Formal Program Verification Using Symbolic Execution

ROGER B. DANNENBERG AND GEORGE W. ERNST

Abstract—Symbolic execution provides a mechanism for formally proving programs correct. A notation is introduced which allows a concise presentation of rules of inference based on symbolic execution. Using this notation, rules of inference are developed to handle a number of language features, including loops and procedures with multiple exits. An attribute grammar is used to formally describe symbolic expression evaluation, and the treatment of function calls with side effects is shown to be straightforward. Because symbolic execution is related to program interpretation, it is an easy-to-comprehend, yet powerful technique. The rules of inference are useful in expressing the semantics of a language and form the basis of a mechanical verification condition generator.

Index Terms—Control constructs, program proving, program verification, rules of inference, side effects, symbolic execution, verification conditions.

I. Introduction

An accepted way of proving things about programs is to use rules of inference like those introduced by Hoare [3]. Such rules are usually formulated so that they can be applied to the last statement of a program, yielding one or more shorter programs and possibly some formulas in logic to be verified. By iteratively applying rules of inference, the task of proving program correctness is reduced to that of proving statements in predicate calculus.

Another proof technique is based on the notion of symbolic execution [2]; statements are processed in the same order that an interpreter would execute them, as opposed to the "backward" order of the first method. It appears as though both techniques are equally powerful and logically equivalent. However, the close analogy between symbolic execution and interpretive execution seems to make the symbolic execution method easier to comprehend. In particular, symbolic execution is a natural paradigm for expressions which can contain function calls with side effects. One of our goals is to develop rules of inference which formalize symbolic execution. To our knowledge, such a formalism has not appeared in the literature to date.

Our second goal is to present rules of inference for some advanced control constructs. Rules of inference for a loop statement based on Zahn's construction [10] are presented, as are rules for procedures with multiple labeled exits.

Finally, we will introduce an extended notation which allows the formal treatment of expressions with side effects. Side effects are strictly disallowed in verification-oriented languages such as Pascal [9] and Euclid [5], but it will be shown that this restriction can be relaxed with little difficulty.

The ideas in this paper grew out of a project to design and implement a mechanical verification condition generator [1]. Early in the project it became clear that we needed a concise formal notation for stating verification rules for the constructs found in contemporary programming languages. We found the notation presented in this paper to be a good solution to this problem.

After an introduction to symbolic execution, notation and rules for a simple language are described. The rules are then extended to handle multiple-exit loops, and procedures with multiple labeled exits. Finally, a formal treatment of expressions with side effects is presented.

II. Concepts of Symbolic Execution

In symbolic execution the values of program variables are represented by symbolic constants or expressions. For example, the value of variable v might be represented by "B + 3," where B is a symbolic constant (not a program variable). The collection of all variables and their values is called a *state*. As a program path is "executed," assumptions about the state are recorded in the *path condition*.

For example, suppose we wish to verify the following program which sets x to its absolute value:

```
pre x=y;
if x < 0 then x := -x endi;
post x >= 0 & (y=x or y=-x)
```

The first and last lines contain input and output specifications called ***pre-*** and ***postconditions***. The keyword **endi** is the closing bracket for **if**. In the initial state, variables x and y have arbitrary constants X and Y as their values. For now, we will write the state as follows:

State = {x is X, y is Y}.

Next we evaluate the precondition by replacing each variable by its value in the current state, obtaining X=Y. Note that X and Y are values while x and y are variables. We assume the precondition is true when the program begins by setting the path condition to X=Y:

Manuscript received September 24, 1979. This work was supported in part by the National Science Foundation under Grants MCS-77-24236 and SPI-78-21198.

R. B. Dannenberg was with the Department of Computer Engineering and Science, Case Institute of Technology, Case Western Reserve University, Cleveland, QH 44106. He is now with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

G. W. Ernst is with the Department of Computer Engineering and Science, Case Institute of Technology, Case Western Reserve University, Cleveland, OH 44106.

0098-5589/82/0100-0043$00.75

State = {x is X, y is Y}, PathCond = (X=Y).

The path taken through the program depends on the value of the expression x<0. We symbolically evaluate x<0 to obtain X<0, and first assume that it is true by "anding" it to the path condition, obtaining

State = {x is X, y is Y}, PathCond = (X=Y & X<0).

