the graph for a segment of a program and Figure 5 shows the resulting constraints and domains from symbolically executing four paths. To simplify this example, none of the variables in the conditional statements have been modified and the variables X and Y have been symbolically represented by x and y. Note that on the path 1,2,4, either branch (4-5) or (4-6) is consistent with the existing set of constraints while on the path 1,2,3,4, only branch (4-6) is consistent. A conditional statement on a path that results in more than one consistent predicate is referred to as a decision point. The path selection component determines which branch to select at decision points.

There are various algebraic methods that could be applied to the set of constraints to determine consistency. ATTEST uses a linear programming system that accepts both real and integer variables [6]. Both the algebraic and axiomatic approach have similar limitations and, in general, cannot handle all possible situations. The algebraic approach has the advantage of providing actual data values for testing. Though ATTEST currently supplies only one data value for a path, there has been some work done in choosing a set of data points to test each path which would increase the probability of detecting errors on the path [7].

VI. Path Selection

Symbolic execution systems have historically given a low priority to path selection [8,9,10]. The various methods of path selection that

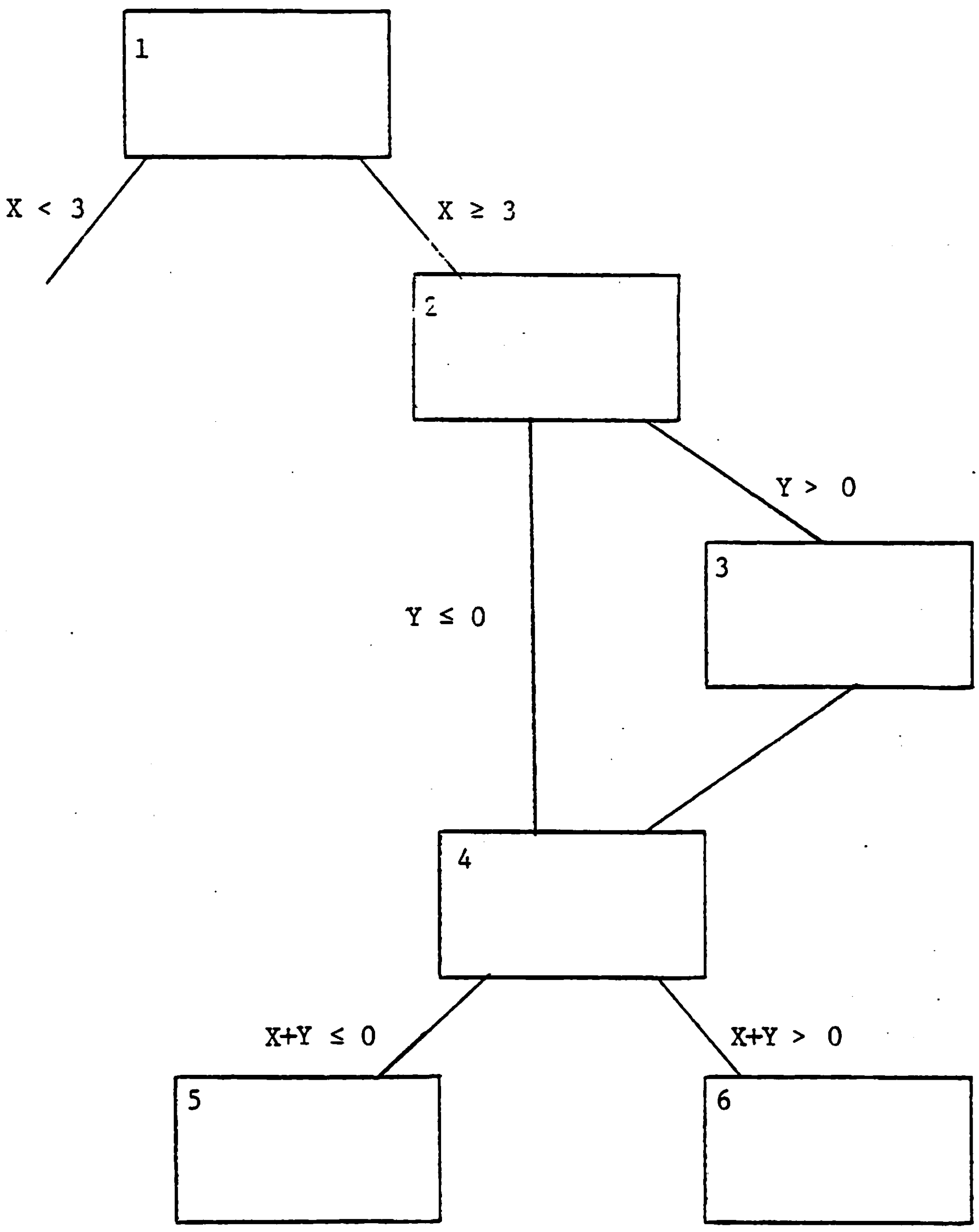


Figure 4

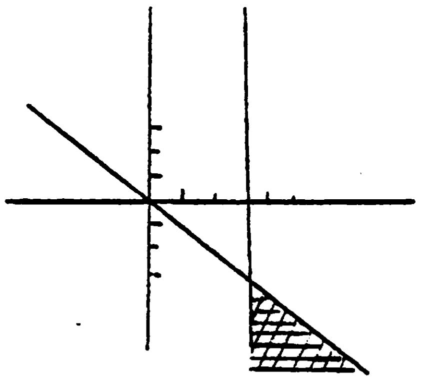
Figure 5

<u>Path 1,2,4,5</u>

P_1 $x \geq 3$

P_2 $y \leq 0$

P_3 $x + y \leq 0$

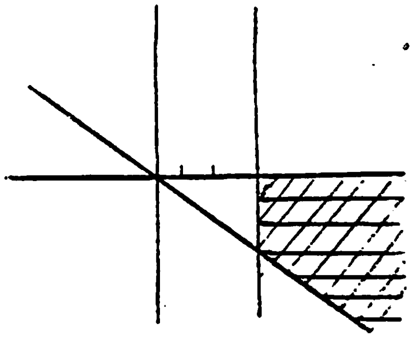


<u>Path 1,2,4,6</u>

P_1 $x \geq 3$

P_2 $y \leq 0$

P_3 $x + y \geq 0$



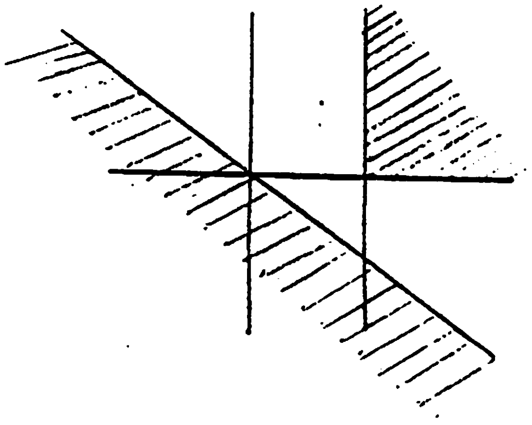
Path 1,2,3,4,5

$x \geq 3$

$y > 0$

$x + y \leq 0$

(nonexecutable)

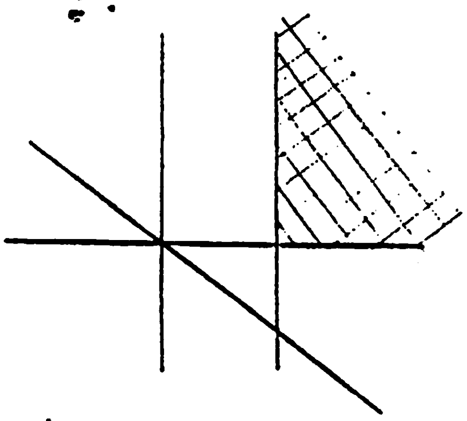


Path 1,2,3,4,6

$x \geq 3$

$y > 0$

$x + y > 0$



have been tried include manual dynamic selection, manual static selection, automatic static selection, and automatic total selection. The problems with each of these methods will be explained and ATTEST's dynamic method of selecting a subset of paths to satisfy a user-selected testing criterion will be described.

