SELECT--A FORMAL SYSTEM FOR TESTING AND DEBUGGING PROGRAMS BY SYMBOLIC EXECUTION*

Robert S. Boyer
Bernard Elspas
Karl N. Levitt
Computer Science Group
Stanford Research Institute
Menlo Park, California 94025

Keywords: Program Testing; Test Data Generation; Symbolic Execution; Program Verification; Program Debugging; Solution of Systems of Inequalities.

Abstract

SELECT is an experimental system for assisting in the formal systematic debugging of programs. It is intended to be a compromise between an automated program proving system and the current ad hoc debugging practice, and is similar to a system being developed by King et al. of IBM. SELECT systematically handles the paths of programs written in a LISP subset that includes arrays. For each execution path SELECT returns simplified conditions on input variables that cause the path to be executed, and simplified symbolic values for program variables at the path output. For conditions which form a system of linear equalities and inequalities SELECT will return input variable values that can serve as sample test data. The user can insert constraint conditions, at any point in the program including the output, in the form of symbolically executable assertions. These conditions can induce the system to select test data in user-specified regions. SELECT can also determine if the path is correct with respect to an output assertion. We present four examples demonstrating the various modes of system operation and their effectiveness in finding bugs. In some examples, SELECT was successful in automatically finding useful test data. In others, user interaction was required in the form of output assertions. SELECT appears to be a useful tool for rapidly revealing program errors, but for the future there is a need to expand its expressive and deductive power.

I. Introduction

This paper discusses an experimental system, SELECT, for aiding a user in systematically debugging programs. It differs from conventional debuggers in that it attempts to carry out a formal processing of the program paths under user interaction in order to (1) generate potentially useful test data, (2) provide simplified symbolic values for program variables at the output of a path and (3) prove the correctness of a path with respect to user-supplied assertions. The path also attempts to give preliminary evaluations on the potential utility of formal testing tools. In particular, we show that the automatic generation of test data can be useful but certainly should not be viewed as a total solution to the problem. A more promising method is to incorporate user interaction in the test generation process. SELECT achieves this to a degree by permitting the user to insert assertions at points within the program that serve to constrain the test data to desired regions.

A. Mechanical program verification and its shortcomings

Much attention has been paid to the present high cost of producing reliable software, and to the frequency of latent bugs even after a great deal of effort has been expended in testing and debugging. Among the tools that have been proposed and studied to alleviate this situation are a variety of techniques for formally verifying the mathematical correctness of programs. All of these formal program verification techniques, which have been embodied in a number of experimental program verifiers, make use of formal specifications of the program's intent, written in a formal assertion language. The verification process then consists in analyzing the actions carried out by the program, and checking (usually by proving mathematical theorems) that the output assertions will be satisfied whenever the input data meets the conditions specified by the input assertions. The details of the analysis process, and more particularly, of the checking process differ from system to system, but all of the present experimental verification systems operate in the general way we have described.

It is generally conceded, even by proponents of particular verification systems or of as yet unimplemented techniques, that they are a long way off from being able to handle practical programs of substantial size. We shall list some of the reasons why this is so, and then proceed to discuss briefly why even after the present obstacles have been overcome, formal program verification would still fall short of providing the tools needed for practical program debugging. The main present obstacles to formally verifying realistic programs are:

1. The necessity (in most systems) for inventing intermediate assertions (inductive assertions, Floyd predicates, and so forth), and the difficulty experienced by the programmer-verifier in coming up with these assertions.

2. The difficulty in framing input and output assertions that adequately express the programmer's intent.

3. The lack of sufficiently rich assertion languages in which to express these input/output assertions and intermediate assertions for programs covering a wide variety of data types and functional primitives.

*The work reported here was supported by the National Science Foundation under Grant Number GJ-36903X, entitled "The Feasibility of Software Certification."

4. Technical limitations in present-day theorem-proving techniques (inadequate speed and large data storage requirements).

5. An apparent need for considerable programmer intervention at too detailed a level in the theorem-proving process.

Obstacles 1-3 are all concerned with formal assertions and the language in which they are expressed. One may expect that the considerable research efforts now under way will eventually overcome these difficulties in large measure. Moreover, several investigations (Elspas [1974], Wegbreit [1974], Katz-Manna [1973]) now in progress are aimed at simplifying the programmer's task of inventing intermediate assertions by making this a semiautomatic process, with the computer generating the routinely derivable assertions. It is also likely that continued work on mechanical theorem proving (especially if it is focussed on the particular needs of program verification) will make available sufficiently rapid and storage-efficient proof techniques in several years time. However, it is likely that some degree of programmer intervention will always be desirable in this process of verifying program verification conditions.

In addition to the obstacles discussed above, formal program verification suffers from a more basic deficiency which, it would appear, cannot be fully overcome within the framework of what it purports to do. When it works at all, a mechanical (or semimechanical) program verifier can certify correctness only of a program that is actually "correct" (with respect to the stated input-, output-, and intermediate assertions). If, as is usually the case in practice, the program is not correct, the verifier system's output is not particularly useful in helping the user to decide whether:

1. The assertions are "correct," and the program contains a bug, or

2. The program really does what the programmer intended, but the assertions were inappropriately framed, or

3. Both the program and assertions require modification, or

4. Both the program and the assertions are "correct," but the theorem prover contains a bug (or simply was inadequate for proving the required theorems).

To state the matter succinctly, while there is a need for a system for mathematically certifying programs that are correct, there is also a need for one that will rapidly reveal defects in programs that are not correct. In particular, such a system should be able to uncover discrepancies between what a program will actually do and what the programmer intended it to do, even when the programmer may have only a fuzzy idea of what he intended it to do, e.g., by revealing to him that some unintended action will be performed under an unanticipated input data condition. Clearly, such a system must be able to operate without making use of programmer-supplied formal assertions of intent. Thus, the emphasis is on analyzing what the program actually does, rather than on the programmer's intentions. A system of this sort could be useful in constructing programs, by helping to clarify the programmer's initially fuzzy intentions, as well as for ex post facto testing.

Another rather fundamental aspect of formal verification serves as some hindrance to its practicality. Mathematical proofs of program correctness are contingent on a large number of assumptions about the language semantics, the actual compiler implementation, the treatment of error conditions, overflow, underflow, division by zero, and finite machine arithmetic, in the actual implementation on which the program is supposed to run. Thus, even after a mathematical proof of correctness, one cannot be certain that a program will run as intended on a given machine. Testing in the real machine environment on actual data would appear to be a useful complementary technique to formal verification since it is not contingent on such assumptions.

