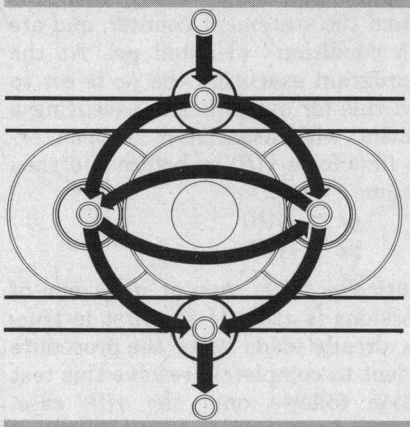
Applications of Symbolic Execution to Program Testing

John A. Darringer
James C. King
IBM Thomas J. Watson Research Center



The advanced method of symbolic evaluation can be applied to program testing situations with results close to those of formal correctness proofs—but without the high cost.

Symbolic execution provides a basis for a program analysis tool that allows one to choose intermediate points in a spectrum ranging between individual test runs and general correctness proofs. One can perform a single "symbolic execution" of a program that is equivalent to a large (possibly unbounded) number of normal test runs. Not only can test results be checked by careful manual inspection, but if a machine interpretable specification is supplied, the results can be checked automatically. Furthermore, by varying the amount of symbolic data and program specification introduced, one can move from a normal execution (no symbolic data) to a symbolic execution that provides a proof of correctness.

The particular contexts for performing the symbolic execution can also be varied—to produce results useful for estimating the test coverage of a set of particular test cases, to devise specific test data to force the program to follow specific execution paths, and to systematically explore a set of interesting execution paths.

An experimental system for symbolic program execution called Effigy has been developed and described in previous papers.[1,2] The next two sections review the particular terminology and notions of symbolic execution developed in the Effigy project. The remainder of the paper describes how symbolic execution can be used to solve a variety of program testing problems. Related work has been done by Boyer et al.,[3] Clarke,[4] and Howden.[5,6] Similar work emphasizing program correctness proofs has been done by Topor and Burstall,[7,8] Deutsch,[9] Good,[10] and Boyer and Moore.[11]

Symbolic execution

The notion of symbolically executing a program follows quite naturally from normal program execution. First assume that there is a given programming language (say, Algol-60) and the normal definition of program execution for that language. One can extend that definition to provide a symbolic execution in the same manner as one extends arithmetic over numbers to symbolic algebraic operations over symbols and numbers. The definition of the symbolic execution is such that trivial cases involving no symbols are equivalent to normal executions, and any information learned in a symbolic execution applies to the corresponding normal executions as well.

An execution of a procedure becomes symbolic by introducing symbols as input values in place of real data objects (e.g., in place of integers and reals). Here "inputs" is to be taken generally as meaning *any* data external to the procedure, including those obtained through parameters, global variables, explicit **read** statements, etc. Choosing symbols to represent procedure inputs should not be confused with the similar notion of using symbolic program-variable names. A program variable may have many different specific values associated with it during a particular execution, whereas a symbolic input symbol is used in the static mathematical sense to represent some unknown yet *fixed* value.

Once a procedure has been initiated and symbolic inputs are assigned, execution can proceed as in a normal execution except when the symbolic inputs are encountered. This can occur in two ways: com-

0018-9162/78/0400-0051$00.75

putation of expressions involving procedure inputs, and conditional branching that is dependent on the symbolic inputs.

Computation of expressions. The programming language has a set of basic computational operators such as addition (+), multiplication (×), etc., which are defined over data objects such as integers. Each operator is extended to deal with symbolic data. For arithmetic data this can be done by making use of the usual relationship between arithmetic and algebra. The arithmetic computations specified by these operators can be "delayed" or generalized by the appropriate algebraic formula manipulations. For example, suppose the symbolic inputs **a** and **b** are supplied as argument values to a procedure with formal parameter variables A and B. Denote the value of a program variable X by v(X). Then initially, v(A) = **a** and v(B) = **b**. If the assignment statement $C := A + 2\times B$ were symbolically executed in this context, C would be assigned the symbolic formula (**a** + 2×**b**). The statement $D := C - A$, if executed next, would result in v(D) = 2×**b**.

Similar symbolic generalizations can be made, at least in theory, for all computational operations in the programming language. In the most difficult case, one could at least *record* in some compact notation the sequence of computations that would have taken place had the arguments been non-symbolic. The success in doing this in practice depends upon how easily these recordings can be read and understood and how easily they can be subsequently manipulated and analyzed mechanically.

Conditional branching. Consider the typical decision-making program statement, the **if** statement, taking the form:

if B **then** S_1 **else** S_2

where B is some Boolean-valued expression and S_1 and S_2 are other statements. Normally, either v(B) = **true** and statement S_1 is executed or v(B) = **false** and statement S_2 is executed. However, during a symbolic execution v(B) could be **true**, **false**, or some symbolic formula over the input symbols. In the latter case, the predicates v(B) and ¬v(B) represent complementary constraints on the input symbols that determine alternative control-flow paths through the procedure. This case is called an "unresolved" execution of a conditional statement, a notion that will be refined as the presentation develops. Since both alternative paths are possible, the only complete approach is to explore both: the execution forks into two "parallel" executions, one assuming v(B), the other assuming ¬v(B).