Manual dynamic path selection requires the user of the system to select the next statement whenever a decision point is encountered. This technique is beneficial when symbolic execution is being used as a debugging tool or when the user is interested in a particular path. However, there are two drawbacks to using this technique as a general testing strategy. First, it is very tedious and not the most efficient use of personnel. Second, it is surprisingly difficult to manually direct the system to untested areas of the program. Programs tend to contain a considerable number of nonexecutable paths that, in our experience, are deceivingly difficult to avoid. This is particularly true when the user of the system did not write the program being tested. ATTEST supports an interactive path selection capability but it is only recommended for debugging.

Manual static path selection requires the user to completely specify a program path before analysis is initiated. As mentioned in the previous case, users tend to inadvertently select a large proportion of nonexecutable paths.

In automatic static selection, paths are automatically selected prior to symbolic execution. This method is usually based on the graph structure of the program. However, without additional semantic information, this has the same drawbacks as the previous two methods. There has been some work on statically detecting inconsistent pairs of predicates [11,12] which reduces the total number of control paths that need to be considered. However, static algorithms for selecting paths that exclude these incompatible predicates have been shown to be intractable [13].

The last alternative has the disadvantage of inundating the user with paths. Even when an upper bound is imposed on the number of loop iterations, the number of paths is usually unmanageable. Figure 6 shows a loop with one conditional branch. With the upper bound on the number of loop iterations set to N, the number of possible paths is:

$$\sum_{j=1}^{N} 2^j = 2^{N+1}-2.$$

Even when N is small, the number of paths explodes and usually generates more information than the user can examine. In addition, many of the paths will be uninteresting, differing only in the number of loop iterations.

ATTEST has chosen a more pragmatic approach to testing that is, by necessity, less comprehensive than total path analysis. ATTEST's automatic path selection component chooses a subset of a program's paths according to several testing criteria. Each one of the testing criterion will be discussed and the methods employed to satisfy each criterion will be briefly explained.

