An Introduction to Proving the Correctness of Programs

SIDNEY L. HANTLER

and

JAMES C. KING

Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

This paper explains, in an introductory fashion, the method of specifying the correct behavior of a program by the use of input/output assertions and describes one method for showing that the program is correct with respect to those assertions. An initial assertion characterizes conditions expected to be true upon entry to the program and a final assertion characterizes conditions expected to be true upon exit from the program. When a program contains no branches, a technique known as symbolic execution can be used to show that the truth of the initial assertion upon entry guarantees the truth of the final assertion upon exit. More generally, for a program with branches one can define a symbolic execution tree. If there is an upper bound on the number of times each loop in such a program may be executed, a proof of correctness can be given by a simple traversal of the (finite) symbolic execution tree.

However, for most programs, no fixed bound on the number of times each loop is executed exists and the corresponding symbolic execution trees are infinite. In order to prove the correctness of such programs, a more general assertion structure must be provided. The symbolic execution tree of such programs must be traversed inductively rather than explicitly. This leads naturally to the use of additional assertions which are called "inductive assertions."

Keywords and Phrases: Program correctness, program proving, program verification, proving correctness of programs, symbolic execution, symbolic interpretation

CR Categories: 1.3, 4.13, 5.21, 5.24

INTRODUCTION

Interest in verifying that computer programs behave as they were intended to behave has existed since the advent of modern electronic computers. As the size and complexity of computer programs have increased, so has the importance of assuring that these programs behave reliably. Naturally, attention has been focused on the problem of specifying precisely what constitutes reliable behavior and on developing a thorough method for checking that a program will always meet those specifications.

It is the intent of this paper to give a tutorial presentation of one approach for showing that a program meets its specification. The basic approach of using "correctness assertions" and the particular form of induction used are due to Floyd

CONTENTS

[7]. The strategy explained in this paper for composing a proof is similar to methods developed by Deutsch [5] and Topor [21]. The presentation is informal; no theorems are stated or proved. For the person who understands what it means to execute a program and who understands simple algebraic and mathematical concepts, the ideas presented here are quite straightforward. Rigorous presentations of similar material are available elsewhere [6, 13, 18].

We begin by defining a very simple programming language. Though simple, the language contains the important basic features of commonly used programming languages. All examples in the paper are written in this language, and in Section 2, the concept of correctness for programs written in the language is developed. The symbolic execution of programs is introduced in Section 3 as the basic tool for building correctness proofs. In Section 4 the proof method is further developed for programs with looping structures. In order to show the generality of the technique, it is extended to handle subroutines and functions in Section 5. Finally, in Section 6, there is a discussion of the state of the art of program verification and its computer automation, with an emphasis on research into problems that remain unsolved.

1. PROGRAMMING LANGUAGE AND SEMANTICS

In this section we describe a simple programming language of a PL/I style, suitable for introducing the notion of correctness. In order to facilitate the exposition and minimize the technical details, we choose a particularly simple language, with only basic statement types and simple arithmetic expressions.

Procedures are declared by statements of the form:

name: **PROCEDURE** $(p_1, p_2, p_3, \ldots, p_n)$;
⟨statement-list⟩
END;

where *name* is the procedure name and p_1, p_2, p_3, . . . , p_n are procedure parameters. As usual, the body of the procedure consists of a list of statements placed between the **PROCEDURE** and the corresponding **END**. There are two types of procedures: 1) functions, which are referenced from within arithmetic expressions; and 2) subroutines, which are invoked explicitly by a **CALL** statement. This distinction is discussed in more detail later.

Program variables are integer valued and are declared by the **DECLARE** statement. The statement

DECLARE $variable_1$, $variable_2$, . . . , $variable_n$ **INTEGER**;

creates integer valued variables named $variable_1$, $variable_2$, . . . , $variable_n$. These variables are known only within the procedure in which they are declared and a "new" generation is created on each procedure call (cf., PL/I *automatic* variables). Arithmetic operations on the values of

program variables yield new values. Values of program variables and integer constants may be added (+), multiplied (×), and subtracted (−). The basic assignment statement has the form:

variable ← ⟨expression⟩;

where *variable* is a declared program variable and ⟨expression⟩ is an arithmetic expression in declared variables, integer constants, and function names applied to the appropriate number of arguments.

A function name occurring in the right-hand side of an assignment statement causes the function procedure associated with that name to be invoked. Thus if *name* is a procedure with a single parameter which returns the value 7 when invoked with the argument 3, the result of executing

variable ← (2 × *name*(3)) + 4;

is that the value of *variable* becomes 18.

Statements may be grouped together into a compound statement by means of the **DO**; ⟨statement-list⟩ **END**; construct. When enclosed by the **DO-END** pair, the list of statements is treated as if it were a single statement.

Boolean primitives are constructed from Boolean constants *true* and *false* and arithmetic expressions (as on the right side of assignment statements) connected by the relational operators: less than (<), greater than (>), equal (=), and their complements (≥, ≤, ≠). Boolean expressions (denoted by ⟨Boolean⟩ below) are constructed using the Boolean primitives connected by: and (&), or (|), implies (→), and not (¬). The value of a Boolean expression is either *true* or *false*.

The binary conditional is of the form:

IF ⟨Boolean⟩ **THEN** *statement*$_1$ **ELSE** *statement*$_2$

where *statement*$_1$ and *statement*$_2$ are statements or compound statements. As usual, either *statement*$_1$ or *statement*$_2$ is executed, depending on the truth value of the ⟨Boolean⟩.

The iterative statement is of the form:

DO WHILE ⟨Boolean⟩; ⟨statement-list⟩ **END**;

When control reaches the **DO WHILE** statement, if the value of the ⟨Boolean⟩ is *true*, the statement list is executed and control is returned to the **DO WHILE** statement. If the ⟨Boolean⟩ is *false*, control passes immediately to the statement following the **END** statement.

As mentioned above, function procedures are invoked by reference to the procedure name within an arithmetic expression. Parameters are passed exactly as described in the following discussion about subroutine procedures. In addition to the changes that may be effected through the parameters, function procedures return one special value to be used in the invoking expression evaluation. Subroutines are invoked and parameters are passed by means of the **CALL** statement. For example, the statement

CALL *name* $(a_1, a_2, a_3, \ldots, a_n)$;

causes the subroutine named *name* to be invoked and the names of the formal parameters $p_1, \ldots, p_n$ to be associated with the names of the actual arguments $a_1, \ldots, a_n$, respectively. Parameter passing follows the PL/I "by reference" convention, i.e. references to the actual arguments are passed to the subroutine (or function). When an argument is an expression, its value is stored in a temporary storage location, a reference to which is passed to the subroutine.

For convenience in referring to the initial values of the parameters of a procedure from within correctness assertions, the initial values of the parameters are stored in special procedure variables which are denoted by primed symbols. For example, after execution of the **CALL** statement above, the values of $a_1, \ldots, a_n$ upon entry to *name* are preserved in the variables $p_1', \ldots, p_n'$, respectively.

Return from an invoked procedure is

```
1  ABSOLUTE:
     PROCEDURE(X);
3      DECLARE X, Y INTEGER;
4      IF X < 0
5        THEN Y ← −X;
6        ELSE Y ← X;
8      RETURN (Y);
9    END;
```

FIGURE 1. Function procedure *ABSOLUTE*.

achieved by means of the **RETURN** statement, which may appear at any place in a procedure. Each procedure has exactly one **RETURN** statement. A **RETURN** statement is of the form:

RETURN;

in a subroutine, and

RETURN (⟨expression⟩);

in a function. The latter statement returns the value of the ⟨expression⟩ as the value of the function.

2. CORRECTNESS OF PROGRAMS

Having defined a simple programming language in Section 1, we now discuss the meaning of "correctness" of procedures written in that language. We will provide a method for formalizing the intended behavior of a procedure. In particular, constraints on the inputs to a procedure and expected relations between inputs and outputs will be expressed as assertions over the program variables. An *input assertion* is a statement of the form:

ASSUME (⟨Boolean⟩);

and usually appears immediately after the **PROCEDURE** statement. For example, the input assertion

ASSUME ($p_1 > 0$);

asserts that the value of the parameter p_1 is assumed to be positive on procedure entry. An *output assertion* is a statement of the form:

PROVE (⟨Boolean⟩);

and usually appears immediately before the **RETURN** statement of a procedure. For example, the output assertion

PROVE ($(X = Y') \ \& \ (Y = X')$);

indicates that the values of the variables X and Y have been interchanged. Note that this is the relationship between inputs and outputs which would be satisfied by a correct interchange procedure.

Naturally, the notion of "correctness" of a procedure should reflect this relation among the input assertion, output assertion, and procedure body. *A procedure is said to be correct (with respect to its input and output assertions) if the truth of its input assertion upon procedure entry insures the truth of its output assertion upon procedure exit.* Notice that the question of program termination is suppressed in this definition. Intuitively, a procedure is correct provided that it behaves as expected when it terminates. This is often called "partial correctness," with the term "correctness" or "total correctness" reserved for procedures that are partially correct and terminate for all inputs.