Assume the execution has forked at an unresolved conditional statement and consider the further execution for the case where v(B). The execution may arrive at another unresolved conditional statement with associated Boolean, say C. Expressions v(B) and v(C) are both over the procedure input symbols, and it is possible that either v(B) ⊃ v(C) or v(B) ⊃ -v(C). Either implication being **true** would show that the assumption made at the first unresolved execution, namely v(B), is strong enough to resolve the subsequent test, namely to show that either v(C) or ¬v(C).

Because the assumptions made in the case analysis of one unresolved conditional statement execution may be effective in resolving subsequent unresolved statement executions, they are preserved as part of the execution state, along with the variable values and the statement counter, and are called the "path condition" (denoted **pc**). At the beginning of a program execution the **pc** is set to **true**. The revised rule for symbolically executing a condition statement with associated Boolean expression B is to first form v(B) as before and then form the expressions

pc ⊃ v(B)
pc ⊃ ¬v(B).

If **pc** is not identically **false**, then at most one of the above expressions is **true**. If the first is **true**, the assumptions already made about the procedure inputs are sufficient to completely resolve this test and the execution follows only the v(B) case. Similarly if the second expression is **true** it follows the ¬v(B) case. Both of these cases are considered "resolved" or non-forking executions of the conditional statement.

Symbolic execution provides an advanced testing methodology, not directly a proof-of-correctness technique.

A truly unresolved (forking) execution arises in the remaining case when neither expression is **true**, i.e., given the earlier constraints on the procedure inputs (**pc**), v(B) and ¬v(B) are both satisfiable by some non-symbolic procedure inputs. As discussed above, unresolved conditional statement executions fork into two parallel executions—one when v(B) is assumed, in which case the **pc** is revised to **pc** ∧ v(B), the other when ¬v(B) is assumed and then **pc** becomes **pc** ∧ ¬v(B). Note that the forking is a property of a conditional statement *execution*, not the statement itself. One execution of a particular statement may be resolved, yet a later execution of the same statement may not.

The **pc** is the accumulator of conditions on the original procedure inputs which determine a unique control path through the program. Each path, as forks are made, has its own **pc**. No **pc** is ever identically **false** since the original **pc** is **true** and the only changes are to replace **pc** by **pc** ∧ q and only in the case when **pc** ∧ q is satisfiable. (If **pc** ⊃ ¬q is not a theorem then ¬(**pc** ⊃ ¬q) is satisfiable and then so is the logically equivalent expression **pc** ∧ q.) Each path caused by forking also has a unique **pc** since none are identically **false** and they all differ in some term, one containing a q the other a ¬q.

Note that establishing **pc** $\supset$ v(B) or **pc** $\supset \neg$v(B) to be **true** is a general theorem-proving problem. If the programming language admits even quite elementary objects and operators (e.g., simply integers and +, $\times$, and =), this theorem-proving problem is generally undecidable. However, one would hope that, in practice, an adequate theorem prover could be built that would be able to cope with most of the real problems. If for some branch a valid theorem cannot be proven, the worst that happens is that the symbolic execution will follow a path which is actually impossible.

Symbolic execution tree

One can characterize the symbolic execution of a procedure by an "execution tree." Associate with each program statement *execution* a node, and associate with each transition between statements a directed arc connecting the associated statement nodes. For each forking (unresolved) conditional statement the associated execution node has more than one arc leaving it, each labeled by and corresponding to the path choice made in the statement. In the previous discussion of **if** statements there were two choices corresponding to v(B) and $\neg$v(B). The node associated with the first statement of the procedure would have no incoming arcs and the terminal statement of the procedure (followed by the final **end**) is represented by a node with no outgoing arcs.

Also associate the complete current execution state, i.e., variable values, statement counter, and **pc** with each node. In particular, each terminal node will have a set of program variable values given as formulae over the procedure input symbols; and a **pc**, which is a set of constraints over the input symbols, characterizing the conditions under which those variable values would be computed. A user can examine these symbolic results for correctness, as he would non-symbolic test output, or substitute them into a formal output specification which should then simplify to **true**, assuming the value of the associated **pc** is **true**.

The execution tree for a program will be infinite whenever the program contains a loop for which the number of iterations is dependent, even indirectly, on some procedure inputs. It is this fact that prevents symbolic execution from directly providing a proof-of-correctness technique. Symbolic execution is indeed an *execution* and at least in this simplest form provides an advanced *testing* methodology. Program correctness proofs can be performed over symbolic execution trees by introducing correctness predicates and some form of inductive argument. This observation provides an interesting perspective on the relationship between program testing and correctness proofs and is discussed briefly in a later section.

Applications of symbolic execution to testing

A program could be built to perform symbolic execution for a particular programming language; call it a "symbolic interpreter." Such an interpreter can be used in several ways to convince a programmer that a program is correct. Depending on how much of the program's behavior is expressed formally, the value of a symbolic interpreter can range from little (when used for conventional non-symbolic testing) to a great deal (when used in a formal proof of correctness).