B. The purpose and general features of SELECT

Having been aware for some time of both these present-day, and also the ultimate limitations to formal correctness proving discussed above, a group at Stanford Research Institute (of whom the authors constitute a part) began an investigation around January 1974 as to how these limitations might be circumvented. Our intent was to explore formal methods--not guaranteeing mathematical correctness--that are more likely to reveal bugs in a program, without making use of formal assertions, and that might still produce more confidence in the final product program than contemporary testing techniques can provide. At the date of writing of this paper a partially implemented system, SELECT, has been constructed which exhibits in experimental fashion features of semantic analysis of programs, construction of input data constraints to cover selected program paths, identification of (some) infeasible program paths, automatic determination of actual input data to drive the test program through selected paths, and execution (actual and symbolic) of the test program with optional intermediate assertions and output assertions. These features, which will be described in more detail below, have been provided by making use of the expression-simplifier portion of the SRI Program Verifier, adding thereto facilities for symbolic execution of programs written in a subset of LISP, and numerical programs for determining input data sets satisfying algebraic inequalities (path tracing conditions). It should be noted that our system is similar in some ways to an unimplemented system described independently by Howden [1974], and to a system being constructed by King [1975] at IBM.

In the following sections of the paper we present: a description of the features of SELECT, four examples illustrating the modes of operation, a comparison of SELECT with other formal and near-formal testing methods, and an evaluation of the system including some possible extensions.

II. General Description of the SELECT Testing and Debugging System

A. Overview of the System

The experimental SRI testing and debugging system, SELECT*, has as its main features:

1. Symbolic execution of an input (subject) program, covering (implicitly) all possible execution paths from START to HALT. Symbolic values of program variables are updated when assignments are traversed, and symbolic predicates are constructed (and updated) to capture the cumulative effect of the assumptions involved when either the T or F exit from a branch statement is taken. Both types of expressions, involve only the input data symbols.

2. The automatic construction for each potentially executable path (or for a user-selected subset of paths) of a boolean expression that precisely characterizes that region of the input data space for which execution of that path is guaranteed.

3. The determination of simplified expressions for the values of all program variables, in terms of symbolic input values.

4. Automatic derivation of actual (i.e., numerical) input data that can be used as sample test data to exercise each such path.

5. Execution of the test program with execution (real or symbolic) of intermediate assertions (Floyd assertions) and/or output assertions. These assertions can characterize the programmer's intent, or guide the system to choose test data satisfying a constraint not reflected in the code of the path.

In the remaining portions of this section of the paper we discuss these system features in more detail. In particular we discuss: the meaning we give for a program path; the input language which is a subset of LISP, for the programs; the symbolic execution technique used for generating path conditions; the packages for solving systems of equalities and inequalities; the method for systematically traversing the program paths; and the manner by which the user can interact with the system to guide the generation of test data.

B. Path Analysis and Selection

We shall henceforth use the term path to refer to any control sequence for the test program beginning with the entrance or START point, and ending with the exit or HALT point. In particular, multiple executions of a loop within such a "path" will define different paths in our sense. Thus, any program containing one or more loops† contains a potentially infinite number of distinct paths. It is clear that no finite amount of testing (as opposed to formal verification of correctness) can test all such paths. However, the purpose of testing is to reveal bugs and to improve the user's confidence in the program, not to guarantee mathematical correctness. We believe that this purpose can also be served by permitting the user of a testing system to specify (for each embedded loop or FOR, WHILE, etc. statement) how many traversals he wishes the system to execute.

Conceptually, path selection and test data generation are separate processes, but, for reasons of efficiency, we have combined them in the current version of SELECT.

Before describing the path condition generator, it will be convenient to discuss the syntax and semantics of the programming language (input language) for programs that our system can handle.

C. Input Language

The language for programs to be tested by our experimental system was selected to resemble a subset of LISP. Admittedly this is an unrealistic choice from the standpoint of practicality, but it is a natural one for our purposes, first, because it facilitated making use of portions of the SRI Program Verifier (which used a similar input language), and second, because LISP is an eminently suitable language for building experimental systems. Thus, by eliminating the need for complex translators from some more popular input language to LISP, we were able to focus on the real issues involved.

Our language includes the following constructs:

1. Assignment: (SETQ x expr) assigns to the variable (identifier) x the value of the expression expr. Assignments to arrays are expressed by the syntax: (SETA arrayname $expr_1$ $expr_2$).

Here arrayname is an identifier (pointer), $expr_1$ after evaluation determines the location in the array to be changed, and the value of $expr_2$ is placed into that location. The array arrayname must previously have been declared, and the index given by $expr_1$ must fall within the array bounds, else SETA generates an error. (ELT arrayname i) returns the value in position i.

2. Arithmetic functions: The common arithmetic functions are expressed in prefix form by PLUS, TIMES, DIFFERENCE, QUOTIENT, MINUS (unary function), and ABS. PLUS and TIMES can take any number of arguments.

3. Control statements: Control is handled by a free combination of GO, FOR, WHILE, UNTIL, and COND (conditional) statement types.

The GO ("goto") has the syntax: (GO label), and executes an unconditional jump to the named label (appearing as a statement label at the top level of a PROG) in the program.

The FOR statement has the form:

```
(FOR X FROM I TO J
 (...list of executable statements...))
```

with WHILE and UNTIL options expressed as:

```
(FOR X FROM I {TO J} UNTIL predicate
 (... statement-list...))
(FOR X FROM I {TO J} WHILE predicate
 (... statement-list...))
```

The clause {TO J} is also optional in the above three forms.

*The name SELECT was chosen to suggest the system's principal feature: the automatic selection of test data to insure coverage of the program. For those who like acronyms, try Symbolic Execution Language to Enable Comprehensive Testing.

†Other than loops with predetermined numbers of traversals, e.g., "FOR I = 1 UNTIL 10 ⟨statements⟩ ... "

Conditional statements are expressed as

(COND ($expr_1$ $statement\text{-}list_1$)
($expr_2$ $statement\text{-}list_2$)
.....
))

The semantics of COND are as follows: the expressions $expr_1$, $expr_2$,...are successively evaluated in turn until one is found that yields a non-NIL value, at which point the corresponding statement-list is executed and the COND is exited without evaluation of the subsequent clauses. The alternative syntax IF $expr_1$ THEN ... ELSE IF $expr_2$ THEN ...ELSE .... is also accommodated.

4. Scoping of variables is accomplished by the LISP PROG which creates flowchart-like programs within a larger program, with local PROG variables visible only within the PROG. The syntax is:

(PROG list-of-local-variables
$statement_1$ ... $statement_n$)

5. Boolean functions: The common boolean functions (logical connectives) AND, OR, NOT are used (as prefix functions) to combine predicates for use in conditionals, WHILEs, UNTILs, and in assertions, to be discussed later. AND and OR can take any number of arguments. The primitive predicate functions are the six numerical equality-inequality functions, EQ (=), NEQ (≠), GT or GREATERP (>), LT or LESSP (<), GTQ (≥), and LTQ (≤).