The statement "x := - x" is executed by changing the state so that the value of x is - X:

State = {x is - X, y is Y}, PathCond = (X=Y & X<0).

Now we want to show the postcondition is true. Substituting values from the current state into the postcondition yields

- X>=0 & (Y=- X **or** Y=- - X).

Since this is implied by the path condition, this path is verified.

Following the other path (X<0 is false) gives rise to the path condition

PathCond = (X=Y & ~(X<0))

where "~" is the logical negation operator. The postcondition becomes

X>=0 & (Y=X **or** Y=- X).

Again, this follows from the path condition, so this path is verified. The program is correct because all paths have been verified.

III. Notation and Terminology

To formalize this approach, some notation is introduced. A *substitution* S is denoted by

S = [t1/x1, t2/x2, . . . , tn/xn].

where the term ti is to be substituted for variable xi, and the x's are all distinct. An instance of an expression is obtained when a substitution is applied to the expression. This operation is indicated by "|", e.g., B|[e/x] indicates the instance of B in which each occurrence of x is replaced by e.

Substitutions can be *composed* to form a new substitution. The composition of R = [u1/y1, u2/y2, . . .] and S = [t1/x1, t2/x2, . . .] is RS = [u1|S/y1, u2|S/y2, . . . , t1/x1, t2/x2, . . .], where ui|S/yi is omitted if ui|S = yi, and ti/xi is omitted if some yj = xi. For example, the composition of [X/I, X+1/X] and [C/X, 25/Z] is [C/I, C+1/X, 25/Z].

For any expression b, and substitutions R and S, the instance b|(RS) obtained by applying the composition of R and S to b is identical to the instance (b|R)|S obtained by left-to-right application of R and S.

The identity substitution is [ ], i.e., b|[ ] = b.

The application operator has higher precedence than logical connectives, e.g., in the formula P&Q|S, S is only applied to Q. If Q stands for a statement with many components, S is applied to each of them.

Substitutions are used to formalize the state concept in symbolic execution. The value of program variable x in state S (which is a substitution) is x|S.

To formalize symbolic execution, we will use formulas of the form S, PC\A to indicate the correctness of statement list A, given an initial path condition PC and state S. A is a statement sequence, the last of which is a statement of the form **confirm** Q. This specifies that Q must be true of the final state. Hence, the formula [ ], P\A; **confirm** Q corresponds to P{A}Q in the more conventional notation found in the literature.

Rules of inference have the following form:

$$\frac{A}{B}\cdot$$

This means that, given A and B, we can infer C. In practice, the rules are used "backwards"—the rule is used to reduce the problem of proving C to that of proving both A and B.

To be more precise about the concept of correctness, we say that a procedure is *partially correct* if its postcondition is true whenever the procedure terminates (exits), given that the precondition was initially satisfied. A procedure is *totally correct* if it is partially correct and it always terminates. This discussion will not consider termination proofs, and the term "correct" is taken to mean "partially correct."

IV. Simple Rules of Inference

In this section, rules of inference are presented for five statement types—assignment, conditional, iteration, confirm, and procedure call. It is assumed that expressions do not contain function calls, but procedures can alter global variables and variable parameters.

A. Assignment Statement

$$\frac{[exp/x]S, PC \setminus A}{S, PC \setminus x := exp; A}$$

This rule states that the statement x:=exp changes the value of x to the value of the expression exp. For example, this rule reduces the verification of

[3/x], PC \ x := x+1; A

to the verification of

[x+1/x][3/x], PC \ A.

Composing substitutions in the latter yields

[3+1/x], PC \ A.

This corresponds to our intuitive understanding of assignments.

B. Conditional Statement

$$\frac{S, PC \,\&\, B|S \setminus A2; A1 \qquad S, PC \,\&\, \sim B|S \setminus A3; A1}{S, PC \setminus \textbf{if } B \textbf{ then } A2 \textbf{ else } A3 \textbf{ endi}; A1}$$

The keyword **endi** is the closing bracket for **if**. The conditional expression B is evaluated by applying it to the state S. There are two possibilities: B|S or ~B|S. The proof is by cases. In the first case, B|S is assumed and the statement sequence A2 is executed before A1. In the second case, ~B|S is assumed and A3 is executed before A1. These cases result in

the first two lines of the rule. If both cases can be verified, the **if** statement followed by A1 must be correct.