Conventional testing. The semantics of normal execution are a subset of that for symbolic execution. The simplest application of the interpreter is conventional testing—i.e., executing the program for specific inputs and examining the correctness of the specific outputs. However, it is well known that this type of testing is not dependable and usually provides little confidence in a program's correctness.

To clarify some of the different applications let us consider a small example: SEARCH, a program to perform binary search of an integer array, A, for a particular value, X. This program is shown in Figure 1.

```
procedure SEARCH(A, LB, UB, X);
    integer A(LB : UB), LB, UB, X, MID;
    Boolean FOUND;
    FOUND := false;
    while LB≤UB ∧¬FOUND do;
        MID := (LB+UB)/2;
        if      X < A(MID) then UB := MID−1;
        else if X > A(MID) then LB := MID+1;
                                else   FOUND := true;
    end;
    print(FOUND,MID);
end;
```

Figure 1. Procedure SEARCH.

Typically, to perform a conventional test, one must select values in an input space that is arbitrarily large in several dimensions. In the case of SEARCH values must be chosen for the upper and lower bounds, UB and LB, for each element of the array, A, and for the search value, X.

Considering the typically unbounded set of possible tests, it is surprising that testing provides any confidence at all—but it does. Experienced programmers who understand a program are able to select individual tests that they believe represent large subsets of the input space. One might fill a fixed-length array with random values in a sorted order and then execute SEARCH for values not present in the array, present in the middle, present at each boundary, and then argue that if the program works for these cases it must work in all cases. It is this informal technique of experienced testers that is captured by symbolic testing.

Symbolic testing. With symbolic testing the programmer does not have to select particular values for testing his program. Instead he supplies symbolic values to a procedure for all required inputs. Execution of the program proceeds as described previously until an unresolved conditional branch is found. At this point the programmer must decide which way the program should branch or supply additional constraints over the inputs to resolve the condition.

The decisions made constrain the space of possible inputs to a set that drives the program along a distinct control-flow path. That set may contain only one member if particular values have been given to all input symbols, or an arbitrarily large number of members if some input symbols are only partially constrained. The symbolic execution also provides expressions describing the program outputs for all inputs in this set. If those expressions are recognizable as correct, then the program is correct for all inputs in the set. This represents a significant improvement in what is learned per test, but for most programs the number of possible symbolic tests is still arbitrarily large.

A first symbolic test of SEARCH might be SEARCH (a,lb,ub,x) where lower-case letters are used to denote symbolic inputs. Program variables will always be written using upper-case letters. The interpreter would begin the execution, set FOUND to **false**, and at the **while** find that it could not resolve lb≤ub. If we direct the interpreter to assume this predicate is **true**, then the execution would continue to the first **if** statement where the interpreter would need to know if $x < a((lb+ub)/2)$. If we assumed this was **false**, the interpreter would next need to know if $x > a((lb+ub)/2)$. If we assumed **false**, again FOUND would be set to **true** and the program would halt. At this point the **pc** would contain the simplified conjunction of the constraints we had imposed along the path. Specifically, the **pc** would be $lb \leq ub \wedge x = a((lb+ub)/2)$. Also the output program variables would have values: FOUND = **true** and MID = $(lb+ub)/2$. One can see that, in the case described by the **pc**, the program is correct and particularly that this symbolic test covers an arbitrarily large number of conventional tests. Figure 2 shows the top portion of the symbolic execution tree for SEARCH.

Finite symbolic execution subtrees. In addition to generating single paths of a symbolic execution tree, performing one symbolic test at a time, one could use the symbolic interpreter to automatically generate particular finite subtrees, representing sets of symbolic tests. For example, it could generate all symbolic executions "L" statements in length, and produce a list of the symbolic results. For those paths that are less than "L" statements long, symbolic expressions would be listed indicating the final **pc** and program outputs, all in terms of the symbolic inputs. More interestingly, one can constrain this subtree generation in a direct manner. Assume we were interested in a program's behavior when one of its inputs, X, was positive. By initializing the **pc** to $x \geq 0$ we will obtain the subtree corresponding to executions for which the initial value of X was greater than or equal to zero. These constraints can be combined and refined as one explores the space of possible program executions.

Yet another technique for selecting an execution subtree is to initialize some of the input variables to constants or interrelated symbolic values. For example, if we were to attempt to examine the most general symbolic execution tree for SEARCH by setting each input to a unique symbolic value, as shown in Figure 2, we see that the tree is arbitrarily large. But if the length of the array were fixed to, say, 5 by setting LB to 1 and UB to 5, then there would only be 11 different symbolic executions. A somewhat more general way to accomplish restricting the array length to 5 would be to initiaize LB to the symbolic value n and UB to the functionally related value $n+4$. One could also set LB to lb, UB to ub, and initialize the **pc** to $ub=lb+4$ to accomplish the same result. Table 1 shows the 11 cases that could automatically be produced by an exhaustive symbolic execution of SEARCH$(a, n, n+4, x)$. Examination of this subtree confirms the intuitive argument that given any sorted array of length, say m, and a particular value x to be found: either x is present in the array (m cases), or it lies between two adjacent array values (m-1 cases), or it is less than all of them or greater than all of them (2 cases). With $m = 5$ as in our example, the proper number of cases is then 11.