6. Subroutines or function calls: The SELECT system can handle arbitrary function calls within the program to be tested, provided, of course that such a function has been defined by a suitable body of code. Such functions may call other user-defined functions (or may call themselves recursively). When a function call exits, it returns a single value (as in LISP), and may also produce side effects on external variables. The mechanism whereby SELECT handles the symbolic execution of such function calls is entirely that of macro-expansion. This approach leaves something to be desired as will be discussed in Section V below.

D. Path Condition Generator

The purpose of the path condition generator is the construction for any program path (or a set of such paths) of the necessary and sufficient boolean expression (involving the input variables only) for that path to be executed. Thus, it must construct essentially a conjunction of the test predicates (or negations of test predicates, depending on whether a TRUE or FALSE exit is taken from a test) encountered along that path. However, at each test the current symbolic values of the program variables used in that test are used.

There are two distinct approaches that can be taken to this simple expression-handling task, which are referred to as backward substitution and forward substition.

The method of forward substitution turns out to be a more natural approach here, and also avoids some of the complications produced by array assignments. In forward substitution, we essentially model the manual (or mental) action carried out by a human programmer in analyzing what happens along an execution path. The system keeps track of the current (symbolic) values of the program variables, both simple variables and subscripted ones, as control progresses forward along the path. A LISP alist (a list of dotted pairs, e.g.,

((X . (PLUS A B)) (Y . C) (Z . (TIMES A C)))

provides the actual storage mechanism, in this case values for X, Y, and Z. Note that all right-side variables in an alist dotted pair refer to the values at program entry. The alist is updated whenever an assignment statement is traversed. Another list, the hypothesis-list, similarly keeps track of the branch predicates that have been traversed, and represents the conjunction of these boolean predicates (in terms of the values of the program variables current, from the alist, at the time of symbolic execution of the branch test.) For each path an input data example is maintained that would cause the path in question to be executed. The values for input data are derived by calling the solvers described in the next subsections. When the path exit is reached the alist contains the final values of all program variables expressed in terms of the input data; the hypothesis list provides the desired conjunction of predicates, also in terms of the input data, and the input data example is a possible input for the path. The expressions in the alist and the hypothesis list are subjected to the simplifier used in the SRI program verifier.

Array assignments result in an updating of the alist, similar to simple variable assignments, but as with the case of branch predicates, array assignments can result in an updating of the hypothesis-list as well. The key relation involving arrays is the identity of two subscripts, since the implication $I = J \supset A[I] = A[J]$ must always hold. This situation arises when an assignment of the form $P \leftarrow A[J]$ is encountered, as below, and the alist contains some dotted pair, say (A[I] . Q), with a reference to an element of A as its lefthand component.

Code	Alist	Hypothesis-List
...		
P ← A[J]	(...(A[I].Q)...)	(...hypotheses....)
If we can deduce that I=J, then the lists become		
	(...(P.Q)...)	(...hypotheses...(I=J))
otherwise if I≠J		
	(...(P.A[J]...)	(...hypotheses...(I≠J))

If it happens that I=J, then the effect of this assignment (in terms of input values) is to associate P and Q. Thus, as shown above, the alist will become updated with the dotted pair (P.Q), and the hypothesis-list will receive an additional conjunct I = J. Note that, as with branch predicates, it must be determined that the hypothesis list is actually satisfiable after conjoining this new predicate. On the other hand, if the hypothesis list is also satisfiable after conjoining the expression $I \neq J$, then the alist is updated by the dotted pair (P.A[J]), exactly as for simple variables, and the hypothesis-list is augmented with the conjunct $I \neq J$. Note that an assignment of the form A[K] ← V requires only an updating of the alist by the term (A[K].V).

In summary, an assignment of the form P ← A[J] results in the establishment of new virtual paths, via tests for equality (and inequality) of J with each array index appearing in A-array references on the left-side of any dotted pair in the alist. For each such virtual path the hypothesis-list is updated with the relevant predicate, J = index, or J ≠ index, and the alist is appropriately updated relative to the corresponding assumption on J.

It should be emphasized that the backward and forward substitution methods produce entirely equivalent results. King's current system EFFIGY uses a forward symbolic execution technique like the one we are currently using, while Howden [1974] advocates the backward substitution approach. In addition to the advantages in relation to array assignments, the forward approach also permits the early exclusion of "impossible paths." Each potential augmentation of the hypothesis list which occurs whenever a branch predicate or an array assignment is confronted, first precipitates an attempt to determine which predicate outcome will result from the test data accumulated so far. Also, the system determines if there exists any data that would result in the contrary outcome. If such data do not exist, all path extensions of the contrary outcome are excluded from further consideration. In the backward substitution method this path pruning process cannot be carried out until the whole path has been traced.

E. Inequalities Solver

The hypotheses produced by the path condition generator just discussed are not an end in themselves. Instead, they constitute logical constraints from which one hopes mechanically to derive actual instances of input data satisfying these conditions. We may view a given set of path conditions as defining a subregion of the whole input data space such that any data point in this subregion will result in execution of the program along the same specified path of the program. However, the logical conditions serve only to define this subregion in an implicit sense. In order to come up with actual input data samples we need to "solve" the logical conditions. An attempt to solve the conditions is also important in revealing impossible paths. If the conditions do not possess a solution there does not exist input data that will cause execution of the corresponding path.

Most frequently in numerical programming the branch conditions of the program will be equalities and inequalities among (real or integer) variables. Hence, the path test conditions will also be (conjunctions of) such equalities and inequalities. In order to generate sample test data for a given path, we therefore need an equality/inequality solver. In the SELECT system we have experimented with several different programs for doing this task. The first program (GOMORY) [see Gomory 1963] that was written is limited to handling systems of inequalities among (positive or negative) integer variables, where the functions that combine these variables are limited to linear combinations (with constant numerical coefficients). For example,

$$3*X + 2*Y - 153*Z \geq 0$$

$$X + Y + Z \geq 0$$

$$10*X - 5*Y \leq 0$$

is a system of integer linear inequalities. Strict equality can also be handled in an obvious manner. This program uses an integer linear programming algorithm to optimize an (arbitrary) objective function, with the path tracing inequalities serving as constraints.

Since the above GOMORY algorithm is limited to solving for integer variables (these may be positive, negative or zero), some other means was needed to handle programs where the program variables could take arbitrary real values. For this purpose, a mixed integer linear programming algorithm (called BENDERS) was developed by adapting an existing algorithm due to Benders [1962]. As with the GOMORY program, however, this second algorithm is still limited to linear constraints.