C. Confirm Statement

$$\frac{PC \rightarrow Q|S}{S, PC \setminus \textbf{confirm } Q}$$

The **confirm** statement asserts that Q must be true for the program to be correct. A confirm statement is verified by proving that Q is implied by the path condition when the variables in Q are replaced by the values in the current state.

D. Iteration Statement

$$\frac{\begin{array}{l}PC \rightarrow I|S\\ [\ ], I\&B \setminus A2; \textbf{ confirm } I\\ [\ ], I\&\sim B \setminus A1\end{array}}{S, PC \setminus \textbf{while } B \textbf{ maintaining } I \textbf{ do } A2 \textbf{ endw}; A1}$$

The keyword **endw** is the closing bracket for **while**. The iteration statement is a "while loop" that contains a loop invariant. The first line of this rule states that the loop invariant I must be true when the loop statement is encountered. The next two lines consider the cases B and ~B. If B is true, the loop body is executed and we want to show that I is maintained as a loop invariant, as stated by the second line. The state is replaced by the identity substitution because nothing is known about the state (except on the first iteration) due to assignments in A2. Assuming B is not true, the loop terminates and A1 is executed. After an arbitrary number of iterations, the only other assumption we can make is I, the loop invariant. This gives the third line of the rule.

E. Procedure Call Statement

Consider the following procedure declaration:

```
procedure p (var x, const y);
   use var g;
   use const c;
   pre P;
   post Q;
   A
end p;
```

Procedure p has a variable (value-result) parameter x, a constant (value) parameter y, and a global variable g which is altered by p, and a global c which is not altered by p. The body of p is the statement sequence A. The postcondition Q may refer to the initial and final values of x and g; their initial values are #x and #g while their final values are x and g, respectively.

We assume that no two distinct variables in the program have the same name, e.g., if a procedure uses "x" to name a local variable, then no other procedure can declare a variable or parameter by the same name. If this assumption does not hold, variables are simply renamed to produce an equivalent program which is suitable. This eliminates the ambiguities normally resolved through scope rules, since each identifier refers to a unique variable.

The rule of inference for a call to this procedure is

$$\frac{\begin{array}{l}PC \rightarrow P|[v/x, e/y]S\\ [v'/v][g'/g]S, PC \,\&\, Q|S1 \setminus A\end{array}}{S, PC \setminus p\,(v, e); A}$$

where S1 = [v'/x, e/y, g'/g, v/#x, g/#g]S and v', g' are new unique constants.

The first line of the rule indicates that the procedure precondition must be true in the current state. The substitution [v/x, e/y] is used to replace the formal parameters by the actual parameters. Composing the substitution with S evaluates the actual parameters in the current state.

The second line reflects the calling program after the execution of procedure p. The new state is [v'/v][g'/g]S. The global variable g will have the new value g', and the variable parameter v acquires the new value v'. The order of substitution for v and g is important because they might be the same variable. This places a call by value-result interpretation on parameters, because v' may be "copied" onto g'.

The new path condition is PC & Q|S1. Let us look at the substitution S1 applied to the postcondition Q. Unique constants v' and g' are substituted for the formal variable parameter x and altered global g. The constant parameter e (which may be an expression) is substituted for its formal parameter y. The initial value of the actual variable parameter v is substituted for #x, and g is substituted for #g.