A simple procedure is shown in Figure 1. The function *ABSOLUTE* is intended to return the absolute value of its parameter. Inasmuch as no assumptions need be made about the input parameter to *ABSOLUTE*, the input assertion should be **ASSUME** (*true*). The output assertion must specify that, when the **RETURN** statement is executed, the value of the procedure variable Y is the absolute value of the initial value of the parameter X. Thus, an ap-

propriate output assertion (among others), describing what this procedure does, is **PROVE** $((Y = X' \mid Y = -X') \ \& \ Y \geq 0)$. We will see later, that it is often important to specify what a procedure does not do, as well as what a procedure does. A more complete output assertion, specifying that the value of X is unchanged by the procedure *ABSOLUTE*, is **PROVE** $((Y = X' \mid Y = -X') \ \& \ Y \geq 0 \ \& \ X = X')$.

The procedure, with correctness assertions, would then be as shown in Figure 2. In this simple example, it is quite clear that the procedure is correct. In the next section we discuss a formal method of proving that procedures (with input and output assertions) are correct.

3. SYMBOLIC EXECUTION AND SYMBOLIC EXECUTION TREES

A proof of correctness for a program is a proof over *all* program inputs. Certainly such a proof cannot, in general, be made using any finite (small) collection of specific inputs, but must be made with statements about *all inputs*. One can use a standard mathematical technique of inventing symbols to represent arbitrary program inputs, and then attempt a proof involving those symbols. If no special properties of the symbols, other than those expected to hold for all inputs, are necessary for the proof, then the proof is valid for *each* specific input. If special properties of the symbols must be assumed in order to construct a proof, then an exhaustive case analysis can be performed, providing a set of proofs, one for each case, which collectively give a complete proof.

Let us naively attempt to apply this strategy to devise a correctness proof for the simple program *ABSOLUTE* of Figure 2. A typical invocation of *ABSOLUTE* can be represented by using a symbolic argument, say α: *ABSOLUTE*(α). We proceed to execute the program using the symbol α as the input value of X. The **ASSUME** statement execution contributes nothing since its argument is *true*, which places no constraints on the input α. The execution of the **IF** statement is more interesting. Here one must determine if the value of X is negative; that is, if $\alpha < 0$. If α stands for the integer 3 the answer is no, but if α is -3 the answer is yes. To answer this question some assumption about the value of α must be made, and a case analysis is required:

Case 1: Assume $\alpha < 0$. In this case the **IF** test would produce *true* and execution would proceed into the **THEN** clause. Here Y becomes the negative of the value of X, (i.e., $-\alpha$). Arriving at the **PROVE** statement, one must show that, in this case, the present values satisfy $((Y = X' \mid Y = -X') \ \& \ Y \geq 0 \ \& \ X = X')$. Since $Y = -\alpha$ and $X = X' = \alpha$, this becomes

$$(-\alpha = \alpha \mid -\alpha = -\alpha) \ \& \ -\alpha \geq 0 \ \& \ \alpha = \alpha$$

which simplifies to $-\alpha \geq 0$ or more simply

```
1  ABSOLUTE:
     PROCEDURE (X);
2      ASSUME (true);
3      DECLARE X, Y INTEGER;
4      IF X < 0
5        THEN Y ← −X;
6        ELSE Y ← X;
7      PROVE ((Y = X' | Y = −X') & Y ≥ 0 & X = X');
8      RETURN (Y);
9    END;
```

FIGURE 2. Procedure *ABSOLUTE* with correctness assertions.

$\alpha \leq 0$. Establishing the truth of the **PROVE** statement then reduces to showing $\alpha \leq 0$. But we have assumed $\alpha < 0$, so the proof is trivial. In the case $\alpha < 0$, the program is correct.

Case 2: Assume $\alpha \geq 0$. In this case the **IF** test would produce *false* and execution would proceed into the **ELSE** clause. Here Y becomes the value of X or α. Arriving at the **PROVE** statement, one must show that $((Y = X' \mid Y = -X') \ \& \ Y \geq 0 \ \& \ X = X')$ is *true*, when $Y = \alpha$, $X = X' = \alpha$. That is, to show that

$$(\alpha = \alpha \mid \alpha = -\alpha) \ \& \ \alpha \geq 0 \ \& \ \alpha = \alpha,$$

or simply $\alpha \geq 0$, is *true*. Again the proof is trivial since $\alpha \geq 0$ was assumed.

By the nature of the **IF** statement these two cases are exhaustive (either $\alpha < 0$ or $\alpha \geq 0$) and both yield correct results. Therefore the program is correct.

Several points about this example should be made. The assumptions used in the case analysis resulted from an unresolved execution of the **IF** statement. The assumptions were exactly the *evaluated* **IF** test and its negation and were Boolean valued expressions strictly over the input α. These assumptions were needed as hypotheses to establish the truth of the **PROVE** statement in each case.

Symbolic Execution

In this section we attempt to explain the basic "symbolic execution" technique used informally in the preceding example, more carefully and more completely. Within the scope of the programming language used here, consider the consequences of changing the underlying computation facilities of the language implementation (the program execution mechanism) from doing arithmetic operations over integers to doing algebraic operations over symbolic expressions. For example, suppose that the variables X and Y have the symbols α and β as their respective values. As a result of executing the statement $X \leftarrow Y + X$ the value of X becomes the formula $\alpha + \beta$. Executing $Y \leftarrow 3 \times X - Y$ next, would symbolically calculate the formula $3 \times \alpha + 2 \times \beta$ as the new value of Y.

Executing a program on a symbol manipulating machine one might hope to obtain algebraic formulas over the input symbols as the values of the output variables. Then, checking these results against the output assertion, one could establish the correctness, or incorrectness, of the program. As even the extremely elementary example, *ABSOLUTE*, shown above, demonstrates, this is not quite so simple. That example requires a case analysis, since Boolean expressions involving symbols often do not simplify to *true* or *false*. For example, the truth of $\alpha \geq 0$ is not determined without some information about α. However, symbolic execution does provide a complete way to establish program correctness when augmented by such case analyses and by a general inductive technique. Reconsider the definition of program execution given in Section 1, but assume that programs receive symbols or symbolic expressions as input and are executed on a machine capable of performing algebra. At procedure invocation (**CALL** or function reference), the transfer of control and the association of arguments to parameters work the same as before. Similarly the meaning of the **RETURN** statement is unchanged.

The first construct where something more interesting occurs is the assignment statement. The usual execution first replaces all variables in the right-side expression by their values, then performs the indicated arithmetic and assigns the resulting value as a new value of the left-side variable. The symbolic execution performs the algebraic equivalent. The variables in the right-side expression are replaced by their values (parenthesized to maintain the proper scope of operators). Since the values of variables are formulas, the indicated arithmetic operations cannot be done numerically but are simply represented symbolically, as in

algebra. The resulting symbolic expression becomes the new value of the left-side variable.

When the arithmetic expression involves function calls the situation is more complex and is discussed in a later section dealing explicitly with procedure calls. There is also the issue of whether or not algebraic simplification should be performed on formulas resulting from the substitution of values for variables in the right-side expression. If no simplification is done, the formulas accurately characterize the exact computations that would have taken place had the inputs been numbers. In fact, those computations can be done later according to the formulas, getting the same results even with respect to overflow and other machine anomalies.

However, no simplification implies that the formulas may become quite unwieldy. As we will see shortly, theorem proving over these expressions is required, and the difficulty is increased if the formulas are very complex. Since the objective of this paper is to present the basic ideas of proving correctness of programs as simply as possible, this difficult question will not be addressed. When convenient in examples, the formulas will be simplified. In theory, the basic approach is valid whether or not the formulas are simplified. In the extreme, one must choose between very difficult theorem proving and specification writing (in the case of no simplification), and results that, when simplified, may not accurately apply in all cases to an actual computer execution.

The symbolic execution of conditional branching statements also parallels their normal execution but with additional complexity. Consider first the **IF** statement. Its symbolic execution begins by replacing all variables in its Boolean expression by their parenthesized values. The resulting expression may be equivalent to *true*, *false*, or some Boolean expression over the symbolic program inputs. The last situation may result in a case analysis as it did in the *ABSOLUTE* example. Whenever the Boolean result is neither *true* nor *false*, there is at least one numeric program input for which the result is *false* and at least one other for which it is *true*. The execution cannot proceed into either the **THEN** clause or the **ELSE** clause and be valid for all inputs. Thus, the case analysis is required.