The limitation of the above algorithms to linear combinations is an unacceptable, and vexing, one. For example, they could not handle an inequality like $X*Y + 10*Z - W \geq 5$ among its constraints, unless one were prepared to assign to X a trial value, and then attempt a solution (assuming the other inequalities are linear). We therefore considered various alternatives that would not be subject to that limitation. The most promising of these alternatives appears to be a conjugate gradient algorithm ("hill-climbing" program) that seeks to minimize a potential function constructed from the inequalities. There are a number of different ways in which the potential function can be constructed for a given set of inequalities, with the common feature that the desired minimum potential is attained within the path condition subregion. The present version of this algorithm appears to be quite effective computationally (though it may require some user guidance) in coming up with sample data points. It is not at all limited in the kinds of functional combinations allowed between program variables; any computable functions are permissible.

It is possible for the user, by varying parameters of this hill-climbing program, or by modifying the potential function, to operate the program in a number of different modes. For example, if it is desired to select data points on or near the subregion boundaries, there is a mode that will favor the selection of such points. It is also possible to favor the selection of test points at or near the intersections of several boundary surfaces. Finally, one can also insist that the selected test point will be well within the subregion--nominally at its "center" as defined by a potential minimum among several additively combined sub-potential functions, each such sub-potential arising from one of the inequalities of the system. However, the hill-climbing algorithm is not guaranteed to terminate, and requires user interaction to guide it.

F. Automatic Traversal of Paths

If the user so desires, SELECT will handle all distinct input-output paths such that no loop is traversed more than S times, S being supplied by the user. We call S the looping factor. We recall that in tracing out a particular path the system has generated a specific input data example that would cause execution of the path segment. When a branch point is reached it is determined which of the two branch exits the data satisfies, and then the system will continue execution of the path

extension taking this branch. The solvers are invoked to determine if there exist input data that will cause execution of the alternative branch. If there is a solution, a backtrack point is established for future analysis of the alternative branch and its extensions. If no solution is found then the extensions of the alternative branch are ruled out of future consideration. A path is terminated when a program exit is reached. The continued execution of a branch satisfied by the input data example and the backtracking proceed until a loop is traversed more than S times.

G. User Supplied Assertions as an Adjunct to the Program Code

Once having arrived at one or more sample points within a subregion corresponding to a given program path, one may cause the test program to be executed with those test data as input in order to actually test that path. An eyeball examination of the output data will generally be relied upon to determine if the program has computed intended results. It is also possible for a user to observe the symbolic output values produced by SELECT for the paths in order to try to understand what the program is really computing. In these cases the only documentation on the program provided to SELECT is the program code itself.

As another mode of operation it is possible to insert assertions, possibly in the form of programs themselves, at various points in the program including the output. These assertions can serve as

(1) executable assertions that are evaluated when the execution of the program (on numerical values of input data) reaches the assertion point, or as

(2) constraint conditions that enable a user to define subregions of the input space from which SELECT is to generate the test data, or

(3) specifications for the intent of the program from which it is possible to verify the paths of the program. Note that this does not imply that the program itself is correct, which would require that all program paths are verified.

With regard to (3) suppose that the intent of a path of a program PROG can be expressed as an output assertion; this output assertion could itself be a program TESTPROG. SELECT will extend the alist and hypothesis-list by the statements in TESTPROG, and will in turn create additional distinct virtual paths as needed. For all paths, not exceeding the looping factor, that eventually lead to an output of TESTPROG, SELECT will attempt to determine if there exist input data to PROG such that the conditions of TESTPROG are not satisfied. If such input data exists, for at least one path through TESTPROG, then either PROG or TESTPROG is in error. (As a beneficial side-effect counterexample data is produced.) It is clear that this limited form of verification can be carried out by the current system provided the statements of TESTPROG are expressed in the input language accepted by SELECT.

III. Examples

In this section we illustrate the action of SELECT on four small programs. We have chosen each of these programs to distinguish unique features of the system, and to attempt to reveal some of the difficult and controversial issues of automated testing. The first program, WENSLEYSQROOT (Wensley [1958]), is intended to be an iterative algorithm for computing the square root of a number p, $0 \leq p < 1$. However, the program actually contains a subtle error. The application of SELECT to WENSLEYSQROOT illustrates (1) the generation of real number sample test data corresponding to particular execution paths and (2) the systematic selection of executable paths. SELECT was able to generate test data useful for revealing the presence of the bug. The second program, MOVEARRAY, is intended to cyclically shift a subarray of an array by d positions, but a bug in the program causes incorrect operation for certain negative values of d. This example illustrates (3) the generation of integer-valued sample test data, (4) the array semantics that introduce extra hypotheses beyond those associated with branch statements, and (5) the generation of symbolic values for outputted variables that, in this example, permit disclosure of the bug. The third program, BUGGYFIND, is inspired by Hoare's FIND (Hoare [1961, 1971]), but contains a bug that affects a small percentage of permutations of the input array elements. The operation of SELECT on BUGGYFIND illustrates (6) the "ignoring" of impossible paths, (7) the need for output assertions if the system is to consistently generate bug-revealing sample test data, and (8) "path proving," wherein the system attempts to demonstrate that a user-supplied output assertion is satisfied for a path. The feature (8) was effective in revealing the bug in BUGGYFIND.

We wish to make a confession concerning the origin of the bugs in WENSLEYSQROOT and BUGGYFIND. In both cases the bugs are due to errors in translation of the published algorithms into LISP. Our original intention was to apply SELECT to correct versions of these programs, and in both cases it was a "struggle" for SELECT to finally reveal the presence of the bugs.

The fourth program, SCHED-D, is a practical, perhaps typical, data processing program with numerous paths, each corresponding to a different case. It is this type of program, rather than say a clever algorithm like FIND, that is best suited for a path-driven testing system.

WENSLEYSQROOT

The program, written in our input language, is depicted on the next page.

The program is supposed to compute $\sqrt{p}$ (where $0 \leq p < 1$) to accuracy e (where $0 < e \leq 1$), utilizing the following plan. For each excursion around the loop, i.e., returning to LOOP from either branch of the COND statement, the program computes an estimate of $\sqrt{p}$ to an additional bit of accuracy. The program locates p to within a subinterval of length d by successively halving the half-open interval [0,1). After k traversals of the loop, p has been determined to within $d = 2^{-k}$. If $d \leq e$, the program terminates with an estimate x = left endpoint of the subinterval.

```
(WENSLEYSQROOT
  [LAMBDA (P E)
    (* COMMENT: THIS VERSION HAS A BUG!!!!)
    (PROG (D X C TT)
          (SETQ D 1)
          (SETQ X 0)
          (SETQ C (TIMES 2 P))
          (COND
            ((LESSP C 2))
            (T (RETURN NIL)))
      LOOP(COND
            ((LTQ D E)
              (RETURN X)))
          (SETQ D (TIMES .5 D))
          (SETQ TT (DIFFERENCE C
                       (PLUS (TIMES 2 X) D)))
          (COND
            ((GTQ TT 0)
              (SETQ X (PLUS X D))
              [SETQ C (TIMES 2 (DIFFERENCE C
                          (PLUS (TIMES 2 X) D]
              (* COMMENT: THE TWO PRECEDING .
                 STATEMENTS SHOULD BE INTERCHANGED
                 TO MAKE THE PROGRAM CORRECT)
              (GO LOOP))
            (T (SETQ C (TIMES 2 C))
               (GO LOOP]
```

