Programming Languages
B. Wegbreit Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretively executes programs written in a simple PL/I style programming language. It includes many standard debugging features, the ability to manage and to prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program proving is also included.

Key Words and Phrases: symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation

CR Categories: 4.13, 5.21, 5.24

Author's address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, N.Y. 10598.

1. Introduction

The large-scale production of reliable programs is one of the fundamental requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Recent work on proving the correctness of programs by formal analysis [5] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding etc.

This paper describes a practical approach between these two extremes. From one simple view, it is an enhanced testing technique. Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of *classes* of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases. These results can be checked against the programmer's expectations for correctness either formally or informally.

The class of inputs characterized by each symbolic execution is determined by the dependence of the program's control flow on its inputs. If the control flow of the program is completely independent of the input variables, a single symbolic execution will suffice to check all possible executions of the program. If the control flow of the program is dependent on the inputs, one must resort to a case analysis. Often the set of input

classes needed to exhaust all possible cases is practically infinite, so this is still basically a testing methodology. However, the input classes are determined only by those inputs involved in the control flow, and symbolic testing promises to provide better results more easily than normal testing for most programs.

2. Symbolic Execution

The symbolic execution of a program is described in this section in an ideal sense, and then, in Section 6, a particular practical system which has been built (an approximation to the ideal) is discussed. The term ideal is used for several reasons:

1. The assumption is made that programs deal only with integers and, in fact, only with integers having arbitrary magnitude. Machine register overflows are not considered.
2. The "execution tree" (defined later) resulting from symbolic execution of many (most) programs is infinite.
3. The symbolic execution of IF statements requires theorem proving which, even for modest programming languages, is mechanically impossible.

Nonetheless, the discussion of the ideal does provide a standard against which real computer systems for symbolic execution can be measured.

Each programming language has an execution semantics describing the data objects which program variables may represent, how statements written in the language manipulate data objects, and how control flows through the statements of a program. One can also define an alternative "symbolic execution" semantics for a programming language where the real data objects need not be used but can be represented by arbitrary symbols. Symbolic execution is a natural extension of normal execution, providing the normal computations as a special case. Computational definitions for the basic operators of the language are extended to accept symbolic inputs and produce symbolic formulas as output.

Let us consider a simple programming language. Let the program variables be exclusively of type "signed integer". Include simple assignment statements, IF statements (with THEN and ELSE clauses), GO-TO's to labels, and some means for obtaining inputs (e.g. procedure parameters, global variables, read operations). Restrict the arithmetic expressions to the basic integer operators of addition (+), subtraction (−), and multiplication (×). Restrict the Boolean expressions (used in IF statements) to the simple test of whether an arithmetic expression is non-negative (i.e. {arith.expr.} ≥ 0).

The symbolic execution of programs in this simple language is now described taking the normal execution sementics for granted. The *execution semantics* is changed for symbolic execution, but neither the language syntax nor the individual programs written in the language are changed. The only opportunity to introduce symbolic data objects (symbols representing integers) is as inputs to the program. For simplicity, let us suppose that each time a new input value for the program is required, it is supplied symbolically from the list of symbols $\{\alpha_1, \alpha_2, \alpha_3, \ldots\}$. Program inputs are eventually assigned as values to program variables (e.g. by procedure parameters, global variables, or read statements). Thus, to handle symbolic inputs, we allow the values of variables to be α_i's as well as signed integer constants.

The evaluation rules for arithmetic expressions used in assignment and IF statements must be extended to handle symbolic values. The expressions formed in the usual way by the integers, a set of indeterminate symbols $\{\alpha_1, \alpha_2, \ldots\}$, parentheses, and the operations +, −, and × are the integer polynomials (integer valued, integer coefficients) over those symbols. By allowing program variables to assume integer polynomials over the α_i's as values, the symbolic execution of assignment statements follows naturally. The expression on the right-hand side of the statement is evaluated, possibly substituting polynomial expressions for variables. The result is a polynomial (an integer is the trivial case) which is then assigned as the new value of the variable on the left-hand side of the assignment statement.

The GO-TO's to labels function exactly as in normal executions by unconditionally transferring control from the GO-TO statement to the statement associated with the corresponding label.

The "state" of a program execution usually includes the values of program variables and a statement counter (denoting the statement currently being executed). The definition of the symbolic execution of the IF statement requires that a "path condition" (*pc*) also be included in the execution state. *pc* is a Boolean expression over the symbolic inputs $\{\alpha_i\}$. It never contains program variables, and for our simple language, is a conjoined list of expressions of the form $R \geq 0$ or $\neg(R \geq 0)$, where R is a polynomial over $\{\alpha_i\}$. For example:

$$\{\alpha_1 \geq 0 \wedge \alpha_1 + 2 \times \alpha_2 \geq 0 \wedge \neg(\alpha_3 \geq 0)\}.$$

As will be seen, *pc* is the accumulator of properties which the inputs must satisfy in order for an execution to follow the particular associated path. Each symbolic execution begins with *pc* initialized to *true*. As assumptions about the inputs are made, in order to choose between alternative paths through the program as presented by IF statements, those assumptions are added (conjoined) to *pc*.