For example, consider the following procedure, derived from our previous program which computes absolute values:

```
procedure abs (var x);
   pre TRUE;
   post x>=0 & (x=#x or x=- #x);
   if x<0 then x=- x endi
endp
```

The variable parameter x is changed to its absolute value. Notice how #x is used in the postcondition to refer to the initial value of x. Assuming abs is correct, suppose we want to verify

$$[\ ], B \setminus v{:=}-2; \text{abs}(v); \textbf{confirm } v=2$$

where B is some path condition. The rule for assignment statements is applied, yielding

$$[-2/v], B \setminus \text{abs}(v); \textbf{confirm } v=2.$$

Now we use the rule for procedure calls. The first line of the rule says to prove

$$B \rightarrow \text{TRUE}|[v/x][-2/v].$$

The second line of the procedure call rule yields

$$[v'/v][-2/v], B \,\&\, (x>=0 \,\&\, (x=\#x \textbf{ or } x=-\#x))|S1 \setminus \textbf{confirm } v=2$$

where S1 = [v'/x, v/#x][-2/v] = [v'/x, -2/#x, -2/v]. This is simplified to

$$[v'/v], B \,\&\, (v'>=0 \,\&\, (v'=-2 \textbf{ or } v'=--2)) \setminus \textbf{confirm } v=2.$$

The **confirm** rule indicates we must now prove

$$B \,\&\, (v'>=0 \,\&\, (v'=-2 \textbf{ or } v'=--2)) \rightarrow (v=2)|[v'/v].$$

The consequent is simplified by applying the substitution, yielding

$$B \& (v' >= 0 \& (v' = -2 \text{ or } v' = --2)) \rightarrow v' = 2$$

which is true. Therefore, the path and hence the calling program is verified.

V. Multiple-Exit Loops

In this section, several new rules are introduced to allow an equivalent of Zahn's loop construction [10]. The general form of our loop construct is

```
loop altering v;
   A1;
   maintain I;
   A2
end1;
select
   L1: B1 enda;
   L2: B2 enda;
   ...
   Ln: Bn enda
ends;
```

The phrase **altering** v indicates that the variable v can be modified by the loop body, which consists of statement list A1, a **maintain** statement, and statement list A2. Either A1 or A2 may be empty. One and only one **maintain** statement is included in the loop body to provide the loop invariant.

Each **loop** statement must be followed by a **select** statement which contains labeled statement lists called **alternatives**. Within A1 and A2, there may be statements of the form **exit** Li, which cause immediate transfer of control of the alternative labeled Li. Control is only transferred to within the **select** statement following the immediately enclosing loop; however, to effect multiple-level exits, alternatives may contain **exit** statements themselves.

A. Multiple-Exit Loop Statement

$$\frac{\begin{array}{l}S, PC \setminus A1; \textbf{confirm } I; \textbf{end}; A3 \\ [v'/v]S, PC \,\&\, I \mid [v'/v]S \setminus A2; A1; \textbf{confirm } I; \textbf{end}; A3\end{array}}{S, PC \setminus \textbf{loop altering } v; A1; \textbf{maintain } I; A2 \textbf{ end1}; A3}$$

where v′ is a new unique constant.

With this loop statement, there are two cases to be proven. In the first case (line 1), the loop body is executed up to the **maintain** statement by executing A1. The **confirm** statement indicates that the loop invariant must be true at this point. The reason for the

end; A3

after the **confirm** statement is that A1 may contain an **exit** (discussed in the next section) which transfers control to A3; the keyword **end** marks the end of the loop. A3 contains the **select** and subsequent statements. In the second case (line 2), the loop invariant is assumed, with unique constants as values for altered variables. The loop body is then executed, starting and ending at the **maintain** statement. Again, the loop invariant must be confirmed. By induction, it can be shown that the assertion I in the **maintain** statement will always be true if the two cases hold.

A further explanation of the **altering** clause is in order. In the literature (and also in Section IV-D), the iteration rule is usually based on the assumption that the loop body can alter every variable in the state. Therefore, the state is discarded by using the identity substitution, and the only information carried out of the loop is contained in the invariant I and the exit condition (i.e., ~B in Section IV-D). This makes it necessary to use invariants that state details about variables *not* altered by the loop, because the loop invariant must be strong enough to eventually allow the proof of the final confirm statement. To simplify the task of finding sufficiently strong invariants, an **altering** clause is added to the **loop** statement to specify which variables might be modified by the body of the loop. Now, most of the state will usually be unchanged by the loop, and the loop invariant only needs to specify how the loop affects the variable that it alters. The concept of the **altering** clause is due to Ogden [8].