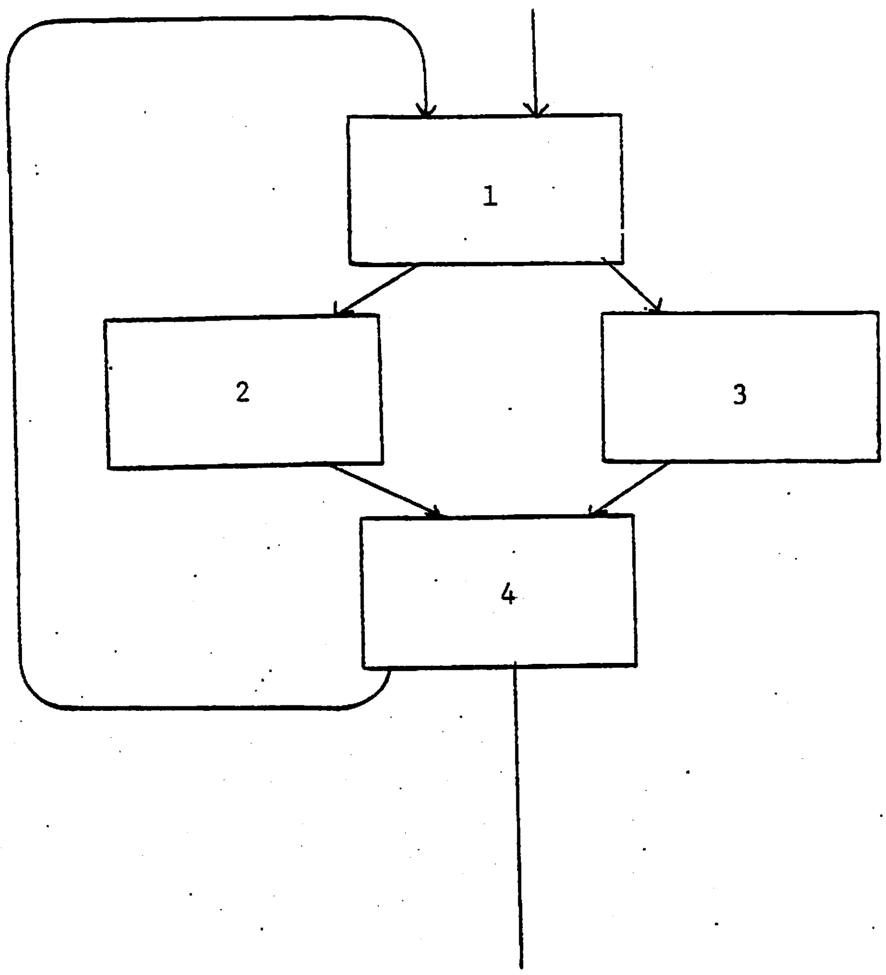


Figure 6

ATTEST's testing criteria includes recognizing two types of structural subcases: loop boundary conditions and language dependent conditions; and three methods of path selection: statement coverage, branch coverage, and total path coverage.

In order to exercise loop boundary conditions, ATTEST attempts to create path descriptions that will execute the program's loops a minimum and maximum number of times. The current implementation does this only for FORTRAN DO loops. The linear programming algorithm from the test data generation component is used to find the minimum and maximum value of the symbolic representation of the final loop iteration variable subject to the constraints imposed by the path predicates. When no upper bound exists, a user selected default value is used. A DO loop in FORTRAN has a lower bound of at least one.

The language dependent subcases that ATTEST automatically analyzes include array index out of bounds, division by zero, computed GO TO index out of bounds, and variable dimension out of bounds. Other subcases such as overflow and underflow could also be included. All language dependent subcases are handled in a similar manner. When the subcase is encountered during symbolic execution, a predicate P_i is formed which represents the error condition for the construct. The test data generation component then determines the consistency of P_i with the existing set of predicates. If the predicate is inconsistent, no test data exists that would cause this subcase to occur on the chosen path. If the predicate is consistent, then a data set is generated that would cause the error. Since all the special cases recognized by ATTEST would result in a run time error, a warning message is also returned to the user. The predicate P_i is subsequently removed from the set of path predicates. Figure 7 shows the predicates that are created for each of the language dependent special cases recognized by ATTEST. Note, most of the constructs have an upper and lower bound that must be checked separately.

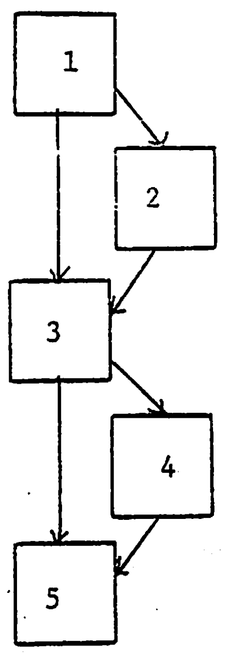
The remaining criteria are concerned with selecting program paths. The total path selection criterion is provided for those applications requiring extensive testing. A simple path enumeration method is used but each path description is generated dynamically. When an inconsistent predicate is selected and then rejected, an alternate branch may be chosen. Except for a few conditional constructs in FORTRAN (computed GOTO and assigned GOTO), at least one of the branches emanating from a statement should result in a consistent predicate.

Statement coverage and branch coverage are the criteria for selecting a subset of the program paths. A user can select either criterion or combine the two. Figure 8 demonstrates the difference between the three path coverage alternatives for a simple program graph.

Statement coverage and branch coverage are implemented in a similar manner so only the former will be described here. The system initially creates the program's transitivity matrix. The transitivity matrix shows which statements are potentially reachable from any other statement in the program. The matrix is formed from a purely structural analysis of the program graph and does not take into consideration program semantics. The system also maintains a vector of all statements that have not been executed. This vector and the transitivity matrix can be

FORTRAN STATEMENT	PREDICATE
X = Y/Z	S(Z) = 0
X = A(I)	S(I) < 1
	S(I) > dimension of A
GOTO(10,20,30),I	S(I) < 1
	S(I) > 3
DIMENSION A(N)	S(N) < 1
	S(N) > maximum dimension of A

Figure 7



all statements:	1,2,3,4,5
all edges:	1,2,3,4,5
	1,3,5
all paths:	1,2,3,4,5
	1,3,5
	1,2,3,5
	1,3,4,5

Figure 8

compared to determine the potential number of new statements that may be reached from any statement in the program.