Recall from the example that the assumptions which determine the cases are needed later to establish the truth of the **PROVE** predicate. The assumptions are also needed to avoid considering impossible subcases that may arise at subsequent conditional statement executions. For example, consider the execution of the two successive **IF** statements:

IF $X < 0$ **THEN** $Y \leftarrow 88$;
IF $X = 3$ **THEN** $Y \leftarrow 99$;

with the value of $X = \alpha$. There are four syntactic paths through these two statements, but only three are semantically possible. The impossible subcase can be detected if the conditions on α necessary to execute the choices of the first statement (i.e., $\alpha < 0$, $\alpha \geq 0$) are remembered and used to determine consistent choices on the second statement. These observations lead to the notion of a "path condition," abbreviated pc. It is a part of the symbolic execution-state and takes as its value the conditions over the program's symbolic inputs that determine each case, subcase, sub-subcase, etc. The pc is initialized to *true* at the beginning of a symbolic execution and is updated each time a new case is considered.

The complete description of the symbolic execution of an **IF** statement of the form:

IF ⟨Boolean⟩ **THEN** *statement*$_1$ **ELSE** *statement*$_2$

is as follows:

1) Evaluate the ⟨Boolean⟩ obtaining a value, B, over the symbolic inputs.

2) Now decide if subcases for B and $\neg B$ should be formed. If pc $\rightarrow B$, new subcases are unnecessary since enough assumptions

have already been made (recorded in the pc) to determine that $statement_1$ would always be executed next. The symbolic execution proceeds directly to $statement_1$. Similarly if pc $\rightarrow \neg B$, the symbolic execution proceeds directly to $statement_2$. If neither (pc $\rightarrow B$) nor (pc $\rightarrow \neg B$), new subcases for B and $\neg B$ are required as described in steps 3 and 4, respectively.

3) Establish a subcase assuming B. Update the pc with new conditions B by replacing it by (pc_{old} & B), where pc_{old} is the most recent value of the pc (i.e., do the assignment pc $\leftarrow$ pc & B). In this case, the symbolic execution proceeds to $statement_1$ with the revised pc.

4) Establish a subcase assuming $\neg B$. Update the pc with new conditions $\neg B$ by replacing it by (pc_{old} & $\neg B$). In this case, the symbolic execution proceeds to $statement_2$ with this revised pc.

The case of an **IF** statement execution in which the evaluated Boolean reduces directly to *true* or *false* falls out at step 2. Since the pc is never allowed to be *false* (an impossible path), then (pc $\rightarrow$ *true*) is *true* and (pc $\rightarrow$ *false*) is *false* for any pc.

One can also define the symbolic execution of the **ASSUME** and **PROVE** statements which specify the program's correctness as follows:

ASSUME (⟨Boolean⟩):

1) Evaluate the ⟨Boolean⟩ by substituting parenthesized values for variables. Call the result B.

2) Update the pc to the value (pc_{old} & B).

This has the effect of confining the subsequent symbolic execution to the case where the ⟨Boolean⟩ is *true*, which is the intention of the input assertion.

PROVE (⟨Boolean⟩):

1) Evaluate the ⟨Boolean⟩ by substituting parenthesized values for variables. Call the result B.

2) If (pc $\rightarrow B$) print "verified" otherwise print "not verified."

This statement prints "verified" or "not verified" depending on whether or not the program variable's values satisfy the output assertions in this case. The conditions defining *this case* are given by the pc.

The complete symbolic execution of a program like *ABSOLUTE* of Figure 2 can be compactly represented by a "symbolic execution tree." The tree for that example is shown in Figure 3. It is similar to a program flowchart, with each statement execution being represented by a node, and a transfer of control between statement executions by an arc. The nodes are labeled with the program statement numbers, and the arcs leaving statements are labeled by the changes to the execution state, if any, caused by the execution of the preceding statement. Of course, a conditional statement execution node will have more than one arc leaving it when the choice of successor statement remains unresolved. Nodes for nonexecutable statements (e.g., **DECLARE**) are omitted from the trees shown here to conserve space. Since the tree of Figure 3 covers all possible executions of the program *ABSOLUTE*, in each case printing "verified," *ABSOLUTE* is correct.

We have yet to discuss the symbolic execution of **DO WHILE** statements. Without them in our language there is no means for program looping, and non-looping programs always have finite symbolic execution trees. As in the proof of the *ABSOLUTE* procedure, symbolic execution provides a convenient way to prove the correctness of procedures with finite symbolic execution trees. Such programs are correct provided that "verified" appears at each leaf of their symbolic execution trees.

However, infinite symbolic execution trees may occur when their corresponding procedures contain the iterative **DO WHILE** statement of the form

DO WHILE ⟨Boolean⟩; ⟨statement-list⟩ **END**;

Its symbolic execution follows naturally from the symbolic execution of the **IF** state-

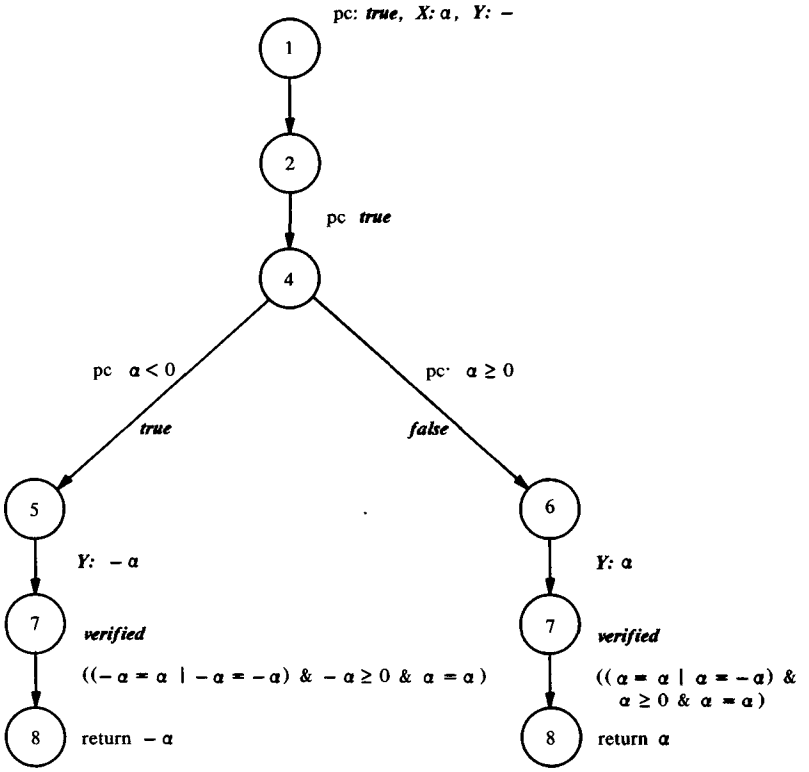

FIGURE 3. Symbolic execution tree for procedure *ABSOLUTE*.

```
1   GCD:
      PROCEDURE (M, N);
2       ASSUME (M > 0 & N > 0);
3       DECLARE M, N, A, B INTEGER;
4       A ← M;
5       B ← N;
6       DO WHILE (A ≠ B);
8         IF A > B
9           THEN A ← A − B;
10          ELSE B ← B − A;
11      END;
12      PROVE (A = (M, N));
13      RETURN (A);
14    END;
```

FIGURE 4. Procedure *GCD* with correctness assertions.

ment. The decision to execute the statement list, go on to the statement following the **END**, or develop those choices as alternative subcases is determined by examining the Boolean expression as was done for **IF** statements.

An example of an infinite symbolic execution tree is shown in Figure 5 for the procedure of Figure 4. The procedure of Figure 4 computes the greatest common divisor of its positive inputs M and N. The procedure's correctness is specified using the standard

mathematical notation where (M, N) stands for the greatest common divisor of M and N. For example, $(3, 12) = 3$, $(20, 15) = 5$, and $(4, 4) = 4$. Let a and b be integers. The greatest common divisor can be characterized by three axioms:

$$(a, a) = a \quad \text{if } a > 0,$$
$$(a, b) = (b, a),$$
$$(a, b) = (a + b, b).$$

Note that the infinite portion of the tree, as shown in Figure 5, is caused by the infinite sequence of unique conditions involving the symbolic inputs.

A program which has an infinite symbolic execution tree may have no particular input which causes an infinite program execution. The symbolic execution tree is infinite because there is always yet another, different, execution which requires more statement executions. Of course, a program which has a nonterminating execution has an infinite symbolic execution tree.

How can the method presented above be applied to programs which generate infinite symbolic execution trees? A general answer to this question is provided by using an inductive technique to "traverse" the infinite paths. This is discussed at length in Section 4. Otherwise one can reduce the problem to the one already discussed by restricting attention to finite subtrees of infinite symbolic execution trees. Recall that symbolic execution actually furnishes a proof of correctness for procedures with finite symbolic execution trees.

We illustrate this point by considering variants of the procedure *GCD* of Figure 4. Suppose, for instance, that we replace the initial **ASSUME** statement of that procedure by **ASSUME**(*false*). The resulting procedure not only has a finite symbolic execution tree (in fact, an empty tree), but it is also guaranteed to be correct. Naturally, the empty subtree of an infinite symbolic execution tree is an extreme and uninteresting subtree to study. A better choice of subtree results from a more subtle restriction of the initial assertion of the program.