B. Exit Statement

$$\frac{S, PC \setminus \textbf{branch } L; A2}{S, PC \setminus \textbf{exit } L; A1; \textbf{end}; A2}$$

where A1 does not contain the statement **end**.

This rule simply states that an **exit** statement causes control to transfer to the statement following the immediately enclosing loop. Rules for the **branch** statement are given below. Recall that the first statement in A2 is a **select** statement because it follows a **loop** statement.

C. Branch Statement

$$\frac{S, PC \setminus A1; A3}{S, PC \setminus \textbf{branch } L; \textbf{select } L: A1 \textbf{ enda}; A2 \textbf{ ends}; A3}$$

$$\frac{S, PC \setminus \textbf{branch } L1; \textbf{select } A2 \textbf{ ends}; A3}{S, PC \setminus \textbf{branch } L1; \textbf{select } L2: A1 \textbf{ enda}; A2 \textbf{ ends}; A3}$$

where L1 and L2 are distinct identifiers.

These rules describe the action of transfering control to a labeled alternative. Note that control passes to the end of the **select** statement after the **alternative** statement list is executed. These rules assume that the branch label must be listed in the **select** statement.

For example, we can apply the exit rule to

```
S, PC \ exit TWO; ... ; end;
        select ONE: ... enda;
               TWO: p(x) enda;
             THREE: ... enda ends; ...
```

obtaining

```
S, PC \ branch TWO; select ONE: ... enda;
                           TWO: p(x) enda;
                         THREE: ... enda ends; ...
```

The second **branch** rule applies, yielding

S, PC \ **branch** TWO; **select** TWO: p(x) **enda**;
THREE: . . . **enda ends**; . . .

Now the first **branch** rule is used to obtain

S, PC \ p(x); . . .

VI. Multiple-Exit Procedures

The general form of the procedures dealt with in this section is given in Fig. 1. This form is like that of Section IV-E except two labeled postconditions are provided to allow several control paths to emerge from a procedure call. Statements in the body of procedure p are denoted by A. To simplify the notation in this section we restrict procedures to one variable (value-result) parameter, one constant (value) parameter, two global variables, only one of which can be modified, and two exit labels. The generalization to an arbitrary number of these constructs is straightforward.

Each procedure call is followed by a **select** statement with labeled **alternatives** corresponding to the labels in the postcondition. An example of such a call is

```
p(v, 10);
select
   L1: . . . enda;
   L2: . . . enda
ends;
```

Within p, statements of the form **exit** L1 and **exit** L2 are allowed. When an **exit** is executed, control is transferred back to the calling procedure. The alternative corresponding to the **exit** label is executed next as was the case with the loops in Section V.

A. Procedure Declaration

The procedure declaration in Fig. 1 is verified by proving

[x/#x, g/#g], P | [x/#x, g/#g] \

A **end**; **select** L1; **confirm** Q1; **enda** L2; **confirm** Q2 **enda ends**.

Each procedure declaration in a program is verified independently in a similar manner. The substitution is used to save the initial values of x and g in #x and #g so that the postconditions (and loop invariants) in the body of p can refer to them. Notice that we are utilizing the property that the value of a variable not mentioned in a substitution is the variable itself, i.e., [x/#x] is the same as [x/#x, x/x]. Thus, x and #x have the same initial value, x.

B. Multiple-Exit Procedure Call Statement

$$\frac{\begin{array}{l}PC \rightarrow P \mid [v/x, e/y]S \\ [v'/v]\,[g'/g]S, PC \,\&\, Q1 \mid S1 \setminus \textbf{branch } L1; A \\ [v'/v]\,[g'/g]S, PC \,\&\, Q2 \mid S1 \setminus \textbf{branch } L2; A\end{array}}{S, PC \setminus p(v, e); A}$$

where S1 = [v'/x, e/y, g'/g, v/#x, g/#g]S and v', g' are new constants.

This rule assumes that p has been declared as in Fig. 1. Recall that the first statement in A is a **select** statement, because a

```
procedure p (var x, const y);
   use var g;
   use const c;
   pre P;
   post L1: Q1, L2: Q2;
   A
endp;
```

Fig. 1. Generic procedure declaration.

call on a multiple-exit procedure must be followed by a **select statement**. The new procedure-call rule is like the first (see Section IV-E) except for each label, a different postcondition is assumed, and a different path is taken. The **branch** statements direct program flow to the proper **alternative**. The substitution S1 is identical to S1 in Section IV-E.