The symbolic execution of an IF statement begins in a fashion similar to its normal execution: the evaluation of the associated Boolean expression by replacing variables by their values. Since the values of variables are polynomials over $\{\alpha_i\}$, the condition is an expression of the form: $R \geq 0$, where R is a polynomial. Call such an expression *q*. Using the current path condition (*pc*)

form the two following expressions:

(a) $pc \supset q$
(b) $pc \supset \neg q$.

At most one of these expressions can be *true* (eliminating the trivial case where *pc* is identically *false*). When exactly one expression is *true*, we continue the execution of the IF statement as usual by passing control either to the THEN part, when expression (a) is *true*, or to the ELSE part, when expression (b) is *true*. All normal executions, whose input values satisfy *pc*, would follow the same alternative as this symbolic execution; all would take the THEN alternative ($pc \supset q$) or all would take the ELSE alternative ($pc \supset \neg q$). In this case, the execution of the IF statement is called a "nonforking" execution.

The more interesting case occurs when neither expression (a) nor expression (b) is *true*. In this situation there exists at least one set of inputs to the program which satisfy *pc* and would take the THEN alternative *and* there exists at least one other set of inputs which satisfy *pc* and would lead to the ELSE alternative. Since each alternative is possible in this case, the only complete approach is to explore both control paths. So the symbolic execution is defined to fork into two "parallel" executions: one following the THEN alternative, the other, the ELSE. Both of these executions assume the computation state which existed immediately before execution of the IF statement but proceed independently thereafter. In this case the execution of the IF statement is called a "forking" execution. Note that the forking/nonforking characteristic is associated with a particular execution of an IF statement and not with the statement itself. One execution of a particular IF statement may be forking, while a subsequent execution of the same statement may be nonforking.

Since, in choosing the THEN alternative, the inputs are assumed to satisfy *q* (the evaluated IF statement Boolean), this information is recorded in *pc* by doing the assignment $pc \leftarrow pc \wedge q$. Similarly choosing the ELSE alternative leads to $pc \leftarrow pc \wedge \neg q$. *pc* is called the "path condition" because it is the accumulation of conditions which determines a unique control flow path through the program. Each forking execution of an IF statement contributes a condition over the input symbols which is determined by the particular choice of path. *pc* remains unchanged for nonforking executions of IF statements, since no new assumptions are made or needed. *pc* can never become *false* since its initial value is *true* and the only operation performed on *pc* is an assignment of the form:

$pc \leftarrow pc \wedge r$ (where r is either q or $\neg q$).

but only in the case when $(pc \wedge r)$ is satisfiable ($pc \wedge r \equiv \neg(pc \supset \neg r)$, which is satisfiable if and only if $(pc \supset \neg r)$ is *not* a theorem).

Fig. 1. Procedure SUM.

```
1 SUM: PROCEDURE (A,B,C);
2        X ← A + B;
3        Y ← B + C;
4        Z ← X + Y − B;
5        RETURN (Z);
6 END;
```

3. Examples

Consider the simple program shown in Figure 1. It is written in a PL/I-style syntax and computes the sum of three values. With integer inputs of 1, 3, and 5 the conventional execution of this program, as shown in Figure 2, computes the output 9. The symbolic execution shown in detail in Figure 3 has established that for any three integers, say α_1, α_2, α_3, the program will calculate their sum, $\alpha_1 + \alpha_2 + \alpha_3$.

Now consider the somewhat more complicated example shown in Figure 4 for raising an integer X to the power Y. With the symbols α_1 and α_2 supplied as input for X and Y the symbolic execution would proceed as shown in Figure 5.

4. Symbolic Execution Tree

One can generate an "execution tree" characterizing the execution paths followed during the symbolic execution of a procedure. Associate a node with each statement executed (labeled with the statement number) and with each transition between statements a directed arc connecting the associated nodes. For each forking IF statement execution, the associated node has two arcs leaving the node which are labeled "T" and "F" for the *true* (THEN) and *false* (ELSE) parts, respectively. Also associate the complete current execution state, i.e. variable values, statement counter, and *pc* with each node. The execution tree for POWER (α_1,α_2) (Figures 4 and 5) is given in Figure 6.

The trees formed in this way from a symbolic execution have the following interesting properties.

1. For each terminal leaf in the tree (corresponding to a completed execution path) there does exist a particular nonsymbolic input to the program which, when executed in the normal fashion, will trace the same path (list of statements executed). This is equivalent to saying that *pc* never becomes identically *false*. A brief argument establishing that was made at the end of Section 2.

2. *pc*'s associated with any two terminal leaves are distinct (i.e. $\neg(pc_1 \wedge pc_2)$). The two paths from the common root of the execution tree leading to any two terminal leaves have a unique forking node where the two paths diverge. At that forking node some q was added to one *pc* while $\neg q$ was added to the other. Since neither *pc* becomes inconsistent (*false*) they must maintain this difference.

The significance of these comments is emphasized in

Fig. 2. Execution of SUM(1, 3, 5). A dash represents unchanged values, i.e. the same values as those given in the line above; the question mark represents undefined (uninitialized) values.