Suppose that the initial **ASSUME** statement were modified to **ASSUME** $(M > 0 \ \& \ N > 0 \ \& \ M = C \times N \ \& \ C \leq 1000)$. We would then be restricting attention to that finite subtree of the *GCD* procedure corresponding to the case in which one of the variables is a small multiple of the other. Without inductive assistance of any kind, symbolic execution can provide a proof of correctness of this modified procedure.

Inasmuch as our principal interest is in the original *GCD* procedure rather than the modified procedure, it is perhaps better to think of the consideration of finite subtrees of an infinite symbolic execution tree as a form of testing. The finite subtrees represent the test cases of interest. Notice that testing using symbolic execution differs from more traditional testing techniques in at least two respects. First, ordinary testing covers at most a finite number of specific inputs, while testing by symbolic execution usually covers an infinite number of specific inputs. Second, when correctness assertions are supplied in procedures, symbolic testing provides a proof of correctness for the test cases being considered, rather than merely providing output values for each test case considered.

As a testing technique, symbolic execution appears to be an extremely promising new tool. It is the topic of a recent PhD thesis by Clarke [4], and Boyer et al. [1] have explored the generation of test cases using symbolic execution. The authors and their colleagues have developed a prototype symbolic execution system called EFFIGY [16] which includes features for program testing as well as for program proving.

4. INFINITE SYMBOLIC EXECUTION TREES AND INDUCTION

Programs contain a finite number of statements. Since the nodes on an infinite sym-

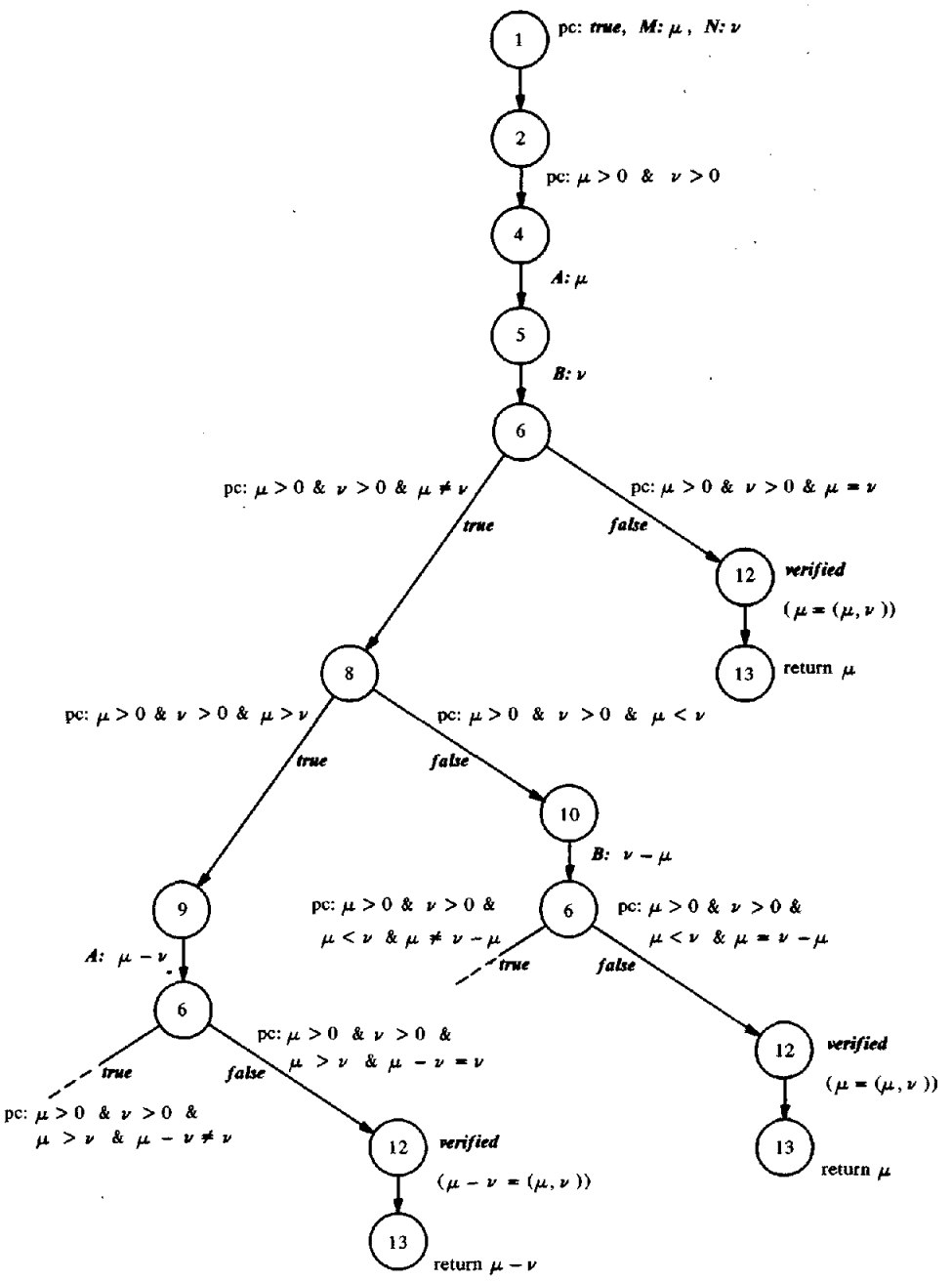

FIGURE 5. Symbolic execution tree for procedure *GCD*.

bolic execution tree are labeled by program statements, some statement labels must occur an infinite number of times. Thus, the infinite portions of the symbolic execution tree are generated by looping in the program. In the case of our programming language the sole loop construct is the **DO WHILE** statement.

Each loop traversal can be isolated by placing a "cut" (mark) at least once within

every loop. The induction to be described is valid even if a loop is cut more than once, so it is trivially possible to cut all loops by placing a cut between every two program statements. Generally one cuts each loop just once. In our language, the cuts can be made by placing a mark between each **DO WHILE** and its statement body.

For simplicity in the subsequent discussion, consider that a cut has also been placed immediately after the **PROCEDURE** statement. Then imagine a symbolic execution of the program which begins at one of the cuts. Beginning at such an arbitrary point in a program, the "inputs" are, in fact, represented by the program state, so consider *all* program variables initialized to unique symbolic values, and the pc initialized to *true*. A symbolic execution, from this point on, is representative of all cases for which execution reaches the cut, independent of the values of the program variables; the new unique symbolic values represent all cases. The symbolic execution stops whenever any subsequent cut or the final program **RETURN** is encountered. Since each program loop has been cut, this symbolic execution will terminate in all cases and have a finite symbolic execution tree.

Call each such symbolic execution starting at a cut a *cut (symbolic) execution* and the corresponding tree a *cut (symbolic execution) tree*. If one were to place an **ASSUME** statement at a cut C and **PROVE** statements at each cut which terminates C's cut tree (the **RETURN** already has a **PROVE** just preceding it), the proof of correctness (with respect to those input/output assertions) for the cut execution of C can be discussed. Since the cut tree is finite, a proof as described in the previous section can always be attempted. If it succeeds, any execution which begins at the cut with values satisfying the cut input assertion is guaranteed to reach another cut and the program values at that point will satisfy the associated output assertion.

The proof of correctness of the entire procedure can be constructed from proofs of the cut executions. What is needed are the input/output assertions hypothesized for each cut tree and an explicit argument for the composition of the overall proof from the pieces. One appropriate assertion associated with each cut makes both possible. The word "appropriate" is used because these cut assertions (more commonly called "inductive assertions" or "inductive predicates") correspond to the inductive hypothesis in the usual proof by mathematical induction and are often quite difficult to discover.

```
1   GCD:
      PROCEDURE (M, N);
2   cut2---  ASSUME (M > 0 & N > 0);
3            DECLARE M, N, A, B INTEGER;
4            A ← M;
5            B ← N;
6            DO WHILE (A ≠ B);
7   cut7---    ASSERT ((A, B) = (M, N) & A ≠ B);
8              IF A > B
9                THEN A ← A − B;
10               ELSE B ← B − A;
11           END;
12           PROVE (A = (M, N));
13  return---RETURN (A);
14       END;
```

FIGURE 6. Procedure *GCD* with inductive assertion.

Suppose that by some "inductive genius" appropriate assertions are placed at each cut, except the first one, by use of a new statement of the form **ASSERT** (⟨Boolean⟩). Figure 6 shows the cuts and the inductive assertion for the *GCD* procedure of Figure 4. Two definitions for the symbolic execution of the **ASSERT** statement are supplied depending upon the context in which it is encountered. Refine the definition of cut execution such that the **ASSERT** statement is encountered as the first statement (just after the variables have all been set to unique symbols). (In this case, assume the actual cut-mark is placed just *above* the **ASSERT** statement.) When executing the **ASSERT** statement in this context, it is treated exactly as if it were an **ASSUME** statement; it supplies a cut execution input assertion. When each cut that terminates a cut execution is encountered, execute the associated **ASSERT** statement as if it were a **PROVE** statement. (In this context, assume the actual cut-mark is placed just *below* the **ASSERT** statement.) Note that one **ASSERT** statement can be treated as both an **ASSUME** statement and as a **PROVE** statement depending on the context.