An estimated desirability value is computed for every feasible branch at a decision point. A feasible branch is consistent with the existing set of path predicates and emanates from the current statement on the path. If the testing criteria includes executing all statements, then the potential number of new statements would affect the estimated desirability value. At each decision point, the path selection component selects the feasible branch with the maximum desirability value. Using Figure 5, assume statements 1, 2, and 3 have been symbolically executed and both branches (3,4) and (3,5) are consistent with the one existing predicate. Branch (3,4) would be chosen next since it has the potential of reaching one new statement (5).

The desirability value depends on the selected testing criteria and on information that has accumulated from previous path selection attempts. For example, the number of times a branch has already been selected on previous paths is taken into account in order to increase the variety of subpaths within a set of path descriptions. A complete description of the path selection component is in [14].

VII. Conclusion

There are currently few systems to aid in the selection of test data though testing is known to be a labor intensive and tedious task. The ATTEST system attempts to generate data sets to assist in structural program testing. The current status of ATTEST and some of the areas in need of further investigation are mentioned below.

The symbolic execution component handles most FORTRAN constructs though there are difficulties with arrays and file manipulations. A method of including I/O specifications is described in [15] and is partially implemented. The test data generation component is restricted to systems of linear predicates but our experimental results show that most predicates can be simplified or transformed to linear expressions. Some of these transformations are already implemented in ATTEST. Other methods of solving systems of inequalities are also being examined.

The path selection component has just recently been implemented. For all test programs attempted to date, at least eighty percent of the requested path coverage was achieved in a reasonable number of attempts. Since path selection is striving for the best gain and not one particular path, it is usually efficient. However, the method does deteriorate when only a small number of untested statements remain. Other testing criteria are being explored as well as methods for designating critical areas of the code that should receive additional consideration. Methods of incorporating the work on incompatible predicates are also being investigated.

Appendix

A sample run using ATTEST is shown below. The testing criteria used for this run included loop boundary conditions, language dependent conditions, statement coverage, and branch coverage. The predicates generated for the paths are all fairly simple and do not demonstrate the full capabilities of the test data generation component. However, the capabilities of the path selection component are demonstrated.

```
Block                          Source

  1          INTEGER MRCHTB(5,2),TBSIZ,DEPT,INVNUM,MIN,MAX
  1          DATA TBSIZ/5/
  0  C
  0  C  INVENTORY NUMBERS
  0  C
  1          DATA MRCHTB(1,1)/56965/,MRCHTB(2,1)/31415/,MRCHTB(3,1)/14736/,
  1         C     MRCHTB(4,1)/12345/,MRCHTB(5,1)/91789/
  0  C
  0  C  MIN AND MAX INVENTORY NUMBERS
  0  C
  1          DATA MIN/11111/,MAX/99999/
  0  C
  0  C  DEPARTMENTS HANDLING INVENTORY
  0  C
  1          DATA MRCHTB(1,2)/3/,MRCHTB(2,2)/1/,MRCHTB(3,2)/3/,
  1         C     MRCHTB(4,2)/2/,MRCHTB(5,2)/1/
  0  C
  0  C
  2          READ(5,100) INVNUM
  3          CALL LINSCH(INVNUM,MRCHTB,TBSIZ,MIN,MAX,DEPT)
  4          WRITE(6,100)INVNUM,DEPT
  5          STOP
  0  C
  0   100    FORMAT(I5)
  1          END
```

```
Block                          Source

  1           SUBROUTINE LINSCH(INVNUM,MRCHTB,TBSIZ,MIN,MAX,DEPT)
  1           INTEGER INVNUM,TBSIZ,MRCHTB(TBSIZ,2),MIN,MAX,DEPT
  0  C
  0  C
  2           IF(INVNUM.LT.MIN)
  3          $GOTO 900
  4           IF(INVNUM.GT.MAX)
  5          $GOTO 900
  0  C
  6           DO 10 INDEX=1,TBSIZ
  7             IF(MRCHTB(INDEX,1).EQ.INVNUM)
  8          $GOTO 20
  9   10      CONTINUE
 10           DEPT=0
 11           RETURN
  0  C
 12   20      DEPT=MRCHTB(INDEX,2)
 13           RETURN
  0  C
  0  C   ERROR RETURN
  0  C
 14   900     DEPT=-1
 15           RETURN
  1           END
```

Path 1:

Main Procedure: 1,2,3, Procedure LINSCH: 1,2,4,6,7,9,7,9,7,9,7,9, 7.9,10,11,Main Procedure: 4,5.