After statement	X	Y	Z	A	B	C
			Values would be			
1	?	?	?	1	3	5
2	4	—	—	—	—	—
3	—	8	—	—	—	—
4	—	—	9	—	—	—
5			(Returns 9)			

Fig. 3. Symbolic execution of SUM (α_1, α_2, α_3). Path condition is abbreviated *pc*.

After statement	X	Y	Z	A	B	C	*pc*
			Values would be				
1	?	?	?	α_1	α_2	α_3	*true*
2	$\alpha_1+\alpha_2$	—	—	—	—	—	—
3	—	$\alpha_2+\alpha_3$	—	—	—	—	—
4	—	—	$\alpha_1+\alpha_2+\alpha_3$	—	—	—	—
5			(Returns $\alpha_1+\alpha_2+\alpha_3$)				

the example of Figure 7. Note: DO statements are introduced for simplicity. They can be expanded easily into IF/GOTO loops to conform with the preceding notation. The execution tree for TWOLOOPS (of Figure 7) is shown in Figure 8. Even though statement 4 of the second loop has the same syntactic branching potential as statement 2 of the first loop, the assumptions made while forking at statement 2 are preserved (in *pc*) and are used in statement 4, which then avoids generation of a fork. These symbolic execution trees are similar to the execution trees defined by Paterson for program schemata in [13]. Paterson's trees are also discussed in [12].

5. Commutativity

The symbolic execution defined above for this simple integer language satisfies an interesting commutative property. The operation of instantiating the symbols $\{\alpha_i\}$ with specific integers, say $\{j_i\}$, and the operation of executing the program are interchangeable. That is, if one conventionally executes a program with a specific set of integers $\{j_i\}$ as inputs (instantiation of α_i's by j_i's first, followed by execution), the results will be the same as executing the program symbolically and then instantiating the symbolic results (assigning j's to α's). The meaning of "instantiating the symbolic results" is first, for each terminal leaf in the execution tree, substitute j's for α's in all program variable values and in *pc*. Then the "results" are the values for the terminal node whose *pc* becomes *true*. This commutativity can be diagrammed as shown in Figure 9, where **P** represents the program, **E(P(X))** is the result of executing program **P** on inputs **X**, and **K** is a set of specific integer inputs.

Of course, this commutativity relationship is why symbolic execution is of interest. The symbolic executions capture exactly the same effects as conventional executions. Symbolic execution is not merely an arbitrary alternative execution definition but a natural extension of the conventional definition. As in the relationship between arithmetic and algebra the specific (arithmetic) computations dictated by the program operators are generalized and "delayed" using the appropriate algebraic formulas.

6. An Interactive Symbolic Executor—EFFIGY

The author and his colleagues have been developing an interactive symbolic execution system called EFFIGY. Work began on this system in early 1973 and is still in progress. The system offers a spectrum of services to the user. Basic debugging and testing facilities are provided for symbolic program execution with normal program execution provided as a special case. An "exhaustive" test manager is available for systematically exploring the alternatives presented in the symbolic execution tree. The system can automatically check test case results against output assertions if they are supplied. Finally, the system offers a program verifier which uses symbolic execution and user supplied assertions to generate the verification conditions, generally following the ideas of Deutsch [4]. This paper focuses on symbolic execution as an independent concept and on its use for program testing.

The language for which symbolic execution is possible in EFFIGY, has been enhanced with each new version of the system and now is written in a PL/I style syntax and includes:

1. External procedures with PL/I parameter passing conventions.
2. Integer valued variables (only) (FIXED BINARY (31, 0)) and single dimensional arrays of integer values. These variables may be declared as STATIC or AUTOMATIC and may have INITIAL attributes.
3. Assignment statements, IF statements (with THEN and ELSE clauses), compound statements using DO . . . END, and GO-TO statements.
4. Iterative DO and DO WHILE statements.
5. Elementary READ and WRITE statements.
6. The arithmetic, relational, and logical operators are exclusively: $+$, $-$, $*$, $/$, $**$, ABS, MOD (remainder); $\geq$, $\leq$, $>$, $<$, $=$, $\neq$; & (and), | (or), $\neg$ (not), $\supset$ (implies).

A full complement of interactive debugging facilities

is also available, including:

1. *Tracing.* The user can request to see the statement number, the source statement, the computational results, or any combination of these for any or all statements within procedures as they are executed.
2. *Breakpoints.* The user can insert "breakpoints" before or after any statement or between any statement pair. At these points execution is interrupted and control passes to the user's terminal. The user can then examine the state of the execution, set variables, and resume execution.
3. *State saving.* As a user explores the various paths of his program he may wish to save the state of execution to later return and explore alternative paths. "SAVE" and "RESTORE" are provided for this purpose.

Special features and commands basic to *symbolic* execution are also included in the system. The user may define arbitrary identifiers to be symbolic program inputs (the α's from before) by enclosing them in double quotes (") and using them in place of specific integer constants. For example, the user could invoke a procedure SUM (from Figure 1) for testing by:

CALL SUM (1, 3, 5);	Normal execution over integers.
CALL SUM ("A", "B", "C");	Symbolic execution using the symbols A, B, and C.
CALL SUM ("A", 3, 5);	A combination.