Special cases at the beginning (**PROCEDURE**) and at the end (**RETURN**) of the program are obvious but must be mentioned. The initial cut after the **PROCEDURE** statement is never encountered as a terminating cut and is followed immediately by the **ASSUME** statement for the overall program, so no **ASSERT** statement is needed. The input cut assertion, for paths starting at the initial cut, is provided by executing the program's **ASSUME** statement. Whenever a cut execution is terminated by the program **RETURN** statement, the program's original **PROVE** statement will have just been executed; so here too no **ASSERT** statement is needed. The result of the program's **PROVE** serves as the result for the cut execution.

The following claim for this proof of correctness method is now easy to establish using an inductive argument. *If inductive assertions can be placed at each cut (except the first) by means of* **ASSERT** *statements such that the cut executions for all cuts are correct with respect to those assertions, then the program is correct. If such assertions do not exist, then the program is not correct.*

The proof of a cut execution establishes that for *any* set of values of the program variables that satisfy the cut input assertions (including those which result from an actual execution of the procedure to this point), the execution ultimately arrives at a subsequent cut and the associated output assertion is satisfied by the resulting values of the program variables. But since only *one* assertion has been associated with each cut, it is both the output assertion for all cut executions arriving at the cut and the input assertion for the cut execution leaving the cut. Any values which satisfy it as an output assertion also satisfy it as a subsequent input assertion.

For any particular program input which satisfies the program's input assertion, the values computed upon arrival at the next cut satisfy its associated cut assertion. But that, in turn, guarantees that the values computed upon arrival at the next cut satisfy its associated cut assertion. But the values computed upon arrival at the final program **RETURN** satisfy the program output assertion. Since a proof for *any* program input can be made from the cut proofs, the program is correct for all inputs. The program of Figure 6 has two cuts and therefore two cut executions. The cut trees for these two cut executions are shown in Figures 7 and 8. Since "verified" is printed at each leaf of these trees, the program is correct.

5. PROCEDURES

Any proof technique must be able to cope effectively with programs which call subroutines and functions. (We will denote the

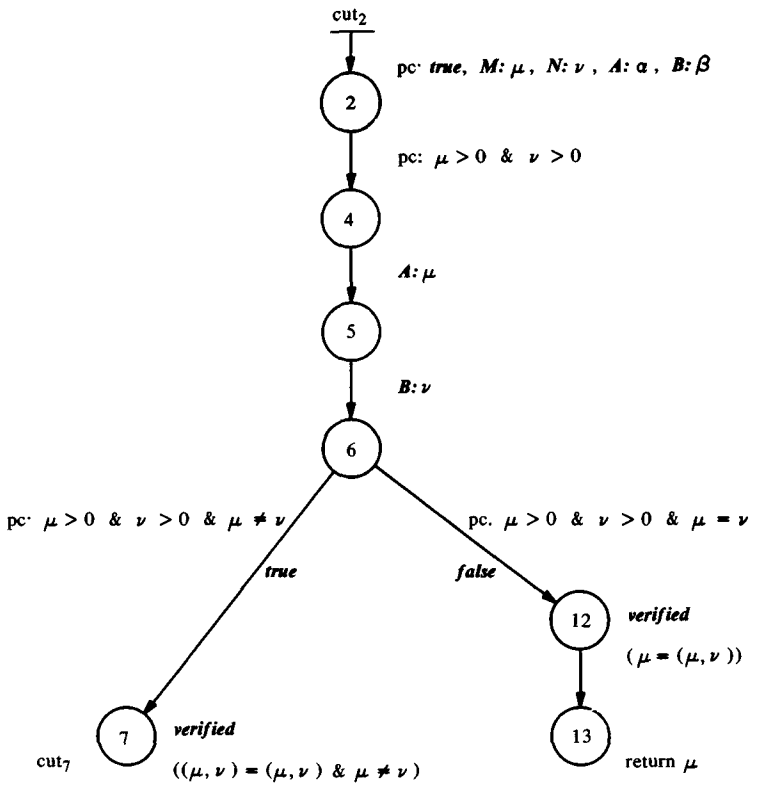

FIGURE 7. Cut tree for cut_2 of *GCD*.

subroutines and functions, subprocedures.) The notion of symbolic execution naturally extends to such calls, which involve transfer of control (with provision for return), some reassociation of variables and values and the creation/destruction of local procedure variables. All these operations remain conceptually the same whether the values of variables are symbolic formulas or numbers. A symbolic execution tree for a symbolic program execution including procedure calls is also conceivable, as is a proof of correctness as already presented. Consider the revised greatest common divisor procedure called *GCD2* shown in Figure 9. It has been modified so as to call the *ABSOLUTE* procedure of Figure 1. The cut execution for cut_2 is identical to that of Figure 7 for procedure *GCD*. The cut execution for cut_7 is shown in Figure 10. The symbolic execution "executes into" the procedure *ABSOLUTE*. The nodes in the tree resulting from executing statements in *ABSOLUTE* are denoted by triangles to distinguish them from those of *GCD2*.

However, this method requires one to "start from scratch" with every proof, reproving properties of each subprocedure at each invocation. A method which allows one to prove a procedure is correct and then use this proof as a lemma in the proof for a calling procedure is needed. Such a method is described next.

Once a procedure (subprocedure) has been proved to be correct with respect to some input/output assertion pair, two consistent sources of information about that procedure's behavior exist: 1) The procedure itself as an executable algorithm (the procedure's codebody); and 2) properties proved correct as described in the input/output assertions.

There are two important points to note about the program characterization pro-

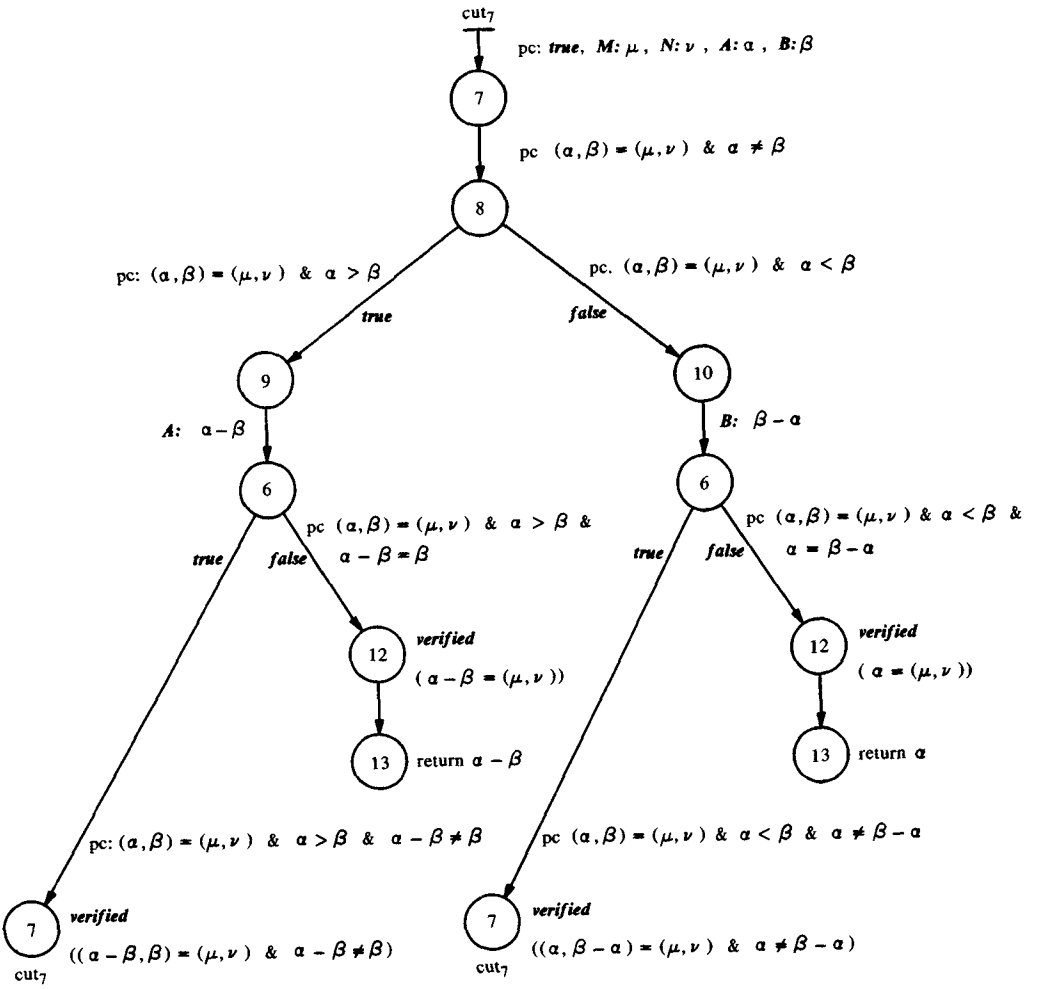

FIGURE 8. Cut tree for cut_7 of *GCD*.

vided by the input/output assertions:

1) They do not necessarily characterize everything the program does, but just some of the effects. Any program is correct with respect to the output assertion *true* which states nothing about the results. The output assertion is supplied by the programmer and includes only what he feels is important.

