Methodology for the Generation of Program Test Data

WILLIAM E. HOWDEN

***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 either generate a test case that causes some path in that class to be followed or determine that no such data exist. 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.

***Index Terms*—Flowchart analysis, inequality solution techniques, program paths, program testing, systems of predicates.**

I. INTRODUCTION

THE VALIDATION phase of the software production process has received increasing attention in the last few years (e.g., [1]-[3]). In the program verification approach to validation 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 *et al.* [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. Krause *et al.* [6] describe a system for extracting a sequence of paths from a program which "covers" each branch in the program. A scheme has been devised by Paige and Balkovitch [7] for testing a program against its specifications. Ramamoorthy *et al.* [8] have constructed a system which checks a program for anomalous statements and constructions.

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 methodology decomposes a program into a finite number of standard classes of program paths. A class of paths is "feasible" if input data exist which cause some path in the class to be followed. The methodology attempts to determine which classes of paths are feasible, and to generate a set of test cases which cause one path from each feasible class to be followed. The general test data problem is equivalent to the halting problem. An algorithm which could determine the feasibility of any class of paths would be capable of determining whether data existed which would cause an arbitrary program to halt. A complete standard set of test cases contains one test for each feasible class of paths. The methodology attempts to generate a large subset of a complete set.

II. GENERAL APPROACH

The methodology can be described as consisting of five phases. The first phase analyzes a program and constructs descriptions of standard classes of paths through the program. The second phase 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 consist of the assignments, loops, function calls, and branch predicates which characterize the data causing the paths to be followed. 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.

Manuscript received July 22, 1974; revised November 12, 1974. The research described in this paper was carried out as part of the McDonnell Douglas Astronautics program in the research and development of program validation software and was funded by the National Bureau of Standards.

The author is with the Computer Science Division, Department of Applied Physics and Information Science, University of California, La Jolla, Calif. 92037.

III. GENERATION OF CLASS DESCRIPTIONS

A. Boundary-Interior Test Paths

There are a potentially infinite number of paths through a program which contains loops. There are several methods for choosing a finite number of these for testing. Little theoretical or empirical work has been carried out to justify the use of one of these methods in preference to the others. The phase one process, which is described in this section, uses the boundary-interior method for choosing test paths. The basic idea of the method is to group the paths through a program into a finite set of classes of paths in such a way that a test of one path from each class constitutes an intuitively complete set of tests.

In the boundary-interior approach to testing, it is assumed that a complete set of tests must test alternative paths through the top level of a program, alternative paths through loops, and alternative boundary tests of loops. 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 separates paths into separate classes if they differ other than in traversals of loops. If two paths P_1 and P_2 are the same except in traversals of loops they are placed in separate classes if

(1) one is a boundary and the other an interior test of a loop;

(2) they enter or leave a loop along different loop entrance or loop exit branches;

(3) they are boundary tests of a loop and follow different paths through the loop;

(4) they are interior tests of a loop and follow different paths through the loop on their first iteration of the loop.

B. Class Descriptions

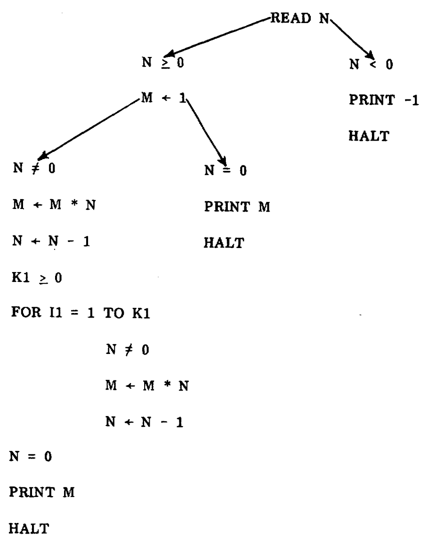
The first phase of the methodology constructs program-like descriptions of classes of program paths. Class descriptions consist of branch predicates, assignment statements, I/O statements, and "FOR-loops." The FOR-loop notation is used to represent an arbitrary number of traversals of a loop. The use of a FOR-loop may involve the introduction of dummy variables into a program. The complete set of class descriptions for a program can be represented in the form of a description tree. Fig. 2 contains the boundary-interior class description tree for the program in Fig. 1. The leftmost path in the tree describes the class of all paths which test the interior of the loop in the program. The other paths are boundary tests.