The definition of symbolic execution given earlier dictates that for *forking* (unresolvable) IF statement executions the execution proceeds in parallel on both alternate paths (the THEN alternative *and* the ELSE alternative) thus generating a *complete* execution tree. For most programs this is an infinite process. The simplest, and perhaps most general, solution to this problem is to let the user interactively choose the single alternative path to be taken at any one time. This basic facility is provided in EFFIGY. By use of the SAVE and RESTORE commands, mentioned earlier, the user can save the state of the execution and return later to explore alternate paths. This allows the user to "walk" the execution tree starting at the root in any way he chooses.

Whenever the system encounters a forking execution of an IF statement (both alternatives being possible) it notifies the user and allows him to choose. He may:

Fig. 4. Procedure POWER.

```
1 POWER: PROCEDURE(X, Y);
2         Z ← 1;
3         J ← 1;
4 LAB:    IF Y ≥ J THEN
5         DO; Z ← Z * X;
6             J ← J + 1;
7             GO TO LAB; END;
8         RETURN (Z);
9 END;
```

1. Type "go true" and the system follows the THEN alternative changing *pc* accordingly,
2. Type "go false" and the system follows the ELSE alternative changing *pc* accordingly, or
3. Type "assume (*P*); go".

In the third form P is a predicate which is first *evaluated* using the current values of program variables and then added (conjoined) to the path condition (*pc*). The "go" in this case directs the system to re-execute the IF statement using the modified *pc*.

For example, suppose the program variable X has the value a, *pc* has the value $a > 0$, and the IF statement being executed has the form:

IF $X > 5$ THEN S_1 ELSE S_2;

When evaluated $X > 5$ becomes $a > 5$. Using *pc* the choice of path is unresolvable by the system since neither:

(a) $a > 0 \supset (a > 5)$ nor
(b) $a > 0 \supset \neg(a > 5)$.

The system invites the user to choose. If he types "go true" *pc* is updated to $a > 5$ (formed from $a > 0$ & $a > 5$) and statement S_1 is executed next. If he types "go false" *pc* is updated to $a > 0$ & $\neg(a > 5)$ and statement S_2 is executed next. If he types "assume ("a" > 10); go" *pc* becomes $a > 10$ ($a > 0$ & $a > 10$) and the IF statement is re-executed. This time

$a > 10 \supset a > 5$

is *true* and the execution proceeds to statement S_1. The user could have gotten the same result by typing "assume ($X > 10$); go" since $X > 10$ gets evaluated to $a > 10$ at the first step.

When the system asks for the user's choice at a forking IF statement execution, it is, in fact, in the same state as if the user had stopped the execution with a breakpoint *before* the IF statement. As such, before the user types "go true", "go false", or "go", he may examine program variables, set breakpoints, adjust the trace settings, etc. Even though the definition of the ASSUME statement is most easily motivated by the use described above, its execution is independent of the IF statement. As such, ASSUME statements can be entered at any breakpoint as well as included in procedures and will have the following effects:

1. Evaluate the ASSUME's Boolean expression.
2. Conjoin the result to the current *pc* (if consistent).

An ASSUME statement could be used at the beginning of a procedure, for example, to keep the system from considering inputs on which it was not designed to operate.

The discussion in Section 2 was restricted to integer polynomials for pedagogical simplicity. Canonical forms for polynomials are well known, and so is the fact that the set of polynomials is closed under addition and multiplication. The generalization from arithmetic to algebra is well understood in this case. The EFFIGY system is implemented to deal with the more general class

of expressions described above. Of course, one would like to have an EFFIGY system which operates on a common programming language like PL/I. That imposes an extremely ambitious burden on the formula manipulation and simplification component of the system. That component of the EFFIGY system was, in fact, borrowed from a program verifier previously built by the author [8]. The capabilities and limitations of EFFIGY's formula manipulation are inherited directly from that earlier system which is discussed in detail in [9].

7. A Further Example

A brief description of the basic EFFIGY system is also given in [10]. That paper contains a script from an actual EFFIGY session as an Appendix which may be of interest to some readers. An explanation of how the program of Figure 10, SEARCH, could be checked out on EFFIGY follows. These steps can be, and were, actually performed on the EFFIGY system.

The program SEARCH was written to perform a binary search for an argument X in an array A of elements stored in ascending order. The search is confined to the array elements with subscripts from L up to and including U. If a match is found, the subscript value of the array element matching X is returned in J and FOUND is set to 1. Otherwise, FOUND is set to 0 and J is set to the value such that A(J) < X < A(J+1). The symbolic execution tree for this program is infinite since the initial value of (U−L) may be arbitrarily large.