2) The information is not generally a description of how to compute the results, but rather a description of properties which the results satisfy. The output assertion on the program of Figure 2 ($(Y = X' \mid Y = -X')$ & $Y \geq 0$ & $X = X'$) does not describe how to calculate such a value for Y. The input/output assertions are more usable in a correctness proof for a calling procedure, than the subprocedure codebody since they do not involve the dynamics of how to calculate the procedure results but simply describe them.

The basic approach for dealing with subprocedures appeals to the same idea as does the main technique: use symbols to represent arbitrary values of program variables. The effect of executing a procedure is the alteration of some of the values of the calling procedure's variables and, in the case of a function call, the additional effect of returning a value for the function. New unique symbols are invented, one for each variable of the calling procedure which could have

```
1  GCD2:
       PROCEDURE (M, N);
2  cut2---  ASSUME (M > 0 & N > 0);
3        DECLARE M, N, A, B, D INTEGER;
4        A ← M;
5        B ← N;
6        DO WHILE (A ≠ B);
7  cut7---    ASSERT ((A, B) = (M, N) & A ≠ B);
8          D ← ABSOLUTE(A − B);
9          IF A > B
10           THEN A ← D;
11           ELSE B ← D;
12       END;
13       PROVE (A = (M, N));
14 return---RETURN (A);
15     END;
```

FIGURE 9. *GCD2* procedure which calls *ABSOLUTE*.

had its value altered by the procedure call. Instead of symbolically executing the procedure codebody, the values of the potentially affected calling program's variables are replaced by these new symbols. If the subprocedure has been proved correct, its output assertion holds for these new values and provides the information about these values needed for the proof.

The complete process can be explained precisely as the normal symbolic execution of an "abbreviated procedure" which is derived simply from the original subprocedure as follows:

1) Change the procedure's initial **ASSUME** statement to a **PROVE** statement, leaving its argument unchanged.

2) Change the procedure's final **PROVE** statement to an **ASSUME** statement, leaving its argument unchanged.

3) Replace the complete code body of the procedure by a sequence of assignment statements, one for each variable which can be altered by the procedure, of the form:

$$v_i \leftarrow \textbf{NEWSYMBOL};$$

The built-in function **NEWSYMBOL** is defined to return as its value a new symbolic value each time it is called.

The abbreviated procedure for the procedure *ABSOLUTE* of Figure 2 is shown in Figure 11.

Consider a program P which makes refrence (either by **CALL**s or by function references) to procedures $Q_1, Q_2, \ldots, Q_n$. Assume that each procedure Q_i has been proved correct with respect to its **ASSUME/PROVE** statements. Replace each procedure Q_i by its abbreviated procedure. Suppose that P has **ASSUME/PROVE** statements and that each loop of P has been cut by an **ASSERT** statement. A proof of correctness of P, with respect to its input/output assertions, under the assumption of the correctness of the procedures Q_i, proceeds just as described before in Sections 3 and 4. The cut execution for cut_7 from *GCD2* of Figure 9, using the abbreviated procedure of Figure 11, is shown in Figure 12. The cut execution for cut_2 remains the same as before and is shown in Figure 7. These two cut executions establish the correctness of the procedure *GCD2*.

Whenever an invocation of a procedure occurs during the symbolic execution of paths of P, the abbreviated procedure is invoked. Suppose Q is such an abbreviated procedure. The proof of Q assures one that

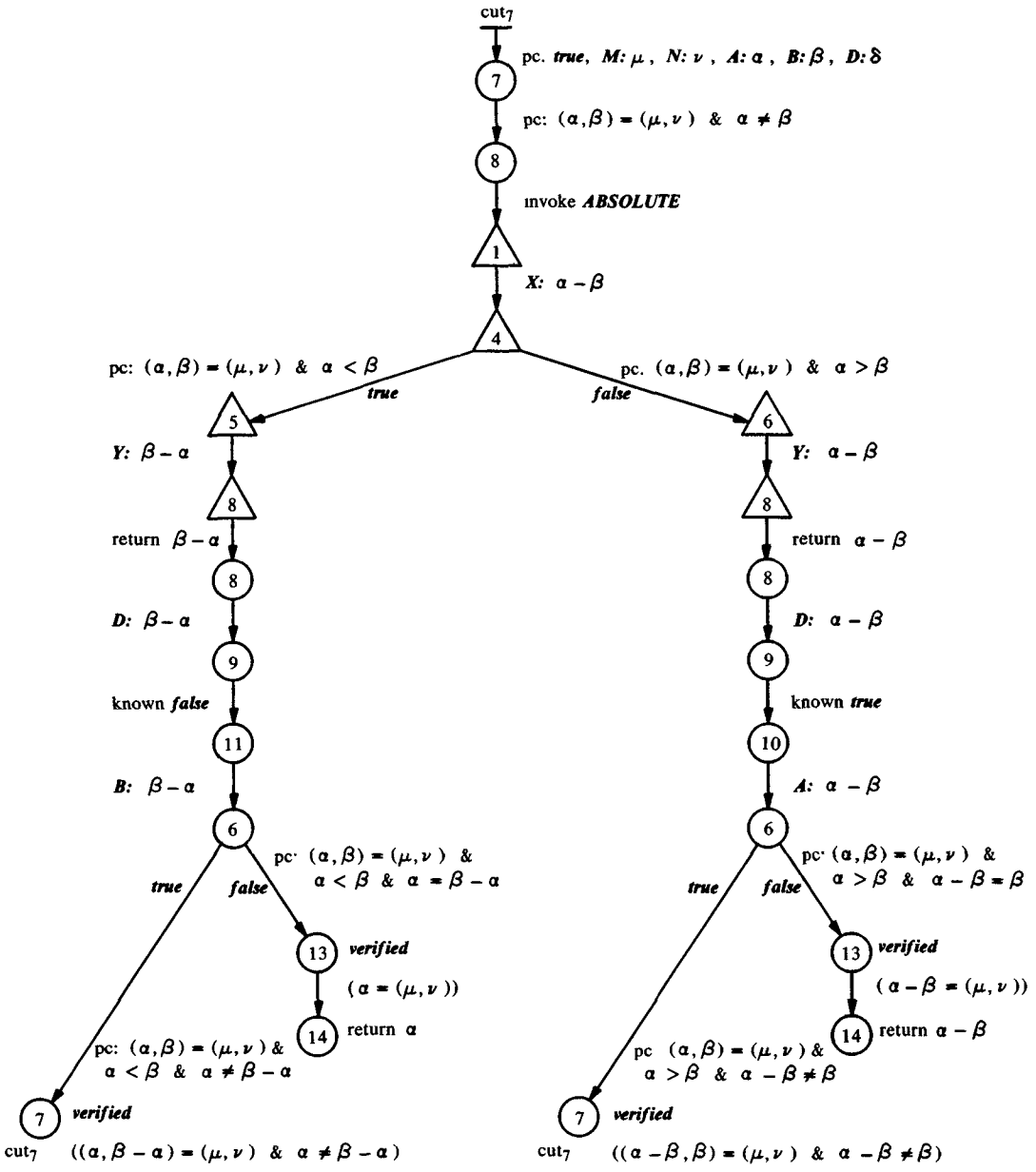

FIGURE 10. Cut tree for cut_7 of *GCD2*.

if the initial input assertion of Q is satisfied by the values of the variables at its invocation, the output assertion will be satisfied by the values at the **RETURN**. Therefore, one must now show that the values input to this invocation satisfy Q's original **ASSUME** statement. The first step in creating the abbreviated procedure (i.e., changing **ASSUME** to **PROVE**) provides the simple key to accomplishing this check. The symbolic execution into the abbreviated procedure up to and including the **PROVE** statement accomplishes the check in each case. A **PROVE** statement is defined to print "verified" or "not verified" depending upon whether or not its associated predi-

```
1  ABSOLUTE:
     PROCEDURE (X);
2      PROVE (true);
3      DECLARE X, Y INTEGER;
4      X ← NEWSYMBOL;
5      Y ← NEWSYMBOL;
6      ASSUME ((Y = X' | Y = −X') & Y ≥ 0 & X = X');
7      RETURN (Y);
8    END;
```

FIGURE 11. Abbreviated procedure *ABSOLUTE*.

cate is *true*. It should be noted that the pc (used by the **PROVE** statement) is considered as part of the underlying (symbolic) machine state, describing conditions over the symbolic constants. It does not relate to program variables and therefore is unaffected by the procedure invocation and has a global scope across the complete program execution.