C. Class-Description Generation Process

Phase one constructs the class descriptions for a program by traversing its description tree. The structure of the phase one process can be described as a recursive

```
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.



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

Fig. 2. Description tree.

finite state automaton. The automaton described in Fig. 3 can be used for constructing the boundary-interior class descriptions for programs in which all loops are properly nested.

The automaton begins in the MAIN state. MAIN is used for processing the main or top level paths through a program. When a subloop is encountered the automaton enters the ENTRANCE state. ENTRANCE reads through a loop, searching for the loop's first statement. PATH follows alternative paths through a loop. ITERATE constructs the FOR-loop constructs which denote an arbitrary number of traversals of a loop. The EXIT state exits from a loop. It reads down paths from the first statement of a loop to some exit branch out of the loop. Each of the states PATH, ENTRANCE, and EXIT causes a recursion to take place if it encounters a subloop during its processing. When a recursion takes place the new automaton for processing the subloop is entered at the ENTRANCE state. BRANCH is invoked when a conditional branching statement in the program is encountered. It initiates the processing of the program down each of the branches. For branches which leave the loop currently being proc-

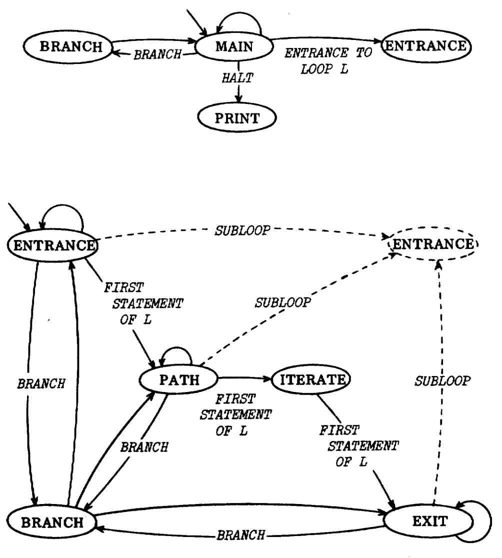


Fig. 3. State diagram for class description generation process.

essed, this will involve the return of control to some previously interrupted automaton. Each of the automaton states adds description statements on to the current class description as it processes program source statements.

IV. IMPLICIT INPUT DATA DESCRIPTIONS

The predicates in a path, together with the input and computational statements which 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 two of the methodology constructs implicit input data descriptions of the sets of data which cause classes of paths to be followed. Fig. 4 contains the implicit input data description for the interior test class description in Fig. 2.

Phase two can be described in two parts. The first part deletes the output statements from a class description and replaces input statements by assignment statements. Phase two will be incomplete for certain classes of input statements which cannot be easily replaced by assignment statements. Values read by a simple READ statement not involving arrays can be represented by the dummy input variables #1, #2,$\cdots$. In Fig. 4 the input statement "READ N" has been replaced by the assignment $N \leftarrow \#1$.

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 a list of "predicate affecting" variables. It uses these lists to determine which assignment statements do not affect predicates and can be deleted from a description.

V. TRANSFORMING IMPLICIT INTO EXPLICIT DESCRIPTIONS

A. *Partially Explicit Input Data Descriptions*

Phase three of the methodology is a symbolic interpretation process which attempts to evaluate and delete the assignment statements and FOR-loops in an implicit

```
N ← #1

N ≥ 0

N ≠ 0

N ← N - 1

K1 ≥ 0

FOR I1 = 1 TO K1

          N ≠ 0

          N ← N - 1

N = 0
```

Fig. 4. Implicit input data description.

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. The process is similar to verification condition generation for proving correctness.

It is not always possible to delete all of the assignment statements from an implicit description. Phase three usually results in the generation of a "partially explicit description." A completely explicit class description consists of a system of inequalities in input variables and constants. The generation of completely explicit descriptions is prevented by the presence of array references and loops in implicit descriptions. Array references, for example, have the property that they may stand for different array elements depending on the values of the indices in the references. If 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 statements containing the reference.

There are four classes of FOR-loop interpretation problems. The first three involve problems in interpreting statements inside FOR-loops. 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.

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.

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. 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 $(P \wedge X \leftarrow Y) \vee \backsim P$. 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 it 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. 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 fourth class of problems involves the construction of closed forms for loops and for iteratively computed predicates inside loops.

Suppose that phase three were unable to find a closed form for the loop in the implicit description in Fig. 4. Then it would generate the partially explicit description in Fig. 5. The assignment $N \leftarrow \#1$ has been evaluated and deleted 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. The 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. The assignment $N \leftarrow \#1 - 1$ must be retained to denote the value of N on entry to the FOR-loop.

The loop in Fig. 4 can be replaced by the predicate $(N < 0 \vee N > K1 - 1)$ and the assignment $N \leftarrow N - K1$. If phase three were able to find this closed form it could carry out the substitution of a value for N into the closed form expression and delete the remaining assignment statements. The result would be the completely explicit description in Fig. 6.

B. Interpretation Process

The phase three interpretation process consists of two parts. In the first part assignment statements are evaluated and an attempt is made to construct closed forms for loops. Values of variables are substituted into predicates and relations. In the second part 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 symbolic interpretation process can be carried out in several passes. It can be designed as a recursive, multipass procedure which processes loops. Each time a subloop is encountered during a pass over the loop currently being processed the evaluation procedure is recursively reinitiated. In the first pass over a loop the procedure constructs a