In operating on a particular input-output path of this program, SELECT will produce an alist, which will include a numerical value for the output variable x, an hypothesis list, and an input data example which will bind values to the two input variables p and e. The user can observe the values returned for x, p, and e to see if $\sqrt{p} - e < x < \sqrt{p}$--the intended relationship among the values at the output. [For any path with a looping factor of one SELECT does not derive test data revealing the presence of the bug.] A looping factor of two is required to derive test data that reveals the bug.

For example consider one of the single traversals around the main loop. The alist becomes:

$$(d \cdot 0.5)(tt \cdot 2p-0.5)(x \cdot 0.5)(c \cdot 4p-3) \quad .$$

The hypothesis-list contains the conjuncts

$$(p < 1)(\text{NOT}(1 \geq e))(2p - 0.5 \geq 0)(0.5 \leq e) \quad .$$

The input data example produced by the Benders algorithm is

$$p = .9995,\ e = .9995 \quad ,$$

which satisfies the intent of the program, since x = 0.5.

In examining another path SELECT attempts to find if there exist input data that satisfy the first three conjuncts of the above hypothesis list, together with the clause (NOT(0.5 ≤ e))--i.e., the complement of the fourth conjunct. A solution is p = 0.9995, e = .4995. Continuing along the extension of the program code following the false side of the branch (LTQ 0 E), we update the alist with

$$(d \cdot 0.25)(tt \cdot 4p-4.25) \quad .$$

The branch statement (GTQ TT D) evaluates to false with the current example input data (since $p < 1 \supset 4p - 4.25 < 0$). Thus c gets updated, and the true exit from the branch (LTQ D E) leads to the program exit. For this path the computed test data is p = .9995, e = .4995, and with these input values the program returns as an approximate square root of p the value x = 0.5. A user can determine that the intent of the program is violated--although barely--since the error is $\sqrt{.9995} - 0.5 > 0.4995 = e$. Hence, the bug is "revealed."

MOVEARRAY

Here we apply SELECT to a program in which the programmer has forgotten to handle a particular case that he intended the program to accommodate. The intention of the program is to move a subarray A[m:n] of an array A[1:p], $1 \leq m \leq n \leq p$, by d positions. It is assumed that d is bounded so that if $d \geq 0$ then $m + d \geq 1$, and if $d \geq 0$, then $n + d \leq p$. Of course the original array contents at positions m + d, m + d + 1, ..., n + d are destroyed in the process. The programmer presents the following program as a solution:

```
(MOVEARRAY
  [LAMBDA (M N D)
    (FOR I FROM N TO M BY -1 DO (SETA A (PLUS I D)
                                        (ELT A I])
```

For $d \geq 0$ the program is correct, but for $(d < 0) \wedge (n - m > |d|)$, the values of certain positions of the subarray are destroyed before they have an opportunity to be moved. A correct program would commence the iteration with i = m + d for negative values of d and with i = n + d for positive values of d. In any event the application of SELECT to the faulty MOVEARRAY program provides enough information to the user to reveal the presence of the bug.

Let us assume SELECT "executes" MOVEARRAY with the numerical values m = 2, n = 4, and symbolic values for d, p and the array elements. Thus three excursions around the FOR loop are to be taken, and no hypotheses are generated involving index checking. The first excursion established the alist entry

$$(A[4 + d] \cdot A[4]) \quad ,$$

where the quantity on the right side of the equality is always taken to apply prior to program execution. Corresponding to the second excursion we have the assignment $A[3 + d] \leftarrow A[3]$, where A[3] refers to the current value. Depending on the value of d there is uncertainty whether the current A[3] is the original A[3] or the original A[4]. SELECT recognizes the existence of two possibilities and establishes a branch point. Corresponding to one choice the hypothesis

$$4 + d = 3$$

is established and noted by the Gomory algorithm to have the solution d = -1. (SELECT also establishes the complementary hypothesis $4 + d \neq 3$, which in turn will lead to another solution for d.) The alist after the second excursion thus becomes

$$(A[3 + d] \cdot A[4])(A[4 + d] \cdot A[4]) \quad .$$

For the third and final excursion we have the assignment A[2 + d] ← A[2]; but for d = -1 the current value of A[2] is the original value of A[4]. Thus the final alist is

(A[2 + d] · A[4])(A[3 + d] · A[4])(A[4 + d] · A[4]),

and the sample input data is d = -1.

For this case it suffices to analyze the results of the symbolic execution to detect a bug--that is, it is not necessary or desirable to generate sample values for the array elements. The rightmost element of the alist indicates that, for d = -1, the final value of A[3] is to be the original value of A[4], which is as intended. However, the second element of the alist indicates that the final value of A[2] is to be the original value of A[4] which is not as desired. Likewise, the final value of A[1] is seen to be (incorrectly) the original A[4].

BUGGYFIND

The two previous examples illustrated the ability of SELECT to generate relevant sample test data. In WENSLEYSQROOT the derived test data showed that execution combined with observation of the numerical values of output variables for a sufficient number of loop traversals could enable a user to note a bug. The bug in MOVEARRAY was revealed by SELECT deriving a value for one variable and the user observing the symbolic value of output variables. We now discuss a program bug which the above facilities are not effective in revealing, but which seems to necessitate an analysis of the program with a user-supplied output assertion.

The program BUGGYFIND is Hoare's FIND (Hoare [1961]) but with an added insidious bug. We recall that the purpose of FIND is to permute the elements of an array such that all elements to the left of some inputted position f, have value not exceeding A[f], and all elements to the right of f have value not less than A[f]. Of course, any array-sorting program would achieve this goal, but FIND seems to be appreciably more efficient. The strategy in FIND is to move "small-valued" elements to the left and "large-valued" elements to the right. Our program with the error is depicted below.