Table 1. Exhaustive symbolic execution of SEARCH for array length 5.

TEST	PC	FOUND	MID
1	x=a(3)	**true**	3
2	x=a(1) ∧ x<a(3)	**true**	1
3	x<a(1) ∧ x<a(3)	**false**	0
4	x=(2)∧ x>a(1) ∧ x <a(3)	**true**	2
5	x>a(1) ∧ x<a(2) ∧ x<a(3)	**false**	1
6	x>a(1) ∧ x>a(2) ∧ x<a(3)	**false**	2
7	x=a(4) ∧ x>a(3)	**true**	4
8	x>a(3) ∧ x<a(4)	**false**	3
9	x=a(5) ∧ x>a(3) ∧ x>a(4)	**true**	5
10	x>a(3) ∧ x>a(4) ∧ x<a(5)	**false**	4
11	x>a(3) ∧ x>a(4) ∧ x>a(5)	**false**	5

The programmer can examine the symbolic results, deciding if the execution enumerated the proper set of cases and, in each case, deciding if the output are correct as characterized by the final path predicate. His success will depend, in part, on the simplifications that are done in combining constraints, e.g., $(x \leq y \wedge x \geq y)$ reducing to $x=y$, or $(n+n+2)/2$ reducing to $n+1$. Without such simplifications the input constraints and output expressions could be difficult to recognize. However, one must always keep in mind that these simplifications may ignore certain aspects of the programming

language semantics such as overflow or floating point arithmetic. For example, $x+1+1$ may be simplified to $x+2$ for all x including the largest machine-representable integer.

Test case generation. There are numerous papers[3-5,12-14] concerning "test-case generation"—the production of sets of specific input values to be used for testing a program. The goal of early techniques was to produce test cases to execute every statement at least once. Later this was upgraded to include execution of each branch both ways. And more recently the techniques attempt to produce tests to traverse the paths from each setting of a variable to each use of that variable. Meanwhile the expense of generating these test