The first test of SEARCH might take the form:

```
CALL SEARCH (A, 1, 5, "X", FOUND, J).
```

Assume the elements of the array A, A(1), A(2), . . . , A(5), have been set to symbolic values "A(1)", "A(2)", . . . , "A(5)", respectively. The constants 1, 5, and "X" are input arguments and FOUND and J are integer variables which will return the results from SEARCH. Symbolic execution will proceed until statement number 7, at which point the user is asked whether or not "X" = "A(3)". If in response to the system's query the user types "save; go true", the current execution state will be saved as state 1 and the execution will run to completion determining that *pc* = ("X" = "A(3)"), FOUND = 1, and J = 3. Now by typing "restore 1; go false" the user may examine the other possibility when "X" ≠ "A(3)". Continuing in this manner the user may explore the finite subtree determined by inputs 1 and 5, and in this case find eleven terminal leaves:

pc	FOUND	J
X=A (3)	1	3
X=A (1) & X<A (3)	1	1
X<A (1) & X<A (3)	0	0
X=A (2) & X>A (1) & X<A (3)	1	2
X>A (1) & X<A (2) & X<A (3)	0	1
X>A (1) & X>A (2) & X<A (3)	0	2
X=A (4) & X>A (3)	1	4
X>A (3) & X<A (4)	0	3
X=A (5) & X>A (3) & X>A (4)	1	5
X>A (3) & X>A (4) & X<A (5)	0	4
X>A (3) & X>A (4) & X>A (5)	0	5

The user could also cause the system to generate these eleven outputs automatically by use of the TEST facility in EFFIGY:

```
TEST (200) SEARCH (A, 1, 5, "A", FOUND, J)
```

The 200 is used to limit the exhaustive search of the symbolic execution tree to those paths traversing less than 200 statement executions. In this case the limit is unneeded since the tree is finite and small. This test provides some evidence that the program will successfully find any element of the array.

One may next try:

```
CALL SEARCH (A, "N", "N" + 4, "X", FOUND, J)
```

which is a generalization of the previous test. As before, this execution, if pursued exhaustively, will produce eleven terminal cases matching the previous ones where 1 will be replaced by "N", 2 will be replaced by "N" + 1, etc. E.g., the second case listed above would become:

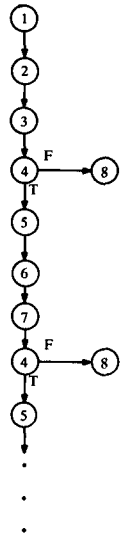
X=A(N) & X<A (N+2) 1 N

Fig. 5. Symbolic execution of POWER (α_1, α_2).

After statement	J	X	Y	Z	*pc*
		Values would be			
1	?	α_1	α_2	?	*true*
2	—	—	—	1	—
3	1	—	—	—	—
4 execution in detail:					
(a) evaluate Y≥J getting $\alpha_2 \geq 1$.					
(b) use path condition and check:					
(i) *true* ⊃ $\alpha_2 \geq 1$					
(ii) *true* ⊃ $\neg(\alpha_2 \geq 1$					
(c) neither are theorems, so fork.					
Case $\neg(\alpha_2 \geq 1)$:					
4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$
8 this case completed. (returns 1 when $\alpha_2 < 1$.)					
Case $\alpha_2 \geq 1$:					
4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	—	—	—	α_1	—
6	2	—	—	—	—
7	—	—	—	—	—
4 execution in detail:					
(a) evaluate Y≥J, getting $\alpha_2 \geq 2$					
(b) use *pc*:					
(i) $\alpha_2 \geq 1 \supset \alpha_2 \geq 2$					
(ii) $\alpha_2 \geq 1 \supset \neg(\alpha_2 \geq 2)$					
(c) neither *true*, so fork.					
Case $\neg(\alpha_2 \geq 2)$:					
4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \neg(\alpha_2 \geq 2)$ (or simply $\alpha_2 = 1$)
8 this case completed. (returns α_1 when $\alpha_2 = 1$.)					
Case $\alpha_2 \geq 2$:					
4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \alpha_2 \geq 2$ (or simply $\alpha_2 \geq 2$)
⋮	⋮				

In this example the symbolic execution will continue indefinitely.

Fig. 6. Execution tree for POWER(α_1, α_2).



Case *pc* is ($\alpha_2 < 1$): RETURNS 1

Case *pc* is ($\alpha_2 = 1$): RETURNS α_1

{RETURNS α_1^n when *pc* is ($\alpha_2 = n$)}

This test gives us the additional assurance that no special properties of the numbers 1, 2, . . . , 5 (such as being powers of 2) affected our previous results.

Another similar but more specialized test is:

CALL SEARCH (A, "N", "N", "X", FOUND, J)

which, when exhaustively explored, creates a three leaf tree:

pc	FOUND	J
X=A (N)	1	N
X<A (N)	0	N−1
X>A (N)	0	N

The two calls:

CALL SEARCH (A, 1, "N", "X", FOUND, J) and
CALL SEARCH (A, "L", "U", "X", FOUND, J)

are the most general tests and can be used to check that "bounds averaging" and the "shrinking" of the bounds range proceed properly. Even though both calls lead to infinite symbolic execution trees, interesting results like the following (derived from the first call) are possible:

pc	FOUND	J
X=A((N+1)/2) & N>0	1	(N+1)/2
X=A(((N+1)/2)/2) & N>0 & X<A((N+1)/2) & (N+1)/2>1	1	((N+1)/2)/2
X>A((N+1)/2) & N>0 & X=A(((N+1)/2+N+1)/2) & (N+1)/2<N	1	((N+1)/2+N+1)/2

Note that while performing all of these tests one never supplies nonsymbolic values to the array A or to the argument X. The properties of these values which are important in the program (e.g. "X" $\leq$ "A(N)") are uncovered during the symbolic execution. All of the testing is done from the original program alone—no correctness predicates are required. As discussed further in Section 8, input, output, inductive, and checking predicates can be used in conjunction with symbolic execution of programs quite effectively for testing and correctness proofs, but they are not *required* for symbolic testing. As with normal execution, when testing a program using symbolic execution, one must use care in choosing "interesting" test cases and in deciding when enough is enough.