In the program, left and right boundaries (originally at positions 1 and NN respectively) are gradually moved together. To the left of the "left" boundary, elements are known to be small and to the right of the "right" boundary, elements are known to be large. The key in determining whether an element is large or small is in its comparison with some fixed value. In Hoare's correct version this fixed value, R, is assigned the value A[F] whenever a new right or left boundary is established. Thus any element smaller than R is known to be smaller than any element larger than R. In BUGGYFIND instead of the comparison being made with respect to R it is made with respect to the current value of A[F]. But, since A[F] is subject to exchange with another element, it is conceivable that, after such an exchange, future comparisons with A[F] do not guarantee the satisfaction of the large-small relationship.

```
(BUGGYFIND
[LAMBDA (NN F)
  (PROG (M N I J TEMP)
        (SETQ M 1)
        (SETQ N NN)
        [WHILE (LESSP M N) DO
        (SETQ I M)
        (SETQ J N)
        [WHILE (LTQ I J)
           DO (WHILE (LESSP (ELT A I)
                            (ELT A F))
                 DO (SETQ I (ADD1 I)))
              (WHILE (LESSP (ELT A F)
                            (ELT A J))
                 DO (SETQ J (SUB1 J)))
              (COND
                ((LTQ I J)
                  (SETQ TEMP (ELT A I))
                  (SETA A I (ELT A J))
                  (SETA A J TEMP)
                  (SETQ I (ADD1 I))
                  (SETQ J (SUB1 J]
        (COND
          ((LTQ F J)
             (SETQ N J))
          ((LTQ I F))
            (SETQ M I))
          (T (GO L]
     L])
```

This bug is quite difficult to detect by conventionally testing the program with random array inputs since the original goal of FIND fails to be satisfied by extremely few input array patterns. Assuming (for an array of four elements) that no pair of array element values are identical, there are only two permutations (out of 24 possible) for which a wrong answer will emerge. Let us now review the operation of SELECT on the code of BUGGYFIND. In particular, let us consider a path for which the GOMORY algorithm could have derived test data that reveals the bug, but failed to do so. That is, some feasible input solutions to the hypothesis-list are relevant to the bug, but for many other feasible solutions BUGGYFIND gives no indication of error.

The program BUGGYFIND is tested with NN = 4, F = 3. One relevant path generates the hypothesis-list

(NOT(A[1] < A[4]))(A[2] < A[1])(A[3] < A[4])
(NOT(A[1] < A[3])) ,

and the alist

(A[1] · A[3])(A[3] · A[4])(A[4] · A[1]) .

A[2] is missing as a lefthand term since it is not assigned a new value on the path. The input example that SELECT generates, to satisfy the hypothesis list is

A[1] = 1, A[2] = 0, A[3] = 0, A[4] = 1 .

The final array can be derived by executing FIND with these values, or by referring to the alist. In any event the final array is

A[1] = 0, A[2] = 0, A[3] = 1, A[4] = 1 .

The final array values obviously give no indication of the lurking bug.

Let us now consider the same input-output path of BUGGYFIND, but with an output assertion appended to the path. The output assertion is reflected in the code of the program TESTFIND,

```
(TESTFIND
  [LAMBDA (NN F)
    (PROG NIL
       (BUGGYFIND NN F)
       (RETURN (AND
          (FOR I FROM 1 TO F ALWAYS
                 (LTQ (ELT A I)(ELT A F)))
          (FOR I FROM (ADD1 F) TO NN
             ALWAYS (LTQ (ELT A F)
                         (ELT A I])
```

which tests the relationship among the final (symbolic) values produced by BUGGYFIND for this path. If SELECT can identify input data such that TESTFIND is not true then there is an error either in BUGGYFIND or TESTFIND. SELECT, in effect, carries out a macro-expansion of TESTFIND and considers the paths of TESTFIND in the same systematic way as it did for BUGGYFIND. What SELECT produces, for the path in question in BUGGYFIND is the answer that there exists data, such that TESTFIND returns FALSE, and that satisfies the hypothesis-list:

$$(NOT(A[2] \leq A[4]))(NOT(A[1] < A[4]))$$
$$(A[2] < A[1])(A[3] < A[4])(NOT(A[1] < A[3]))$$

An example of an input array satisfying these hypotheses is

$$A[1] = 3,\ A[2] = 2,\ A[3] = 0,\ A[4] = 1\ .$$

The resultant output array is

$$A[1] = 0,\ A[2] = 2,\ A[3] = 1,\ A[4] = 3\ ,$$

which reveals the bug since, for F = 3, A[2] > A[3].

This example has illustrated the inability of SELECT always to derive relevant test data. However, the inclusion of an output assertion, as a program involving functions and predicates understandable to SELECT, can reveal a bug that affects at least one of the paths that SELECT inspects. As a final note with regard to BUGGYFIND there are several hundred syntactic paths for an array of length 4, but SELECT weeds out the many impossible paths and produces data for the 18 feasible paths.

Schedule D--A simple non-loop program

A simple example of a program without any control loops is provided by the program SCHED-D (in LISP) shown below, that calculates the net (positive or negative) contribution to income from a combination of the net short-term and long-term capital gains or losses. Mainly for tutorial reasons we have simplified the calculation by assuming that Parts IV-VI of Schedule D, Federal Tax Form 1040 do not apply (as is often actually the case, unless one uses the Alternative Tax Computation, or has a long-term capital loss component carryover from more than three years back, etc.). The reader is referred to the IRS tax instructions for explanations of these terms.

```
(SCHED-D
  [LAMBDA (SHORTGN LONGTGN ADJTAXINC)
    (PROG (L14 L15A L16A)
          (SETQ L14 (PLUS SHORTGN LONGTGN))
          [COND
            [(GT L14 0)
              (COND
                ((LTQ LONGTGN 0)
                  (SETQ L15A 0)
                  (SETQ L
                    (DIFFERENCE L14 L15A))
                  (RETURN L))
                (T (COND
                     ((LTQ LONGTGN L14)
                       (SETQ L15A
                         (TIMES .5 LONGTGN))
                       (SETQ L
                         (DIFFERENCE L14 L15A))
                       (RETURN L))
                     (T(SETQ L15A
                         (TIMES .5 L14))
                       (SETQ L
                         (DIFFERENCE L14 L15A))
                       (RETURN L]
            (T COND
                ((GTQ SHORTGN 0)
                  (SETQ L16A
                    (TIMES .5 (ABS L14)))
                  (GO OUT))
                ((GTQ LONGTGN 0)
                  (SETQ L16A (ABS L14))
                  (GO OUT))
                (T [SETQ L16A
                     (ABS (PLUS SHORTGN
                                (TIMES .5 LONGTGN]
                   (GO OUT]
       OUT (SETQ L (MIN 1000 ADJTAXINC L16A))
           (RETURN (MINUS L])
```

Even with these simplifying assumptions there are enough significan tly different input cases to consider to make this an interesting example. The program enters the computation of Part III, Schedule D at a point where the net short-term gain/loss and net long-term gain/loss have already been determined by summing their components (in Parts I and II). These two (algebraically signed) quantities, called SHORTGN and LONGTGN in the program, together with the adjusted taxable income (ADJTAXINC), a non-negative quantity, constitute the three input parameters to SCHED-D. The program computes a single (signed) quantity L representing the desired combination of short-term and long-term gains or losses. Various local variables in the program, e.g., L14, L15, and L16A, represent entries on correspondingly numbered lines of the tax form.