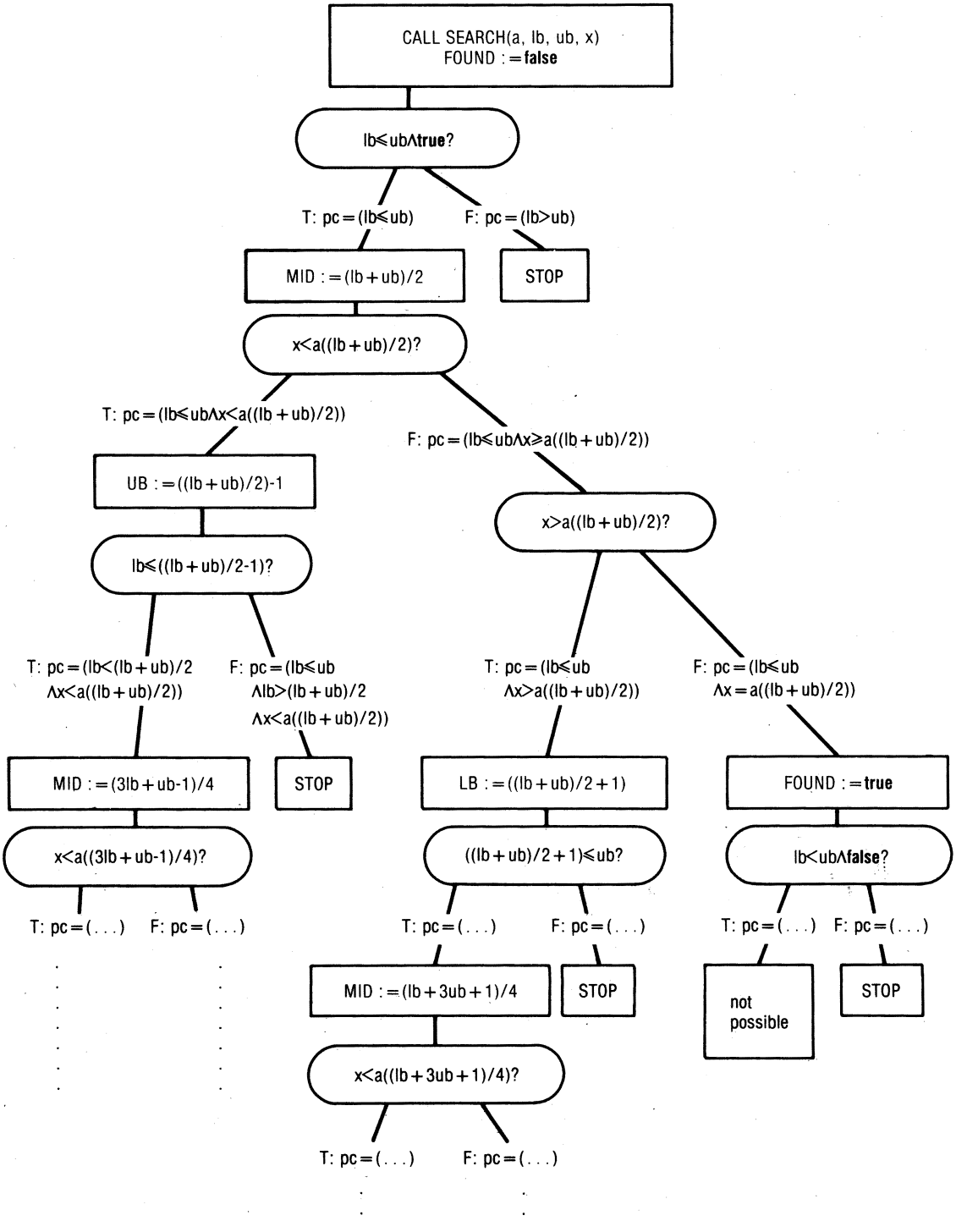


Figure 2. Symbolic execution tree for SEARCH.

cases has been increasing. Since there is no practical algorithm for finding input values to reach a particular point in a program, a wide variety of heuristics are used in all of these test-case generators. The concept of symbolic execution does not solve this problem, but it does illuminate the problem and may promote a better understanding.

While it is difficult to find specific input values that cause execution to follow a particular path, one can always direct a symbolic interpreter to follow a particular path, unless the accumulated path condition is shown to be **false**, indicating an impossible path. Then the remaining problem is to instantiate the **pc** with particular values. As explained above, the **pc** specifies a class of equivalent tests and any feasible solution to the constraints, represented by the **pc**, would be a representative member. Although finding particular values to satisfy a predicate is unsolvable, in general, there are effective algorithms when the constraints are linear or nonlinear in a restricted form. The Select system of SRI[3] and the system of Clarke[4] make use of such algorithms.

It may not be clear why one would want particular values once a symbolic test has been performed. There may be several reasons:

- A symbolic interpreter does embody some definition of the programming language semantics, contains a simplifier for the symbolic expressions generated as values, and applied a theorem prover that tries to show predicates **true** or **false**. As mentioned earlier, the possible shortcomings in the first two and the necessary shortcomings in the latter may cause the semantics of the symbolic interpretation to deviate from that of its defining compiler and run-time support. Overflow and floating point arithmetic are areas where the theorem prover or simplifier do not share the same model as that defined by the language's compiler. Thus, one might want specific values to perform additional conventional tests to check for these deviations.
- Particular values may be necessary to exercise the program in a different environment where performance could be measured.
- Seeing actual outputs may help a programmer to check his mental specification of the program.

Test-case coverage. A problem related to test-case generation is accessing the test-case coverage. Well organized program testers often have a file of different test inputs for a program. When a revision to the program is made, it is run for all of these test cases and the results are checked. One would like to know how good the "coverage" of this test file is for the given program. A general definition of coverage which satisfies everyone seems difficult, but one can be given in terms of the general symbolic execution tree—how many of the paths through the tree have been exercised. Each path usually represents a large set of particular cases. Certainly, if one has the choice of making two tests along the same path or one test on each of two paths, the latter choice should be made for the sake of test coverage.

Suppose (as we show later) that with each particular input test from the test file, one could associate the final symbolic **pc** which characterizes all non-symbolic inputs that follow the same path. For example, the non-symbolic test SEARCH(A, 1, 1, 44) where the array A is of length 1 and has $A(1) = 55$, will follow the same path as any input SEARCH (a, lb, ub, x) that satisfies the **pc**: $lb \leq ub$ & $x<a((lb+ub)/2)$ & $\neg(lb \leq (lb \leq (lb+ub)/2\text{-}1)$. Assume we have a test file consisting of tests, say $t_1, t_2, \ldots, t_n$, and a list of the associated final **pc** for each test, say $p_1, p_2, \ldots, p_n$. One could form the disjunct of the p's, that is, one could form $P = p_1 \mid p_2 \mid \ldots \mid p_n$. If this disjunction ($P$) is identically **true**, then the set of tests "cover" the program. That is, all possible control-flow paths have been executed. If it is not **true**, then its negation ($\neg P$) defines that portion of the input space not tested. By instantiating this negation a new test case can be generated and added to the set. Whether this is a practical approach or not depends on the complexity of the conditional branches in the particular program and, as we have seen before, the difficulty of instantiating such predicates.

This technique can be illustrated using the predicates of Table 1. Suppose we have an array of length 5 containing the values 10, 20, 30, 40, 50 and that we have tested the program for X values of 1, 10, and 15. If we calculate the coverage of these tests as described above, we will find that they correspond respectively to symbolic tests 3, 2, and 5 of Table 1. If we form the disjunction of these predicates, we find the coverage is characterized by:

$$x<a(1) \wedge x<a(3) \vee$$
$$x=a(1) \wedge x<a(3) \vee$$
$$x>a(1) \wedge x<a(2) \wedge x<a(3)$$

With the knowledge that the array is sorted this reduces to:

$$x<a(2).$$

That means that all inputs such that $X<A(2)$ will follow the control paths that have been tested by our three test cases. If we are confident that these paths are correct, then the negation of this predicate tells what part of the input space remains to be tested. For example, we could select $X=100$ for the next test.