8. Program Correctness, Proofs, and Symbolic Execution

To prove the correctness of a program using a method presented by Floyd [6], the programmer supplies an "input predicate" and an "output predicate" with the program. The predicates define the "correct" behavior of the program, the program being correct if for all inputs which satisfy the input predicate the results produced by the program, if any, satisfy the output predicate. Floyd presents a method for checking the consistency of the program and its I/O predicates, and thus for proving its correctness.

One of the steps in Floyd's proof method, the generation of verification conditions, can be done quite simply by executing *program paths* symbolically. Deutsch independently developed the notion of symbolic execution in exploiting this proof technique in his interactive program verifier [4].

Perhaps the simplest way to explain this technique is using the three ancillary language statements: ASSUME, PROVE, and ASSERT used to associate predicates with the program. All three of these statements have a Boolean formula supplied as part of the statement in parentheses after the statement name, e.g. ASSERT(X>0). The free variables of these formulas are assumed to be program variables. ASSUME(B) was defined previously in Section 6. When executed, B is evaluated using the current values of program variables, and the resulting value is conjoined to the path condition (i.e. $pc \leftarrow pc \wedge$ value (B)).

The PROVE(B) statement is executed by forming the expression:

$pc \supset$ value(B)

and attempting to establish that it is a theorem and displaying *true* or *false* accordingly. In the way PROVE is used below, these expressions, in fact, will be the Floyd verification conditions. The ASSERT statement will be used later as either an ASSUME or a PROVE statement depending on the context.

Besides the initial and final predicates which serve to define the program's correctness, Floyd's method requires associating additional "inductive predicates"

with various points in the program. This is usually done such that at least one predicate is within every loop in the program, but it can be done in any way so long as the control flow paths, given by the program segments between predicates, are of finite length. The inductive predicates allow a general inductive argument to be made (see [6] or [7]) which reduces the proof of correctness of the program to the proof of correctness of a finite set of finite length paths.

Assume that input, output, and inductive predicates are associated with a program by placing ASSERT statements into the program at the appropriate points (the initial predicate is an ASSERT statement placed as the first statement in the program, etc.) We now have a fixed set of paths through the program, each of which begins with an ASSERT statement and ends with an ASSERT statement and each must be proved correct. That is, one must show, using *any* set of variable values which satisfy the predicate at the beginning of the path, that the values resulting from execution along the path must satisfy the predicate at the end.

One can prove the correctness of each path by executing it symbolically as follows:

1. Change the ASSERT at the beginning of the path to an ASSUME; change the ASSERT at the end of the path to a PROVE.
2. Initialize the path condition to *true* and all the program variables to distinct symbols say, α_1, α_2, ...
3. Execute the path symbolically. Whenever an unresolved IF statement execution is encountered, do a "go true" or "go false", whichever will make the execution follow the designated path.
4. If the PROVE at the end of the path displays *true*, the path is correct, otherwise it is not.

Provided that the symbolic execution satisfies the commutative property discussed in Section 5, it is straightforward to see that this is a valid proof method. The program proof is carried out in terms of the values of program variables at the beginning of the path, to which we give symbolic names. The path condition accumulates assumptions made about those initial values. Execution of the first ASSUME records, in the path condition, the required assumptions about those initial values. ("One must show, using *any* set of variable values which satisfy the predicate at the beginning of the path that the values resulting from execution along the path must satisfy the predicate at the end.") The formulas computed as a result of assignment statements record the updated program variables' values as a function of the values at the start of the path. The action at unresolved IF statement executions records in the path condition the additional assumptions, again over the path's initial values, required for any execution to follow this particular path.

Finally the execution of the PROVE at the end of the path sets up a theorem candidate (verification condition) which expresses the question: assuming the be-

Fig. 7. Procedure TWOLOOPS.

```
1    TWOLOOPS: PROCEDURE (N);
2                  DO J=1 TO N;
3                    (body of statements) END;
4                  DO K=1 TO N;
5                    (body of statements) END;
6                  END;
```

Fig. 8. Execution tree for procedure TWOLOOPS.

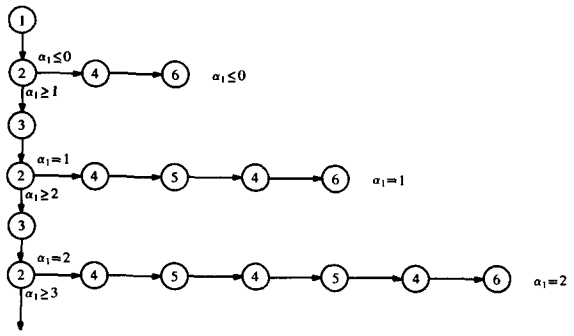


Fig. 9. Commutativity diagram.