There are basically six distinct paths through this flowchart, if we consider that the calculation of the minimum of the three quantities: L16A, $1000, and ADJTAXINC corresponds to a simple procedure call of the function MIN(x,y,z). However, our present mechanism for such procedure calls actually does a macroexpansion of the procedure code, which itself involves four sub-paths, so that a total of 3 + 4*3 = 15 paths through the (expanded) program results. All the program branching occurs on inequality tests of (positive, zero, or negative) real numbers, and although there are no loops, the number of distinct possibilities is large enough so that it is difficult or impossible for most

people to keep all the contingencies clearly in mind without a flowchart or table of cases. When the excluded possibilities (e.g., the Alternative Tax Computation) are taken into account, this becomes quite hopeless without some such aids.

The point to our SELECT system in relation to such examples is that it will automatically generate sample input data (SHORTGN, LONGTGN, ADJTAXINC) for each of the possible program paths. Let us examine how SELECT handles just one of these 15 paths. Referring to the subfunction MIN and to the text of the program SCHED-D, the path in question corresponds to the FALSE exit from the "(GTQ SHORTGN 0)", at which point MIN is invoked on the arguments, X = 1000, Y = ADJTAXINC, Z = L16A = (TIMES .5 (ABS (PLUS SHORTGN LONGTGN))). The branching within the procedure MIN is: (LESSP Y X) is FALSE, and (LESSP X Z) is TRUE, resulting in an output value (MINUS L) = (MINUS X) = -1000. For this path, SELECT constructs input test data: SHORTGN = 0, LONGTGN = -2001, and an adjusted-taxable income ADJTAXINC = 1000. The reader may convince himself by tracing through the program with this data to see that the stated path is executed and that the output (-1000), i.e., a net deduction of $1000, is correct. Naturally, when the program is actually executed with this input data, this is also confirmed.

IV. Related work

A number of researchers have independently arrived at and discussed the same basic ideas on which our system is founded, i.e., generating logical conditions for each program path, and solving these logical equations to provide instances of input data which will drive the program through that execution path. However, with one exception, our system is the only one we know of which actually attempts to implement these ideas in a system that can also carry out a semantic analysis of a program. The exception is an experimental system currently being worked on by J. King, [1975], at the IBM Watson Research Center, Yorktown Heights, New York, King's system (called EFFIGY) is an outgrowth of his earlier [1969] work on formal verification of correctness. EFFIGY performs (as does SELECT) a symbolic execution that develops path conditions by forward substitution. It also makes use of a deductive system (theorem prover) derived from his earlier program verifier to generate test data for each path, and to prove that some paths are "impossible." It handles simple variables and subscripted variables (arrays).

The experimental system described in this paper is closely related to a system plan drawn up independently by Howden [1974]. In particular, Howden also makes use of the notions of analyzing a program into distinct paths, generating logical conditions to correspond with these paths, and solving these sets of conditions to yield, automatically, test data with which to exercise the program. However, to the best of our knowledge, Howden has not carried out any actual implementation of his scheme. Nor does Howden's scheme accommodate array variables.

An automated Verification System (AVS) is being developed by Miller [1974, 1974a] of General Research Corporation. AVS insures that each line of code is tested at least once by given test data, and provides programming aids to the user/programmer in the form of path conditions, and in establishing which further inputs will execute code in the remaining, unexercised portions. It does not, however, attempt to be as exhaustive as our system is in exercising each line of code for each path that might include that line of code. Nor does it do any significant processing of the path conditions.

The TRW Automated Software Quality Assurance System (PACE) of Brown et al. [1973] is basically a collection of tracing and managerial tools which assist in assessing test coverage, but do not generate test data.

We also call the reader's attention to excellent characterizations of the whole subject of program testing by Hetzel [1973, 1973a] in his book (Hetzel [1973b]) devoted to a collection of papers on this subject.

V. Conclusions

We have described an experimental system, SELECT, that can aid a user in testing and debugging programs. The system performs a symbolic execution of the paths (input to output) of the program. Loops are handled in a systematic manner under user control; each loop can be executed as many times as a user feels necessary to convince himself of its correctness. In practical programs many syntactic paths may not be executable paths. In "growing" a path, SELECT attempts to determine if there exist input data that will drive execution through the subpath developed so far. If no such data exists, the path is impossible and SELECT ceases its consideration of that path. Thus the number of paths actually subjected to execution can be significantly reduced.

For each executable path SELECT can produce the following:

- Sample input test data that will cause the path to be executed. The user can run the program with this data and observe the outputted results.
- Simplified symbolic values for variables that do not require actual numerical values in order to ensure execution of the paths. These symbolic values can be observed by the user as an aid to understanding what the path in question is computing.
- A proof that a user-supplied output assertion is satisfied for all data that drives the program through the path.

At present SELECT can decide each of the above three issues if the hypotheses that uniquely define a path correspond to a system of linear inequalities (with integer or real or mixed constraints on the solution values) or to certain nonlinear inequalities. Array semantics and rules for algebraic and logical simplification are also known to SELECT.

The present system can be useful for moderate-size programs in that it symtematizes many of the debugging tricks now in common practice. The system also provides an interaction with the user at a symbolic level that should be comprehendable to the user. Although there is extensive symbolic manipulation being carried out, it is significantly less than that associated with mathematical program verification by, say, Floyd's method. Hence, SELECT can probably handle larger programs than contemporary prototype verification systems, but may provide less assurance regarding the correctness of the program.

On the other hand, testing a program with SELECT-generated data in the actual computer environment will be realistic with respect to features (e.g., round-off, overflow, compiler and operating system) that may be incorrectly axiomatized (or ignored!) in a formal proof.

There are clearly significant limitations to the present SELECT system, the most prominent being the lack of a proper mechanism for handling program abstractions. As a minimum there should be a mechanism for hierarchically handling calls to subroutines--a facility not present in the current system. We have for the present arranged matters in the program analyzer so as to provide the effect of a macroexpansion--substituting the expanded (and instantiated) code into the program itself. [But, this expedient (though it works) appears ultimately to be a self-defeating way to proceed, partly because it leads to an explosion in the number of paths to be followed, and partly because it forecloses any possibility of carrying out the testing of a modularly constructed program in a hierarchical manner.] Our view as to the proper way to handle this situation is to attempt to characterize each subroutine invoked in a program by input and output assertions, much as in the Floyd technique for program verification. Such subroutine specifications need not be complete in the formal correctness sense, nor do we need to carry out formal proofs for each subroutine (although, that possibility is not ruled out). Instead, we need only test each such subroutine (as a separate and distinct program module) until we have sufficient confidence that it meets its input/output specifications. From that point on, whenever that module is invoked it can be replaced, for purposes of testing the overall program, by those input/output specifications. By this means we seek to accommodate program modularity (and hierarchy) in a natural and efficient way into the test data generation process. There are still unresolved issues in this connection. Ultimately, however, we feel that such a combination of testing with elements of the formal verification process is the best way to proceed. The main issues are:

- Choosing a suitable specification language for the assertions.
- Expanding the manipulative powers of the system beyond that of inequalities and algebraic simplification, to permit the processing of higher-level operations that will originate from the subroutine specifications.