Predicates:

$\neg$(INVNUM < 11111)

$\neg$(INVNUM > 99999)

$\neg$(56965 = INVNUM)

$\neg$(31415 = INVNUM)

$\neg$(14736 = INVNUM)

$\neg$(12345 = INVNUM)

$\neg$(91789 = INVNUM)

Test Data:

INVNUM = 11111

Path Output:

DEPT = 0

Path 2:

Main Procedure: 1,2,3, Procedure LINSCH: 1,2,3,14,15, Main Procedure: 3,5

Predicates:

INVNUM < 11111

Test Data:

INVNUM = 0

Path Output:

DEPT = 0

Path 3:

Main Procedure: 1,2,3, Procedure LINSCH: 1,2,4,5,14,15, Main Procedure: 3,5

Predicates:

$\neg$(INVNUM < 11111)

(INVNUM > 99999)

Test Data:

INVNUM = 100000

Path Output:

DEPT = -1

Path 4:

Main Procedure: 1,2,3, Procedure LINSCH: 1,2,4,6,7,8,12,13, Main Procedure: 4,5

Predicates:

$\neg$(INVNUM < 11111)

$\neg$(INVNUM > 99999)

(56965 = INVNUM)

Test Data:

INVNUM = 56965

Path Output:

DEPT = 3

Path 5:

Main Procedure: 1,2,3, Procedure LINSCH: 1,2,4,6,7,9,7,9,7,9,7,9,7,8,12,13, Main Procedure: 4,5

Predicates:

$\neg$(INVNUM < 11111)

$\neg$(INVNUM > 99999)

$\neg$(56965 = INVNUM)

$\neg$(31415 = INVNUM)

$\neg$(14736 = INVNUM)

$\neg$(12345 = INVNUM)

(91789 = INVNUM)

Test Data:

INVNUM = 91789

Path Output:

DEPT = 1

References

[1] L.G. Stucki, "Automatic Generation of Self-Metric Software," Rec. 1973 IEEE Symposium Computer Software Reliability.

[2] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Trans. Software Engineering, June 1975.

[3] B.H. Liskov and V. Berzins, "An Appraisal of Program Specifications," Proceedings of the Conf. on Research Directions in Software Technology, October 1977.

[4] W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Trans. Software Engineering, July 1977.

[5] B. Elspas, K.N. Levitt, R.J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, Vol. 4, No. 2, June 1972.

[6] A.H. Land and S. Powell, FORTRAN Codes for Mathematical Programming, John Wiley & Sons, 1973.

[7] E.I. Cohen and L.J. White, "A Finite Domain-Testing Strategy for Computer Program Testing," Ohio State University Technical Report OSU-CISRC-TR-77-13, August 1977.

[8] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. Software Engineering, September 1976.

[9] R.S. Boyer, B. Elspas, and K.N. Levitt, "Select - A Formal System for Testing and Debugging Programs by Symbolic Execution," Proceedings 1975 Int. Conf. Reliable Software, IEEE Computer Society.

[10] J.C. King, "Symbolic Execution and Program Testing," CACM, Vol. 19, No. 7, July 1976.

[11] K.W. Krause, R.W. Smith and M.A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis," Rec. 1973 IEEE Symposium Computer Software Reliability.

[12] L.J. Osterweil, "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis," University of Colorado Technical Report CU-CS-110-77, May 1977.

[13] H.N. Gabow, S.N. Maheshwari and L.J. Osterweil, "On Two Problems in the Generation of Program Test Paths," IEEE Trans. Software Engineering, September 1976.

[14] J.L. Woods, "Path Selection for Symbolic Execution Systems," Ph.D. Thesis, University of Massachusetts, 1978.

[15] P. Abrahams and L.A. Clarke, "Compile-Time Analysis of Data List-Format List Correspondences," University of Massachusetts Technical Report #78-11, Computer and Information Science Department.