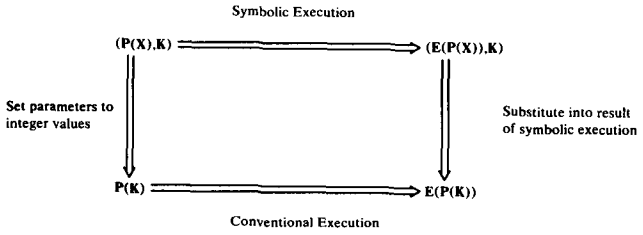


Fig. 10. Procedure SEARCH.

```
1    SEARCH:
          PROC(A, L, U, X, FOUND, J);
2         DCL A(*) INTEGER;
3         DCL (L, U, X, FOUND, J) INTEGER;
4         FOUND = 0;
5         DO WHILE (L¬>U & FOUND=0);
6                J = (L+U)/2;
7                IF X = A(J)  THEN FOUND = 1
8                       ELSE IF X < A(J)
9                            THEN U = J-1
10                           ELSE L = J+1;
11        END;
12        IF FOUND = 0    THEN J = L-1;

13   END;
```

ginning predicate was satisfied and we followed this path (recorded in *pc*), do the current (path end) values of the program variables "satisfy the predicate at the end?"

Given a symbolic execution system such as EFFIGY, it is conceptually straightforward to also have it attempt correctness proofs by the addition of a PROVE statement and managerial controller to enumerate paths and force path choices at unresolved IF statement executions. We have done this and are using it as a tool for pursuing research into correctness proof techniques.

In fact the notions of correctness proof and symbolic execution are closely linked in concept as well as in tools required to perform them. Suppose one has associated Floyd input/output predicates with a program by placing the input predicate as an ASSUME statement at the beginning of the program and the output predicate as a PROVE statement at the end. With the definition of the ASSUME statement given here the initial ASSUME restricts the subsequent analysis, be it symbolic execution or a correctness proof, to only those values satisfying the initial predicate. Adopting the definition for PROVE statements necessary for correctness proofs is also a very useful one for program testing by symbolic execution. In program testing, symbolic or not, one must examine the output produced and judge its correctness. If one can formalize the correctness criteria in terms of an output predicate and place that at the end of the program in a PROVE statement, the symbolic executor will then also perform the proper checking of test results.

One other comment relates to the definition of the PROVE statement. Several languages do provide support for the normal run-time checking of predicates placed in the program with, say, ASSERT statements (e.g. Algol W[14]). Such statements are easily implemented by considering

ASSERT (*B*)

as shorthand for

IF $\neg B$ THEN (some error alert such as SIGNAL ERROR).

Since this is a normal IF statement, one can consider its symbolic execution. If the program is correct the executions of the IF statement should always be resolvable to the *false* path. By the definition for symbolic execution of the IF statement, that happens only if

$pc \supset \text{value}(B)$.

Note that this is exactly what PROVE attempts to prove!

For any program whose symbolic execution tree is finite and for which the correctness criteria has been made explicit with input/output predicates, the exhaustive symbolic execution and the proof of correctness done as above are exactly the same process. Since no inductive predicates are necessary, the set of paths requiring proof are those from the input predicate to the output predicate which are the paths described by the finite execution tree.

For programs with infinite execution trees, the symbolic testing cannot be exhaustive and no absolute proof of correctness can be established. An obvious way to enhance symbolic execution to provide a correctness proof in all cases is to consider some form of induction over the infinite parts of the execution tree. This is in fact, exactly what Floyd's proof of correctness method does. The inductive predicates placed within the loops provide the inductive aid necessary for symbolic execution to "execute" over the infinite branches of the execution tree. Another similar yet different approach has recently been taken by Topor and Burstall [15] to provide the inductive assistance needed to "execute" over the infinite branches of the tree and provide a correctness proof.

There is one notable difference, however, in the capability required of the predicate manipulation facility for correctness proofs, from that required for symbolic testing. If one is strictly confined to symbolic execution without the use of any user introduced predicates, *pc* and the expressions requiring proof are syntactically *and semantically* determined by the programming language. However, the predicate semantics in correctness proofs derive from the *problem* area of the program and not the programming language.

It is this difference that convinces us that symbolic execution for *testing programs* is a more exploitable technique in the short term than the more general one of program verification.

9. Practical Issues

Many of the troublesome issues arising in program proving systems also occur in symbolic execution. For example, a problem common to both areas is finding a practical way to deal with variable storage-referencing. For example, the array notation A(I) references a different particular element of the array A depending on the *value* of I. When the value of I is a symbolic expression, the particular element being referenced is a function of the initial program inputs. In many cases, even when all the information known about those inputs, as held in *pc*, is examined, the particular reference may be inherently ambiguous.

At least two approaches to this particular problem are possible, although neither is very satisfactory.

(1) An exhaustive case analysis can be undertaken, as is done in the analogous unresolved IF statement execution.

(2) The ambiguity can be left unresolved but preserved by storing conditional values for variables. For example, the *value* of A(I) (assuming that the *value* of I is i), might be "if i=1 then x else if i=3 then y else if i=j+1 then z . . .".