A symbolic interpreter can be used to compute the final **pc** corresponding to a particular non-symbolic test in the following simple way. The non-symbolic values for all input variables given by the test are not used directly but are set aside to be used later. Instead, the program is called with symbolic values given to each of its inputs. The execution proceeds until an unresolved branch is encountered. At this point the **pc**, which is in terms of the symbolic inputs, is evaluated with the

non-symbolic input test values substituted for their corresponding symbols. For example, if the test case specified the value 1 for the parameter LB, if the symbolic execution had been done with LB initialized to *lb*, and if the unresolved test simplified to $lb \geq 0$, the symbolic executor would be directed to take the **true** path, ($1 \geq 0 =$ **true**). The values do not enter into the symbolic execution directly, they serve only as an oracle to decide which path the general symbolic execution should take. When the program terminates, the **pc** characterizes those inputs that share the same control flow as the particular test. If the final **pc** were $x=2 \wedge y=12$, it would indicate the test of a very specialized path, while a final **pc** like $x=2 \wedge y>0$ would indicate that this test is a member of a large class of tests all following the same control-flow path.

So far we have only symbolically tested SEARCH for fixed-length arrays. If one could confirm correct test results for symbolic arrays of arbitrary length, the complete program correctness would be established. This leads naturally into considering proof of program correctness as the limiting case of symbolic testing. That problem is discussed next.

Program correctness

One often has a strong intuitive feeling that a program is actually correct after careful testing. Of course, everyone knows that just because a program works properly for some fixed set of inputs, one cannot, generally, conclude anything about inputs not tried. The intuitive feeling is certainly based on something more. The test provided "good coverage." All the "boundary values" were tried. Typical "representatives" of "all the cases" were included. Much of the development of testing methods is an attempt to capture, define, and formalize this intuition. Most of what has been discussed above is an attempt to make some of these notions more precise. Perhaps the strength of our intuition lies in the fact that we have been performing, possibly subconsciously, informal correctness proofs. The particular test runs are devised to experimentally confirm hypotheses that are difficult to check informally in our heads.

Informal induction. The presence of loops in programs means the number of distinct executions is related to the number of their cycles and is therefore, in general, unbounded. The only way a programmer can have any confidence in such a program is through an inductive argument. To verify a program is to provide a formal specification of the program's intent and a formal inductive proof that the program satisfies this specification. In the absence of these formal proofs many experienced programmers make informal inductive arguments asserting their programs' correctness. A symbolic interpreter seems greatly to facilitate the generation of these informal proofs without much of the overhead of a formal verification.

In analyzing the behavior of a loop, one can begin the loop body with symbolic values assigned to all variables affected by the loop. Then after one execution of the loop the new values reflect the changes. In many cases looking at these formulae does aid the formulation of inductive arguments.

Let us consider how we might make an informal inductive argument about the correctness of SEARCH. The symbolic execution tree for SEARCH(a, lb, ub, x) is arbitrarily large. But if we examine the subtree near the root, by limiting executions to some fixed length, we may be able to detect some pattern.

Examining a single loop execution, we find four cases. In two cases the program terminates reporting that x was found or not found in the array. In two other cases the variables *UB* and *LB* are modified and execution continues. It is these later paths that are responsible for the arbitrarily large execution tree. Now we can look at the values of selected program variables at each entry of the loop; these are shown in the table at the bottom of Figure 3. We can see that the interval of the array being searched is always a left or right portion of the previous interval being searched. At this point we might be able to formulate an inductive argument over the interval length:

1. SEARCH is correct for a particular length, e.g., 5.
2. If SEARCH is correct for an interval of length $(ub\text{-}lb)/2$, then it is correct for an interval of length $ub\text{-}lb+1$.
3. Therefore it must be correct for intervals of all length.

Although such inductive arguments may not be as easily made on other examples, they are likely to be similar in nature.