Another desirable augmentation relates to the detection of various improper conditions. There are a number of improper conditions (invalid operations) that can be detected by a forward-acting symbolic executor which cannot be detected by conventional compilers. Among such conditions are the potential of overflow and underflow conditions, division by zero, out-of-bounds array references, referencing an unitialized variable, and perhaps even infinite looping. In some simple cases some of these conditions can even be detected at compile time, but in general their detection demands the presence of a fairly sophisticated deductive system. An excellent discussion of how the analysis and detection of such error conditions can be integrated into a program verification system has been carried out by Sites [1974] in a doctoral thesis. We plan to augment the SELECT system by incorporating features of this type.

Beyond these augmentations and embellishments there is a significant need to experiment with formal testing concepts of the type discussed in this paper. We foresee program verification, or better the synthesis of provably correct programs, as being the most promising approach toward improving software reliability. Although these techniques should ultimately be accepted by most programming shops, the testing of programs will remain for many years as the primary certification tool. We are convinced (as are many others who have studied the problem of testing) that there is little hope that the fully automatic generation of test data will be effective where used alone. A programmer's insight on focusing on certain sections of a program or on certain data relationships seems invaluable in deciding on relevant test data. However, the main fallibility of programmers is in covering all cases of interest--a lack that a symbolic processing system of the type discussed here can handle.

Acknowledgments

The authors wish to acknowledge the contributions of Andrew J. Korsak and Milton W. Green to the work reported here, in particular to the algorithms used for solving systems of linear inequalities and nonlinear inequalities.

We also wish to thank Peter J. Wong, Ralph E. Keirstead, Lawrence Robinson and Jack Goldberg for numerous illuminating conversations on the subjects of programming methodology, program testing and debugging, and linear and nonlinear programming.

References

[Benders 62]
J. F. Benders, "Partitioning Procedures for Solving Mixed-Variables Programming Problems," Numerische Mathematik 4, pp. 238-252 (1962).

[Brown, et al., 73]
J. R. Brown, A. J. deSalvio, D. E. Heine, and J. G. Purdy, "Automated Software Quality Assurance," in Program Test Methods (W. C. Hetzel, ed.), Prentice-Hall, Inc., Englewood Cliffs, N. J.; pp. 181-203 (1973).

[Deutsch 73]
L. P. Deutsch, "An Interactive Program Verifier," Ph.D. dissertation, University of California, Berkeley, California (June 1973).

[Elspas, et al., 72]
B. Elspas, M. W. Green, K. N. Levitt, and R. J. Waldinger, "Research in Interactive Program Proving Techniques," SRI Report 8398-II, Stanford Research Institute, Menlo Park, California (1972).

[Elspas, et al., 73]
B. Elspas, K. N. Levitt, and R. J. Waldinger, "An Interactive System for the Verification of Computer Programs," Final Report, SRI Project 1891, Stanford Research Institute, Menlo Park, California (1973).

[Elspas 74]
B. Elspas, "The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness," Interim Report, SRI Project 2686, Stanford Research Institute, Menlo Park, California (July 1974).

[Gomory 63]
R. E. Gomory, "An Algorithm for Integer Solutions to Linear Programs," in Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe (eds.), McGraw-Hill, New York (1963).

[Hetzel 73]
W. C. Hetzel, "A Definitional Framework," in Program Test Methods (Hetzel 1973b); pp. 7-10.

[Hetzel 73a]
W. C. Hetzel, "Principles of Computer Program Testing," in Program Test Methods (Hetzel 1973b); pp. 17-28.

[Hetzel 73b]
W. C. Hetzel, Program Test Methods, A collection of papers based on the Proceedings of the Computer Program Test Methods Symposium held at the University of North Carolina, Chapel Hill, 1972, and edited by W. C. Hetzel, Prentice-Hall, Inc., Englewood Cliffs, N. J. (1973).

[Hoare 61]
C. A. R. Hoare, "Algorithm 65, FIND," Comm. ACM, Vol. 4, No. 7, p. 321 (1961).

[Hoare 71]
C. A. R. Hoare, "Proof of a Program: FIND," Comm. ACM, Vol. 14, No. 1, p. 39 (1971).

[Howden 74]
W. Howden, "Methodology for the Automatic Generation of Program Test Data," Technical Report No. 41, Dept. of Information and Computer Science, University of California, Irvine (15 February 1974).

[Igarashi, et al. 73]
S. Igarashi, R. London, and D. Luckham, "Automatic Verification of Programs I: A Logical Basis and Implementation," Memo AIM-200, Stanford Artificial Intelligence Lab., Stanford, California (May 1973).

[Katz-Manna 73]
S. M. Katz and Z. Manna, "A Heuristic Approach to Program Verification," Proc. IFCAI-73 (August 1973).

[King 69]
J. C. King, "A Program Verifier," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania (September 1969).

[King 75]
J. C. King, "A New Approach to Program Testing," 1975 International Conference on Reliable Software.

[Miller 74]
E. F. Miller, Jr., et al., "Structurally Based Automatic Program Testing," paper presented at EASCON '74, Washington, D.C., October 7-9, 1974, available from General Research Corporation, Santa Barbara, California, August 12, 1974.

[Miller 74a]
F. F. Miller, Jr., "Overview and Status - Program Validation Project," Report dated 1 October 1974, General Research Corporation, Santa Barbara, California.

[Sites 74]
R. Sites, "Clean Termination of Computer Programs," Ph.D. dissertation, Stanford University, Stanford, California (June 1974).

[Teitelman, et al., 72]
W. Teitelman, D. G. Bobrow, A. K. Hartley, and D. L. Murphy, BBN-LISP, Tenex Reference Manual (Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts (1972).

[Wegbreit 74]
B. Wegbreit, "The Synthesis of Loop Predicates," Comm. ACM, Vol. 16, No. 2, pp. 102-112 (Feburary 1974).

[Wensley 58]
J. H. Wensley, "A Class of Non-Analytic Iterative Processes," Computer Journal, Vol. 1, No. 4, pp. 163-167 (1958).