As an example, consider the following procedure which decrements a nonnegative integer unless it is zero:

```
procedure dec (var x);
   pre x>=0;
   post OK: x = #x-1 & x>=0,
        ZERO: #x=0 & x=0;
   if x>0 then x := x-1; exit OK
          else exit ZERO
   endi
endp;
```

Now suppose we wish to verify

```
[v/#v], v<10 & v>=0 \ dec(v);
                    select   OK: enda;
                          ZERO: v := 9 enda ends;
                    confirm v := (#v-1) mod 10
```

Using the procedure-call rule, the first line tells us to verify

v<10 & v>=0 → x>=0 | [v/x] [v/#v]

which simplifies to v<10 & v>=0 → v>=0, which is true. The second and third lines require the verification of a path for each exit label. The first path is

```
[v'/v] [v/#v],
v<10 & v>=0 & (x = #x-1 & x>=0) | [v'/x, v/#x] [v/#v] \
   branch OK; select . . .
```

which simplifies to

```
[v'/v, v/#v], v<10 & v>=0 & v'=v-1 & v'>=0 \
   branch OK; select OK: enda; . . .
```

Applying the first **branch** rule gives us

```
[v'/v, v/#v], v<10 & v>=0 & v'=v-1 & v'>=0 \
   confirm v = (#v-1) mod 10
```

The **confirm** rule yields the following

```
v<10 & v>=0 & v'=v-1 & v'>=0 →
   (v=(#v-1) mod 10) | [v'/v, v/#v]
```

which simplifies to

v<10 & v>=0 & v'=v-1 & v'>=0 → v' = (v-1) **mod** 10.

This is also true. Taking the ZERO path yields

[v'/v] [v/#v],
v<10 & v>=0 & (#x=0 & x=0)| [v'/x, v/#x] [v/#v] \
branch ZERO; . . .

which simplifies to

[v'/v, v/#v], v<10 & v>=0 & v=0 &
v'=0 \ **branch** ZERO; **select** . . .

Several applications of the **branch** rules give us

[v'/v, v/#v], v<10 & v>=0 & v=0 & v'=0 \
v := 9; **confirm** v = (#v-1) **mod** 10.

Application of the assignment rule and some simplification yields [9/v, v/#v], v<10 & v>=0 & v=0 & v'=0 \ **confirm** v = (#v-1) **mod** 10. The **confirm** rule requires the proof of

v<10 & v>=0 & v=0 & v'=0 → (v = (#v-1)
mod 10)| [9/v, v/#v]

which simplifies to

v<10 & v>=0 & v=0 & v'=0 → 9 = (v-1) **mod** 10

which is true.

VII. Expressions and Functions

Symbolic expression evaluation, without side effects and function calls, is easily carried out by instantiating expressions with substitutions as in the above inference rules. To incorporate functions with side effects, an attribute-grammar [4] is used to formally describe expression evaluation. The treatment of Pascal-like structured variables is also described.

A. Notation

All operations are written in functional notation, e.g., A + B is written +(A, B). This removes concern over operator precedence. Structure references are denoted by

acc(A, Slist)

where A is a structure and Slist is a list of selectors, i.e., array indexes and record field-names (pointers will not be considered here). The function "acc" is called the **access** function. The expression

ch(A, Slist, t),

is a structure whose value is identical to that of A except at the element named by Slist, where the value is t. The function "ch" is the **change** function. The **change** and **access** notation is due to McCarthy and Painter [7]. The concept was extended by Luckham and Suzuki [6].

A new mechanism is now introduced to describe state changes. The **modify** function M(S, x, v) takes as arguments a substitution S, a variable x, and a value v. The result is the substitution in which the value of x is v, and all other variables have the same values as in S. In other words, if S is [t1/x1, t2/x2, . . . , ti/xi, . . .], then M(S, xi, v) is [t1/x1, t2/x2, . . . , v/xi, . . .]. The relation between **modify** and composition is illustrated by

[u/x]S = M(S, x, v) when v = u|S.