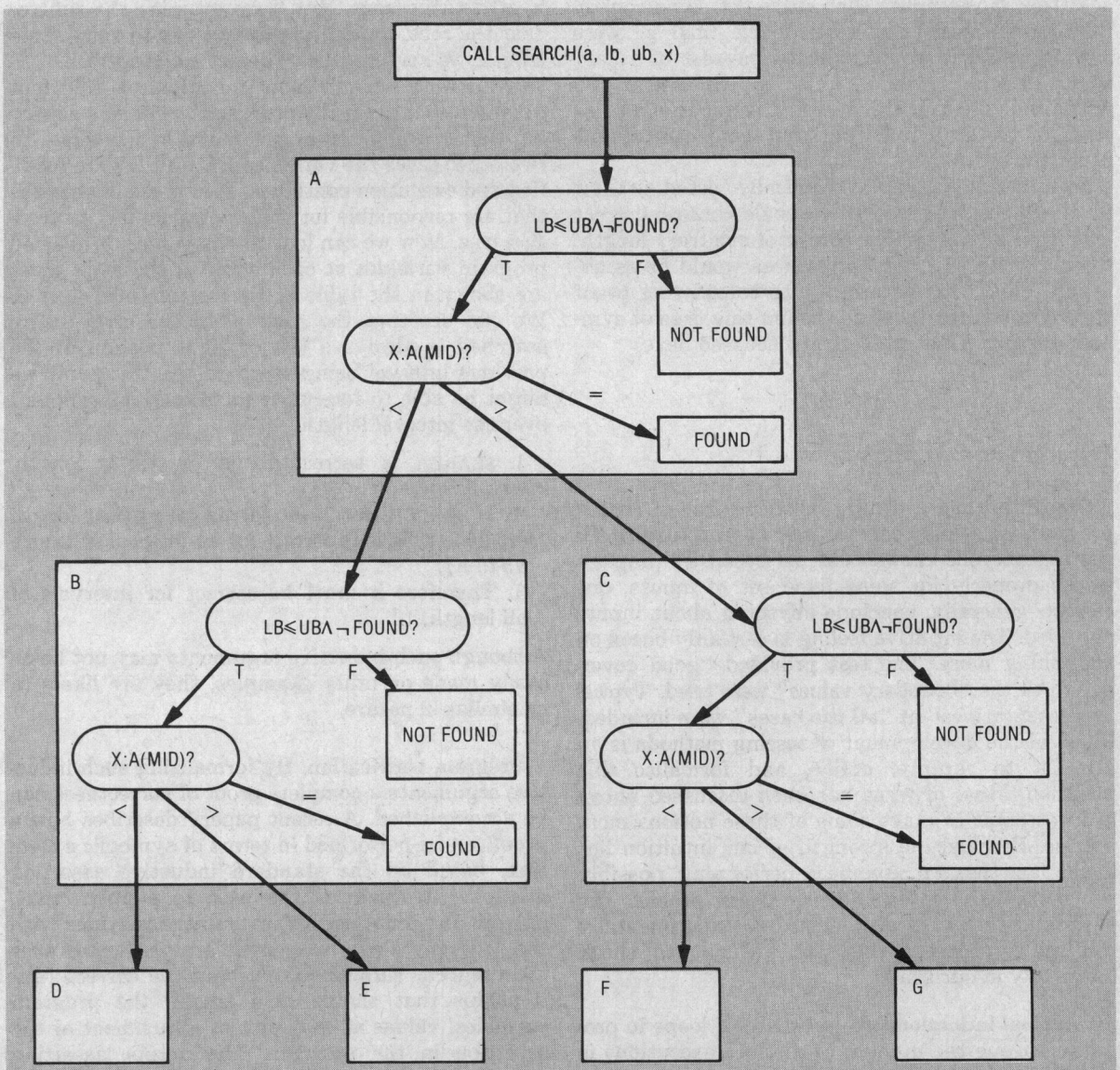
Program verification. By formalizing such inductive arguments a complete proof of correctness can be accomplished. A recent paper[15] describes how a proof can be performed in terms of symbolic execution, based on the standard inductive assertion method.[16] It requires the user to supply input, output, and inductive (or invariant) assertions with the program. These assertions are predicates over the program variables describing the correct relationships that should hold among the program variables' values at the point of attachment of the assertion in the program. The output assertion describes the "correct" relation of the values at the program's termination. The input assertion describes the assumptions required for all legal procedure inputs. The assertions associated with loops provide the inductive assistance needed to characterize the proper relationships of the variables' values within each loop.

The general inductive argument associated with the inductive assertion method reduces the problem of establishing program correctness to a finite set of path proofs of the following form:

• Given a particular finite-length control-flow path beginning at the point of attachment of one assertion, say P, and ending at the point of attachment of another, say Q; and given that for any values of the program variables which satisfy P at the beginning of the path, and cause the execution to follow that path; show that the values, consequently computed and associated with the variables at the end of the path, satisfy Q.

A symbolic interpreter can be used effectively to perform these basic steps in a path correctness proof. For each such path:

1. Set all program variables to unique symbolic values.
2. Evaluate P over these values and set the **pc** to the result.
3. Symbolically execute the path accumulating the evaluated test predicates into the **pc** at each



POINT	LB	UB	INTERVAL
A	lb	ub	ub − lb + 1
B	lb	(lb + ub − 2)/2	(ub − lb)/2
C	(lb + ub + 2)/2	ub	(ub − lb)/2
D	lb	(ub + 3 × lb − 6)/4	(ub − lb − 6)/4
E	(ub + 3 × lb + 2)/4	(lb + ub − 2)/2	(ub − lb − 6)/4
F	(lb + ub + 2)/2	(lb + 3 × ub − 2)/4	(ub − lb − 6)/4
G	(lb + 3 × ub + 6)/4	ub	(ub − lb − 6)/4

Figure 3. Top execution subtree for SEARCH.

possible branch to force the execution to follow that path.