```
#1 ≥ 0

#1 ≠ 0

N ← #1 - 1

K1 ≥ 0

FOR I1 = 1 TO K1

        N ≠ 0

        N ← N - 1

N = 0
```

Fig. 5. Partially explicit description.

```
#1 ≥ 0

#1 ≠ 0

K1 ≥ 0

(#1 - 1 < 0 ∨ #1 - 1 > K1 - 1)

#1 - 1 - K1 = 0
```

Fig. 6. Explicit description.

symbol table of the variables which receive values in the loop. In the second pass it evaluates assignment statements and substitutes values into predicates. The evaluator uses the symbol table for a loop to determine when a value of a variable can be "brought into" a loop. The value of a variable cannot, in general, be brought into a loop if the variable is assigned a value in the loop.

The evaluation procedure carries out a sequence of symbol table and evaluation passes over a loop until no further evaluation is possible. Two symbol/evaluation passes appear to be sufficient for Fortran partially explicit descriptions. After the completion of the symbol table and evaluation passes, the evaluation procedure tries to find a closed form for the loop. It attempts to find closed forms for the iteratively computed values and predicates in the loop. When the closed-form pass over a loop has been completed control is returned to the interrupted pass of the next higher level call of the evaluation procedure.

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.

VI. EXPLICIT SUBSET 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 upper bound is a variable $K \geq 0$. D describes the set of all input data which satisfy 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.

Fig. 6 contains a closed form for the FOR-loop in the example in Fig. 5. Suppose that the evaluator had been unable to construct the closed form for the FOR-loop. Each choice of a nonnegative integer value for $K1$ corresponds to a different subset of the set described by the partially explicit description. Fig. 7 contains the subset description corresponding to the choice $K1 = 0$. The description in Fig. 7 contains no FOR-loops and can be evaluated to produce the explicit subset description in Fig. 8.

A description is *feasible* if there are values in the input domain which satisfy the description. Infeasible descriptions describe the empty subset of the input domain. 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. The subdescription constraining the loop bound in Fig. 5 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.

```
#1 ≥ 0

#1 ≠ 0

N ← #1 - 1

N = 0
```

Fig. 7. Partially explicit subset description.