If the execution of any **PROVE** statement results in "not verified," the proof of correctness fails. This includes the execution of the **PROVE** statement just discussed. If it does not hold, the output assertion of the procedure cannot be guaranteed to hold and its use would be invalid.

Next the symbolic execution of the abbreviated procedure Q would reset all variables accessible to Q to new unique symbols. These represent the new values the procedure Q would compute in an actual execution. In the case that Q is a function, one of these new symbols, being the value of the **RETURN** statement, would also subsequently be returned as the function's value. The abbreviated procedure next contains an **ASSUME** statement (the output condition of the original Q). The symbolic execution of this statement proceeds according to the previous definition. While the execution follows the same exact rules, one may need to generalize his understanding of the **ASSUME** statement and the pc. Changes to the pc previously were a refinement (constriction) of the case being considered. The execution of this **ASSUME** statement causes, rather, an elaboration. The abbreviated symbolic execution had just previously set all the variables, potentially receiving new values within the procedure, to new symbolic values. The execution of the **ASSUME** statement, in updating the pc, now constrains those new symbols to the case where they satisfy the output assertions of the subprocedure. That is, they now represent the subprocedure changes as characterized by its output assertions.

Control now returns to the calling procedure P. The new symbols invented within Q will become values for variables of P returned from Q and, in the case of function procedures, as *the* return value. The symbolic execution of P continues as before. Whenever "knowledge" of the properties of the new symbols is required as in the execution of subsequent **PROVE** statements, it is available in the pc.

Note that the primed variables occurring in the procedure's output assertions have as their values the *original* input values to the procedure. They are not affected by the assignments of new symbols. In general, after the symbolic execution of the abbreviated procedure, the pc contains expressions relating the procedure inputs (the symbolic expression values of the primed variables) to the procedure outputs (the newly invented symbols). Note also that the symbolic execution of the *abbreviated* procedure involves no loops and no cuts and can be considered as one basic step in the symbolic execution of the calling procedure. The symbolic execution of abbreviated procedures involves the new notions of:

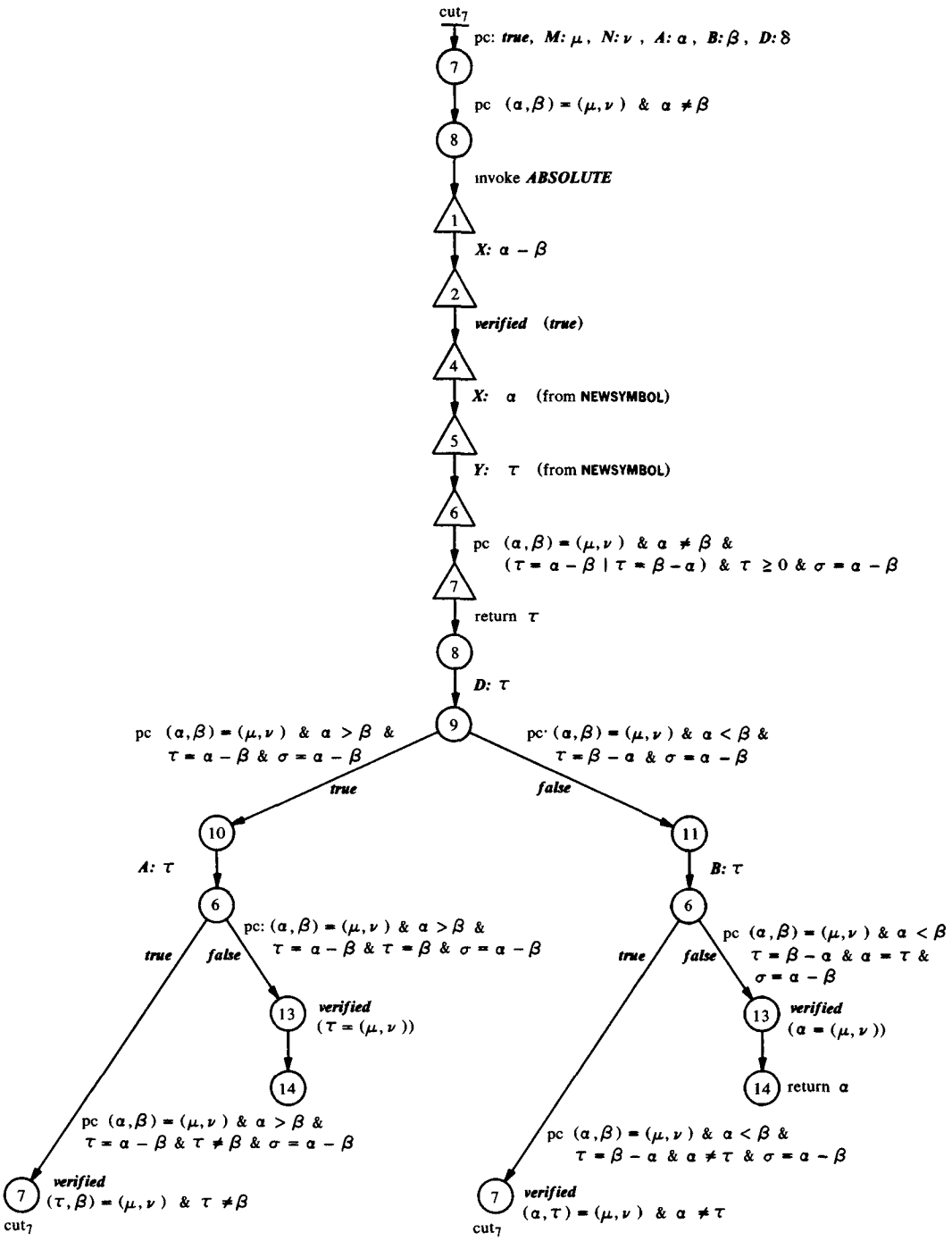

FIGURE 12. Cut tree for cut_7 of *GCD2* (using abbreviated *ABSOLUTE*).

1) The function **NEWSYMBOL** for generating unique new symbols;

2) The remark that the correctness of the program is contingent on *all* executions of the **PROVE** statement printing "verified," including the ones which check the procedure inputs. Those do not occur at a cut as do the others and could be overlooked.

Of course, one does not have to explicitly create an abbreviated procedure but can simply cause the equivalent effects to occur at the procedure invocation.

The definitions given here imply that the output assertions for the subprocedures must include statements of the form $X = X'$ for all variables X which the procedure could alter but, in fact, does not. Of course, such statements would be confirmed during the proof of the procedure itself and are therefore dependable. Thus, for procedures, the output assertions must not only include statements about what the procedure does but also statements about what it does not do (e.g., does not change X). If the programming language itself provides for, and guarantees, the "read-only" nature of some arguments, the statements, $X = X'$ are unnecessary for those parameter variables. In that case, one must also change the definition of abbreviated procedures at step 3 to avoid overwriting these variables with new symbols.

The presence of read-only arguments does eliminate one minor notational nuisance of the method as we describe it. The values of unchanged variables get renamed at each procedure invocation. For example, suppose X is an argument to a procedure and has the value α. Suppose that the subprocedure denotes that parameter by Y and includes $Y = Y'$ in its output assertion. Suppose further that the new symbol invented for Y during the abbreviated execution is β. Then the effect for the calling procedure over the subprocedure's execution is that X will change from α to β, but the pc will include $\alpha = \beta$. With known read-only variables built into the language, X would simply maintain the value α throughout and β would not even be generated.

One feature of the simple language defined and used here causes some problems in proofs involving subprocedures. Consider the procedure shown in Figure 13 which exchanges the values of its two arguments without use of a temporary variable. Its proof of correctness by the method described is quite straightforward. However, if it is called using the same argument for both formal parameters as in:

CALL *EXCHANGE*(Z, Z);

it has the same effect as:

$$Z \leftarrow Z - Z;$$
$$Z \leftarrow Z + Z;$$
$$Z \leftarrow Z - Z;$$

which results in setting Z to zero. (Remember that we have assumed a "call-by-reference" definition like that found in PL/I and FORTRAN.) In this case the output assertion is certainly not satisfied. But what about the alleged proof? At the beginning of each cut execution used in the proof, each

```
1  EXCHANGE:
     PROCEDURE (X, Y);
2      DECLARE X, Y INTEGER;
3      ASSUME (true);
4      X ← X − Y;
5      Y ← X + Y;
6      X ← Y − X;
7      PROVE (X = Y') & (Y = X'));
8      RETURN;
9    END;
```

FIGURE 13. Procedure *EXCHANGE*

procedure variable is initialized to a unique symbolic value *independently and assuming they are distinct variables*. This results in a proof of correctness which only holds for procedure invocations involving distinct arguments.