4. At the path end form and attempt to prove the verification condition: **pc** $\supset$ v(Q).

One can see the close match between the description of the path verification problem given above and the steps required of the symbolic interpreter. The complete process is described more carefully by Hantler and King [15] including how the technique can be expanded to include procedure calls.

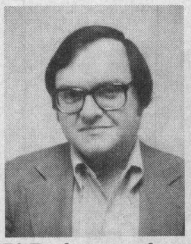
Conclusions

A symbolic interpreter can provide a broad and useful spectrum of services to a programmer attempting to convince himself of his program's correctness. With no information beyond the program, he can perform tests with particular values. In addition, he can perform symbolic tests to determine his program's behavior over a class of executions each following the same control-flow path. If he can supply partial specifications, test results can be checked against them by evaluating them over the current program variables' (symbolic) values and by attempting to establish their truth for the particular input subspace described by the **pc**. A symbolic interpreter can give an indication of the coverage of a set of non-symbolic tests and help in the generation of additional distinct tests. Also, it can be used to analyze the behavior of loops and help in the formulation of inductive arguments necessary for a proof of correctness. Finally, if the inductive arguments can be formalized and loop-invariant assertions added to the program, the interpreter can assist in a complete formal proof of the program's correctness. ■

A symbolic interpreter can help a programmer convince himself of his program's correctness.

References

1. J. C. King, "A New Approach To Program Testing," *Int. Conf. on Reliable Software*, April 1975, pp. 228-233.
2. J. C. King, "Symbolic Execution and Program Testing," *CACM*, Vol. 19, No. 7, July 1976, pp. 385-394.
3. R. S. Boyer, Bernard Elspas, and K. N. Levitt, "SELECT—A Formal System For Testing and Debugging Programs by Symbolic Execution," *Int. Conf. on Reliable Software*, April 1975, pp. 234-245.
4. L. A. Clarke, "A System To Generate Test Data and Symbolically Execute Programs," *IEEE Trans. on Soft. Eng.*, Vol. SE-2, No. 3, September 1976, pp. 215-222.
5. W. E. Howden, "Methodology For The Generation of Program Test Data," *IEEE Trans. on Computers*, Vol. C-24, No. 5, May 1975, pp. 554-560.
6. W. E. Howden, "Symbolic Testing and The DISSECT Symbolic Evaluation System," *IEEE Trans. on Software Engineering*, Vol. SE-3, 1977, pp. 266-278.
7. R. W. Topor, *Interactive Program Verification Using Virtual Programs*, PhD dissertation, Dept. of AI, University of Edinburgh, Scotland, 1975.
8. R. W. Topor and R. M. Burstall, "Verification of Programs By Symbolic Execution Progress Report," unpublished report, Dept. of Mach. Intelligence, University of Edinburgh, Scotland, December 1972.
9. L. P. Deutsch, *An Interactive Program Verifier*, PhD dissertation, Dept. Comp. Sci., University of California, Berkely, May 1973.
10. D. I. Good, *Toward a Man-Machine System For Proving Program Correctness*, PhD dissertation, Comp. Sci. Dept., University of Wisconsin, Madison, June 1970.
11. R. S. Boyer and J. S. Moore, "Proving Theorems About Lisp Functions," *JACM*, Vol. 22, No. 1, January 1975, pp. 48-59.
12. J. C. Huang, "An Approach To Program Testing," *ACM Computing Surveys*, Vol. 7, No. 3, September 1975, pp. 113-128.
13. K. W. Krause, et al, "Optimal Software Test Planning Through Automated Network Analysis," *IEEE Symp. on Comp. Soft. Reliability*, April 1973, pp. 18-22.
14. C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen, "On The Automated Generation of Program Test Data," *IEEE Trans. on Soft. Eng.*, Vol. SE-2, No. 4, December 1976, pp. 293-300.
15. S. L. Hantler and J. C. King, "An Introduction to Proving The Correctness of Programs," *ACM Computing Surveys*, Vol. 8, No. 3, September 1976, pp. 331-353.
16. R. W. Floyd, "Assigning Meanings To Programs," *Proc. Symp. Appl. Math.*, Amer. Math. Soc., Vol. 19, 1967, pp. 19-32.



John A. Darringer, a member of the research staff at IBM's T.J. Watson Research Center, is currently interested in the use of abstraction in the specification and verification of large programs. Before joining IBM in 1972, he was a consultant to the Computer Division of Philips in Apeldoorn, Holland.

Darringer received his BS, MS, and PhD degrees from Carnegie-Mellon University in 1964, 1965, and 1969, respectively.

James C. King is a visiting scholar at the Stanford University Artificial Intelligence Laboratory and the IBM Palo Alto Scientific Center. He is on sabbatical from IBM's T. J. Watson Research Center in Yorktown Heights, N. Y., where he has been a research staff member since 1969. During that time he has also taught courses on an adjunct basis in the Electrical Engineering and Computer Science Department of Columbia University in New York City.

He received the BA and MA degrees in mathematics from Washington State University in 1962 and 1964, respectively, and the PhD degree in computer science from Carnegie-Mellon University in 1969.