```
#1 ≥ 0

#1 ≠ 0

#1 - 1 = 0
```

Fig. 8. Explicit subset description.

VII. 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 is a wide range of difficulty in solving systems of inequalities. Some solution techniques 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. Several of the possible solution methods which can be included in phase five are described below. Without extensive empirical investigation it is difficult to estimate how effective the techniques will be and in what percentage of cases they will be applicable.

Linear real-valued systems which do not contain functions and array references with variable indices can be solved using several different methods. Kuhn's method [9] is a straightforward effective method. Linear programming can also be used although it involves the extra computational effort of finding minimal solutions. An effective method developed by Singhania and described in [10] can be used to produce solutions for nonlinear systems in one variable of degree less than five. Experience indicates that a large number of the descriptions generated by phases three and four can be solved using these and other effective techniques. The explicit subset description in Fig. 8 can be easily solved using a method for linear systems in one variable.

Different partially effective methods can be used for solving general nonlinear and integer-valued systems. In the Kuhn's backtrack search method dummy real-valued linear variables are substituted for nonlinear or integer variables. Kuhn's method is then used to construct a sequence of linear partial systems in which each system has one or more fewer variables than its predecessor. The process solves each of the partial systems in turn, starting with the smallest system. At each stage it checks to see if the partial solution satisfies the constraints of the original system. If not it constructs a new partial solution. If no new partial solution can be constructed it backtracks and constructs a new solution to the previous partial system. The method is successful if it halts with a complete solution that satisfies the original system.

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 dealing with 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. In other cases a variation of the backtrack search method can be applied.

VIII. CONCLUSIONS

The methodology which is described in the preceding sections can be used to generate data for certain classes of 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.

Many of the basic problems of test data generation were discovered during the design of the methodology. Several of these problems are mentioned in the preceding sections. The boundary-interior method used in phase one generates very large sets of classes of paths for large or complex programs. Strategies must be used for choosing a subset of this class or for generating smaller sets. The recursive algorithm for generating boundary-interior class descriptions requires that programs have properly nested loop structures. The phase two procedures for replacing input statements by assignments will not work for certain kinds of input statements. If these kinds of input statements are not avoided then class descriptions will be generated which cannot be prepared for interpretation by the phase three symbolic interpreter. Complex interdependent, nonintersecting program loops will result in the generation of infeasible subset descriptions by phase four of the methodology. The phase five inequality solution routines will only be able to deal with simpler systems of inequalities. Generation of test data cannot be guaranteed for classes of paths containing branch predicates that involve subroutines, functions, or nonlinear expressions in several variables. Phases one and two of the methodology have been implemented. These phases can be used to print out implicit descriptions of the input data which cause standard classes of paths through a program to be followed. Experience with this system will provide empirical data on the complexity of the more commonly occurring classes of paths and their corresponding input data descriptions.

The first four phases of the methodology could be used as part of a partial program correctness system. The systems of inequalities generated by the first four phases can be used to construct verification conditions for particular paths through a program. 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.

ACKNOWLEDGMENT

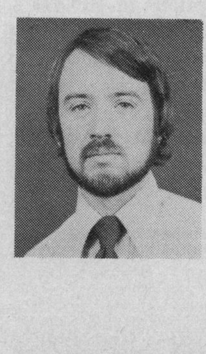
Several students at the University of California, Irvine, participated in the research which is described in this paper. R. Singhania carried out a survey of inequality solution techniques. D. Mies patiently and carefully compiled statistics on the structure of Fortran programs. S. Williams experimented with the boundary-interior classification scheme.

The author would like to acknowledge the many helpful comments of R. Stillman and S. 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 and was carried out under the supervision of Z. Jelinski and L. Stucki of the Information Sciences Division of McDonnell Douglas. The author wishes to express his gratitude to L. Stucki and Z. 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," *Comput. Surv.*, vol. 4, pp. 97–147, June 1972.
[2] W. C. Hetzel, Ed., *Program Test Methods*. New York: Prentice-Hall, 1972.
[3] *Proc. 1973 IEEE Symp. Computer Software Reliability*, New York, N. Y.
[4] L. G. Stucki, "Automatic generation of self-metric software," in *Proc. 1973 IEEE Symp. Computer Software Reliability*, 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 *Proc. 1973 IEEE Symp. Computer Software Reliability*, 1973, pp. 18–22.
[7] M. R. Paige and E. E. Bolkovitch, "On testing programs," in *Proc. 1973 IEEE Symp. Computer Software Reliability*, 1973, pp. 23–27.
[8] C. V. Ramamoorthy, R. J. Meeker, and J. Turner, "Design and construction of an automated software evaluation system," in *Proc. 1973 IEEE Symp. Computer Software Reliability*, 1973, pp. 28–37.