There are two solutions to this problem. One is to disallow procedure invocations which use the same argument variable in more than one parameter position (at least those which are not read-only within the subprocedure). The other is to extend the proof method to handle such cases correctly. If one considers the form of the **CALL** while performing the proof of correctness, the method is easily extended. For the procedure *EXCHANGE* there are only two cases, represented by:

1) **CALL** *EXCHANGE*(Z, W);
2) **CALL** *EXCHANGE*(Z, Z);

For each case the symbolic executions need be faithful to the rules used in the regular executions. That is, case 2 would treat X and Y as the same variable. The proof of that case would then fail since the final value of Z would be zero. One would then know that this procedure has not been proved correct for calls of the second form, which must be disallowed. The more typical exchange program involving a temporary variable (e.g., with body $T \leftarrow X$; $X \leftarrow Y$; $Y \leftarrow T$) can be proved correct with respect to the given input/output assertions for both forms of calls.

For procedures with many parameters, the number of combinations in which two or more arguments may be coincident is quite large. So one might prove the program is correct for the favored case of no coincidence, and then prove only the forms of coincidence which are needed for the higher-level program proofs.

6. PROBLEMS

Much of the work in performing a proof of correctness of a program is tedious and error prone. Considerable research has been done in attempting to get the computer itself to construct or at least to assist in the construction of program proofs [2, 5, 8, 11, 14, 17, 20, 21]. In fact, the method of proof presented here was developed first by Deutsch [5] for his automated program verifier. In this section, a brief summary of the difficulties in constructing program proofs is given. Some of the problems become exaggerated when one tries to automate the process.

Some effort has gone into developing programming languages in which proofs of correctness are easier [9, 10]. In pursuing this goal one must realize the inherent limitations. A programming language is a medium in which to describe algorithms, perhaps algorithms of a certain type, or which operate on certain data. That medium can encourage obscure descriptions of algorithms and make formal analysis difficult or impossible by providing clumsy, overly general, and ill-defined features. However, the reason an algorithm performs a desired computation correctly is independent of the notation in which it is described. For example, the fact that the simplex algorithm can be used to optimize linear functions subject to a set of linear constraints is based on significant mathematical theory. Without that theory, or without rediscovering it, the most elegant simplex algorithm written in the perfect programming language cannot be proved correct.

The proof method based on symbolic execution is appealing because it is an extension of the notion of normal program execution. One can often devise the "proper" proof technique for a class of language constructs by considering their behavior under regular execution. The implementation of a program verifier on a computer, based on symbolic execution, closely follows that of an interpreter for the language.

Much recent work [3, 19, 23] has been concerned with programs that manipulate complex data structures (e.g., list structures). The difficulty with such programs

appears to be the lack of an established notation, manipulative techniques, and known results concerning data structures. The simple mathematical model of variable and value is complicated by the introduction of, what corresponds to in one form or another, computer storage cells. Variables refer to storage which contains values, and the associations between these (variables, storage, values) are changed by the program execution.

Closely related to the data structure problem is the problem of developing a general, flexible, easy-to-read specification language. A program is proved correct with respect to its input/output assertions. The examples in this presentation were chosen for the ease with which their input/output assertions could be expressed.

A major area of concern in implementing program verifiers on a computer is providing the formula manipulation and theorem proving power required. The symbolic execution of programs requires an efficient and comprehensive formula manipulation system. Each execution of a **PROVE** statement requires establishing the truth of a formula of arbitrary complexity. Program proving has spurred work on efficient, domain-dependent computer theorem provers.

The last major problem area discussed is that of composing inductive assertions. The input/output assertions which a programmer must supply are often difficult to determine. However, the need for such input/output assertions does not seem artificial; a programmer must somehow express what the program was intended to do. The necessary inclusion of inductive assertions which cut the loops in the program does seem artificial. These are required not so much to specify the program properties but as an inductive assistance to the program proving method. One can pose the generation of the inductive assertions as a theorem proving problem by formulating one large theorem for the complete program of the form: show that there exist inductive assertions $P_1, P_2, \ldots, P_n$ such that all of the expressions resulting from **PROVE** statements are *true*. In general, this is a very difficult theorem to prove. For simple programs, inductive assertions can be generated automatically and some exploration into their automatic generation and/or enhancement for larger programs using heuristic methods is reported in [12, 22].

SUMMARY

This paper attempts to give a basic introduction to the fascinating world of proving that computer programs meet their specifications. One style of producing program specifications, in the form of input/output assertions, was introduced to allow a definition of a "correct program." Then the symbolic execution of programs was explained as one way of establishing the consistency between the program's code and its input/output assertions.

Note that, among all the program specification methods and program proof methods which have been proposed and developed, we have presented a very narrow glimpse of one. It is the most intuitive approach of which we know and, therefore, one of the most likely candidates for productive future development and use.

ACKNOWLEDGMENTS

Many of the ideas presented here were the outgrowth of our work with Jerry Archibald, Steve Chase, Ahmed Chibib, Claus Correll, and John Darringer on the EFFIGY system [15].

REFERENCES

[1] BOYER, R. S.; ELSPAS, B.; AND LEVITT, K. N. "SELECT—A formal system for testing and debugging programs by symbolic execution," *Internatl. Conf. on Reliable Software*, 1975, ACM, New York, 1975, pp. 234–245.

[2] BOYER, R. S.; AND MOORE, J. S. "Proving theorems about Lisp functions," *J. ACM* **22**, 1, (Jan. 1975), 48–59.

[3] BURSTALL, R. M. "Some techniques for proving correctness of programs which alter data structures," *Machine intelligence 7*, D. Michie (Ed.), American Elsevier, New York, 1972.

[4] CLARKE, LORI, *A system to generate test data and symbolically execute programs*, Report #CU-CS-060-75, Univ. of Colorado, 1975.

[5] DEUTSCH, L. P. "An interactive program verifier," PhD Dissertation, Dept. Computer Science, Univ. of Calif., Berkeley, 1973, Xerox PARC Report CSL-73-1, Palo Alto, Calif.

[6] ELSPAS, B. et al., "An assessment of techniques for proving program correctness," *Computing Surveys* **4**, 2 (June 1972), 97–147.

[7] FLOYD, R. W. "Assigning meanings to programs," in *Proc. Symposium Applied Math.*, Vol. 19, American Mathematical Society, Providence, R.I., 1967, pp. 19–32.

[8] GOOD, D. I.; LONDON, R. L ; AND BLEDSOE, W. W. "An interactive program verification system," *IEEE Trans. on Software Engineering* **1**, 1, (April 1975), 59–67.

[9] GOOD, D. I.; AND RAGLAND, L. C. "Nucleus—a language of provable programs," in *Program test methods*, W. Hetzel (Ed.), Prentice-Hall Inc., Englewood Cliffs, N.J., 1973, pp. 93–117.

[10] HOARE, C. A. R.; AND WIRTH, N. "An axiomatic definition of the programming language PASCAL," *Acta Informatica* **2**, (1973), 335–355.

[11] IGARASHI, S.; LONDON, R. L.; AND LUCKHAM, D. C. "Automatic program verification I: a logical basis and its implementation," *Acta Informatica* **4**, (1975), 145–182. Also in USC Information Sciences Institute Report ISI/RR-73-11, May 1973.

[12] KATZ, S. M.; AND MANNA, Z. "A heuristic approach to program verification," in *Proc. Third Internatl. Joint Conf. on Artificial Intelligence*, SRI Publications Dept. Stanford Calif., 1973, pp. 500–512.

[13] KING, J. C. "Proving programs to be correct," *IEEE Trans on Computers* **C-20**, 11, (Nov. 1971), 1331–1336.

[14] KING, J. C. "A program verifier," PhD Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1969.

[15] KING, J. C. "A new approach to program testing," in *Internatl. Conf. on Reliable Software*, 1975, ACM, New York, 1975, pp. 228–233. Also appears in *Programming methodology, lecture notes in computer science*, **23**, Springer-Verlag Inc., New York, 1974, pp. 278–290.

[16] KING, J. C. "Symbolic execution and program testing," *Comm. ACM* **19**, 7, (July 1976), 385–394.

[17] LONDON, R. L. "The current state of proving programs correct," in *Proc. of ACM Annual Conf* , 1972, ACM, New York, 1972, pp. 39–46.

[18] MANNA, Z. *Mathematical theory of computation*, McGraw-Hill Book Co., New York, 1974.

[19] OPPEN, D. C.; AND COOK, S. A. "Proving assertions about programs that manipulate data structures," in *Seventh Annual ACM Symposium on Theory of Computation*, 1975, ACM, New York, 1975, pp. 107–116.

[20] SUZUKI, N. "Automatic program verification II verifying programs by algebraic and logical reduction," in *Internatl. Conf. on Reliable Software*, 1975, ACM, New York, 1975, pp. 473–481.

[21] TOPOR, R. W. "Interactive program verification using virtual programs," PhD Dissertation, Univ. of Edinburgh, Edinburgh, Scotland, 1975.

[22] WEGBREIT, B. "The synthesis of loop predicates," *Comm. ACM* **17**, 2, (Feb. 1974), 102–112.

[23] WEGBREIT, B.; AND SPITZEN, J. M. "Proving properties of complex data structures," *J. ACM* **23**, 2 (April 1976), 389–396.