The conflict between discrete aspects of computer

arithmetic and the continuous nature of real numbers with infinite precision also is an issue in symbolic execution. To assist in the required theorem proving and to make the formulas more readable, they should be simplified as much as possible. However, many powerful simplifications are prohibited if one insists that the resulting expressions produce the same values as the originals when computer evaluated, because the commutative property discussed in Section 5 would be violated. Also, there is a gap between the truth of a predicate and the ability of an automatic theorem prover to establish its truth. In particular, an automatic system may conclude that some IF statement execution is unresolved when, in fact, it is not. The system would then follow a path that could never be followed in a real execution. The previous claim that *pc* never becomes *false*, which was made at the end of Section 2 can no longer be made. Of course, even for rather simple programming languages the theorem proving required to perform a symbolic execution of programs in that language becomes impossible, in theory. That is, it is impossible to build a theorem prover that will decide the truth or falsity for that class of expressions.

10. Conclusion

A notion of symbolically executing a program has been presented which is closely related to the normal notion of program execution. It offers the advantage that one symbolic execution may represent a large, usually infinite, class of normal executions. This can be used to great advantage in program testing and debugging. The EFFIGY system built by the author and his colleagues was also described. It embodies symbolic execution in a general purpose interactive debugging system. Additional facilities based on the basic symbolic execution capability include an "exhaustive" test manager and a program verifier. Interactive debugging/testing systems are powerful, useful tools for program development. The addition of features based on symbolic execution is a significant improvement and the normal facilities are still available as a special case.

Symbolic execution is also useful in other forms of program analysis, including test case generation [1, 2] and program optimization [3, 16]. A symbolic execution system, such as EFFIGY, offers a natural growth from today's systems; an evolutionary approach for achieving the program validation system of tomorrow is available. Having a running system which can be incrementally enhanced also provides valuable user experience and support. While practical use of the EFFIGY system is still quite limited, considerable insight into the general notion of symbolic execution and its varied applications has been gained during its construction.

Acknowledgments. My colleagues at IBM Research who collaborated with me in this work are S.M. Chase, A.C. Chibib, J.A. Darringer, and S.L. Hantler. They have all contributed significantly to the ideas presented here and to the design and implementation of the EFFIGY system. M.W. Blasgen, S.L. Hantler, and J.R. Buchanan have each provided particularly thorough constructive critical comments on earlier drafts of this report which are greatly appreciated. We also appreciate the support and encouragement received from D.P. Rozenberg, P.C. Goldberg, and P.S. Dauber.

Received December 1974; revised January 1975

References

1. Boyer, R.S., Elspas, B., Levitt, K.N. SELECT—A formal system for testing and debugging programs by symbolic execution. 1975 Int. Conf. on Reliable Software, April 1975, pp. 234–245.
2. Clarke, L. A system to generate test data and symbolically execute programs. Rep. No. CU-CS-060-75, Dep. of Computer Sci., U. of Colorado, Feb. 1975.
3. Darlington, J. A semantic approach to automatic program improvement. Ph.D. Th., U. of Edinburgh, 1972.
4. Deutsch, L.P. An interactive program verifier. Ph.D. Th., Dep. of Computer Sci., U. of California, Berkeley, May 1973.
5. Elspas, B., et al. An assessment of techniques for proving program correctness. *Computing Surveys 4*, 2 (June 1972), 97–147.
6. Floyd, R.W. Assigning meanings to programs, *Proc. Symp. Appl. Math.* Vol. 19, Amer. Math. Soc., Provincetown, R.I., 1967, pp. 19–32.
7. King, J.C. Proving programs to be correct. *IEEE Trans. on Comp. C-20*, 11 (Nov. 1971), 1331–1336.
8. King, J.C. A program verifier. Proc. IFIP Cong. 71, North-Holland, Amsterdam, 1971, pp. 235–249.
9. King, J.C. and Floyd, R.W. An interpretation oriented theorem prover over integers, *J. Computer Syst. Sci. 6,* 4(Aug. 1972), 305–323.
10. King, J.C. A new approach to program testing. 1975 Int. Conf. on Reliable Software, April 1975, pp. 228–233.
11. Krause, K.W., et al. Optimal software test planning through Automated network analysis. IEEE Symp. on Computer Software Reliability, April 1973, pp. 18–22.
12. Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974, Ch. 4.
13. Paterson, M.S. Equivalence problems in a model of computation. Ph.D. Th., U. of Cambridge, England, Aug. 1967. Also published as A.I. Tech. Memo No. 1 (Memo No. 211), MIT, Nov. 1970.
14. Sites, R.L. Algol W Reference Manual. Rep. CS-230 (Clearinghouse No. PB 203601), Computer Sci. Dep., Stanford U., Feb. 1972.
15. Topor, R.W., and Burstall, R.M. Verification of programs by symbolic execution—progress report. Unpublished report, Dep. of Machine Intelligence, U. of Edinburgh, Scotland, Dec. 1972.
16. Urschler, G. Complete redundant expression elimination in flow diagrams. Rep. RC4965, IBM Research, Yorktown Heights, N.Y., Aug. 1974.