[9] H. W. Kuhn, "Solvability and consistency for linear equations and inequalities," *Amer. Math. Mon.*, vol. 63, pp. 27–38, Apr. 1956.
[10] W. E. Howden, L. G. Stucki, and Z. Jelinski, "Final report: Methodology for the effective test case selection, Part I," McDonnell Douglas Astronautics Rep. MDC G5301, Jan. 1974.



William E. Howden was born in Vancouver, B.C., 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, N. J., 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, Calif., 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.

Correspondence

A Method for Obtaining SPOOF's

SIU-CHONG SI AND ALFRED K. SUSSKIND

Abstract—**A compatibility relationship on network paths is defined in such a way that the maximal compatibles are isomorphic to the products in the "structure and parity-observing output function" (SPOOF), a subscripted Boolean expression for the network output that uniquely specifies the network structure. For a given network, the path compatibility relations are easy to find, as are the maximal compatibles and hence the SPOOF for the network output. Because the collection of all the path compatibility relations, conveniently displayed in matrix form, completely characterizes the network, the compatibility matrix can be used for a variety of purposes.**

Index Terms—**Compatibility, maximum compatibles, logic-network representation, SPOOF, sum-of-products form, diagnostics, fault testing.**

Several recent publications [1]–[3] have shown the usefulness of modified Boolean expressions in the study of diagnostics for logic networks. We will use here the particular form described in [1] and called the structure and parity-observing output function (SPOOF). The authors of [1] and [2] have shown how SPOOF's reveal the effects of faults on network behavior, how they can be used to explore fault-equivalence classes and to analyze test effectiveness and efficiency, and how they allow understanding of the logical-structural relationship created by redundancy. In [3] the authors have shown how a complete fault-location test set for single stuck-at faults can be obtained efficiently, i.e., by a nonenumerative procedure, through the use of SPOOF's. The authors have also shown how tests for "logical" short-circuit faults (i.e., unintended connections between signal leads that behave like a wired AND or OR) or "regional faults" (i.e., faults that alter the function of some part of the network in an arbitrary but specified manner) can be derived from the associated SPOOF. In short, tests for any fault that can be modeled in Boolean algebra can be obtained by manipulations of SPOOF's. When the subscripts in a SPOOF are removed, there results what has been called in [4] the *E*-expression which, as shown there and in [5], can be used for efficient generation of *multiple* stuck-at fault-detection tests.

As yet, the use of SPOOF's or *E*-expressions is not widespread, partly because test-generation algorithms based on them are relatively new. In addition, it is laborious to find the SPOOF for a network of even moderate size (say, 100 gates, 10 levels). Some believe that it is too costly to obtain the SPOOF of a moderately complex network.

We present here a novel technique for generating SPOOF's, expected to lead to substantial reduction in computational complexity. It may also be an efficient technique for finding *E*-expressions. Moreover, the "compatibility matrix," introduced below, is a novel means of network representation that has applications outside of diagnostics.

Our starting point is the typical data base in a design automation system: the interconnections between gates, the gate types, and the labeling of input and output leads.

The new approach is based on a compatibility relationship between network paths which we define in such a way that the maximal compatibles are isomorphic to the products in the SPOOF. The determination of the compatibility relationships for a given network is shown to be easy, and, due to the work of others, finding all the maximal compatibles is also straightforward. Furthermore, the particular nested structure in which we display the compatibility relations simplifies the task of determining the maximal compatibles. This nested structure is also beneficial in storage management when our approach is implemented in a computer program.

I. REVIEW OF SPOOF TERMINOLOGY

Whereas the elements of a conventional Boolean expression are *literals*—input variables and their complements—the elements of a SPOOF are "terms." A term is a literal together with a path list, which denotes a path through which the literal reaches the output. Path-list entries are lead labels, some of which may be complemented.

SPOOF terms may be manipulated according to all of the rules of Boolean algebra. However, terms having *different* path lists are to be considered *distinct* algebraic variables. The meaning of this rule is that if J and K are distinct path lists, then all of the following are algebraically irreducible: $x_J x_K$, $x_J + x_K$, $x_J \bar{x}_K$, and $x_J + \bar{x}_K$. As a direct consequence of this rule, SPOOF's are topology preserving, i.e., they reveal the network function as well as the network structure.

Fig. 1 is a gate configuration that realizes the function $f = w\bar{x} + \bar{x}\bar{y}\bar{z}$. The leads have been labeled according to the scheme given in [3]. Following the procedure in [1], one obtains for the SPOOF's of leads 7 and 8, respectively,

$$S_7 = w_{1,7}\bar{x}_{\bar{2},\overline{2a},6,7}x_{2,2b,5,5a,7} + w_{1,7}\bar{x}_{\bar{2},\overline{2a},6,7}y_{3,5,5a,7} + w_{1,7}\bar{x}_{\bar{2},\overline{2a},6,7}z_{4,5,5a,7}$$

$$S_8 = \bar{x}_{\bar{2},\overline{2b},\bar{5},\overline{5b},8}\bar{y}_{\bar{3},\bar{5},\overline{5b},8}\bar{z}_{\bar{4},\bar{5},\overline{5b},8}.$$

The SPOOF for the network output is the sum of the terms in S_7 and S_8, with lead 9 appended to every path list.

Manuscript received July 12, 1974; revised November 14, 1974.

The authors are with the Department of Electrical Engineering, Lehigh University, Bethlehem, Pa. 18015.