Recall that in performing the composition, S is applied to u. The **modify** function changes the value of x to v without applying S. This function will be used in the rules that follow. We further define the notation M(S, acc(A, Slist), v) to mean M(S, A, ch(A, Slist, v)). This generalization provides a mechanism whereby values can be substituted for structure elements rather than just simple variables. This definition is used in the rules for assignments, and procedure and function calls.

We will often want to modify several variables at once; the notation

M(S, v1/x v2/x2, . . . , vn/xn)

will be used as an abbreviation for

M(. . . M(M(S, x1, v1), x2, v2) . . . xn, vn).

All nonterminals in the attribute grammer have eight attributes:

Ci an inherited verification condition list
Cs a synthesized verification condition list
Ss a synthesized state
Si an inherited state
Ps a synthesized path condition
Pi an inherited path condition
V a synthesized value
L a synthesized location.

Attributes are further qualified by nonterminals in the production rule, e.g., Pi(expr) is the inherited path condition attribute for the nonterminal "expr." If the same nonterminal occurs more than once in a production rule, integers are appended to distinguish the occurrences, e.g., "expr1" and "expr2."

The effect of expression evaluation is denoted by

E(expr, S, PC) = (L, V, S', PC', VC)

where S and PC are the state and path condition before the evaluation of expr, S' and PC' are the state and path condition after evaluation of expr; L is the symbolic location if expr is a variable or structure reference; V is the symbolic value if expr; and VC is a list of verification conditions for expr. For example, suppose we evaluate the expression x, and the state has the value 10 for x:

E(x, [10/x], p) = (x, 10, [10/x], p, nil).

The location is x, the value is 10, the new state and path condition are unchanged, and there are no verification conditions.

The attribute grammar is presented below with comments to help the reader.

1) expr ::= id:

Cs(expr) = Ci(expr)
Ss(expr) = Si(expr)
Ps(expr) = Pi(expr)
V(expr) = V(id)|Si(expr)
L(expr) = V(id).

If an expression is an identifier, then its evaluation is performed by finding the value of the identifier in the state. The path condition and state are unchanged. We will not show the production rules for "id." Instead, we will simply state that the value attribute of "id," i.e., V(id), is the identifier itself.

A constant or a field-name also evaluates to itself; we will not give an explicit rule for them.

2) expr ::= acc(id, slist):

Ci(slist) = Ci(expr)
Si(slist) = Si(expr)
Pi(slist) = Pi(expr)
Cs(expr) = Cs(slist)
Ss(expr) = Ss(slist)
Ps(expr) = Ps(slist)
V(expr) = acc(V(id)|Ss(slist), V(slist))
L(expr) = acc(V(id), V(slist)).

To evaluate a structure-reference, the selector-list is evaluated; then the structure is accessed. Notice that side effects from the evaluation of slist can affect the value of "expr." Also notice that the location attribute has the same selector list as the value attribute, but the structure name, V(id), is not instantiated by the state, Ss(slist).

3) slist ::= expr:

Ci(expr) = Ci(slist)
Si(expr) = Si(slist)
Pi(expr) = Pi(slist)
Cs(slist) = Cs(expr)
Ss(slist) = Ss(expr)
Ps(slist) = Ps(expr)
V(slist) = V(expr)
L(slist) = nil.

4) slist1 ::= expr slist2:

Ci(expr) = Ci(slist1)
Si(expr) = Si(slist1)
Pi(expr) = Pi(slist1)
Ci(slist2) = Cs(expr)
Si(slist2) = Ss(expr)
Pi(slist2) = Ps(expr)
Cs(slist1) = Cs(slist2)
Ss(slist1) = Ss(slist2)
Ps(slist1) = Ps(slist2)
V(slist1) = V(expr) V(slist2)
L(slist1) = nil.

Selector lists are evaluated from left to right. The value is the list of evaluated list elements.

5) expr1 ::= op(expr2):

Ci(expr2) = Ci(expr1)
Si(expr2) = Si(expr1)
Pi(expr2) = Ci(expr1)
Cs(expr1) = Cs(expr2)
Ss(expr1) = Ss(expr2)
Ps(expr1) = Ps(expr2)
V(expr1) = V(op) (V(expr2))
L(expr1) = nil.

6) expr1 ::= op(expr2, expr3):

Ci(expr2) = Ci(expr1)
Si(expr2) = Si(expr1)
Pi(expr2) = Pi(expr1)
Ci(expr3) = Cs(expr2)
Si(expr3) = Ss(expr2)
Pi(expr3) = Ps(expr2)
Cs(expr1) = Cs(expr3)
Ss(expr1) = Ss(expr3)
Ps(expr1) = Ps(expr3)
V(expr1) = V(op) (V(expr2), V(expr3))
L(expr1) = nil.

Primitive operations are performed by evaluating the operands from left to right and returning the result of applying the operator to the evaluated operands.

Our rule for function calls deals with functions of the form shown in Fig. 2. As with previous procedure rules, a generic example is used, since its generalization to multiple parameters and global variables is straightforward. Functions do not contain **exit** statements; an implicit **exit** follows the function body. The statement f := e assigns e to be the value returned by f. The function production rule is similar to the rule for procedure calls. The main differences are that a value (the constant f′) is returned, and parameter evaluation can have side effects.

7) expr ::= f(v, e):

Ci(v) = Ci(expr)
Si(v) = Si(expr)
Pi(v) = Pi(expr)
Ci(e) = Cs(v)
Si(e) = Ss(v)
Pi(e) = Ps(v)
Cs(expr) = Cs(e) Ps(e) → P|M(Ss(e), V(v)/x, V(e)/y)
Ss(expr) = M(Ss(e), g′/g, v′/L(v))
Ps(expr) = Ps(e) &
  Q|M(Ss(e), f′/f, v′/x, g′/g, V(e)/y, V(v)/#x,
  (g|Ss(e))/#g)
V(expr) = f′
L(expr) = nil.

where v′, f′, g′ are new unique constants, and Fig. 2 gives the declaration of f. As above, sharped (#) symbols refer to initial values.

The first step in the symbolic execution of f(v, e) is to evaluate the parameters which may contain function calls which change the initial state Si(expr) and path condition Pi(expr) and add items to the verification condition list. Symbolic execution then adds the precondition of f to the list of verification conditions Cs(e) after the appropriate substitution for variables. The new values v′ and g′ of v and g are recorded in the output state Ss(expr). The postcondition of f is added to the path condition after the appropriate substitution for variables. The value returned, f′, is substituted for f in the postcondition. The replacement of formal parameters by actuals is essentially the same as the procedure-call rule in Sections IV-E and VI-B. The additional intricacies in this section stem from

```
function f(var x, const y);
   use var g;
   use const c;
   pre P;
   post Q;
   A
endf;
```

Fig. 2. Generic function declaration.

the possibility of side effects in parameter evaluation and our provision for structure references (e.g., A[i]) as actual variable parameters.

VIII. Rules for Statements

We are now prepared to present rules of inference for statements, allowing expressions with side effects. The rules will be similar to the previous versions, but expression evaluation is accomplished using the "E" function rather than through application of substitutions. We can now define this function more precisely in terms of attributes:

E(expr, Si(expr), Pi(expr)) =
(L(expr), V(expr), Ss(expr), Ps(expr), Cs(expr))

where Ci(expr) is the empty list of verification conditions. The rules for statements are listed in Fig. 3.

The assignment rule says that the expression on the left-hand side is evaluated first, yielding a location, L (see Fig. 3). The right-hand side is then evaluated yielding a value V. The state is modified by changing the value of L to V. The verification conditions, C1 and C2, resulting from the evaluation of expr1 and expr2 must be true. Notice that the final state reflects side effects from expr1 and expr2.

For example, consider the path

[3/i, 5/j], TRUE \ j := inc(A[i]); **confirm** Q

where inc is the function defined below:

```
function inc (var x);
   pre TRUE;
   post x = #x+1 & inc=x;
   x := x+1;
   inc := x
endf;
```

Assuming this function definition is correct we will symbolically execute the assignment statement. First we evaluate the left-hand side as follows:

E(j, [3/i, 5/j], TRUE) = (j, 5, [3/i, 5/j], TRUE, nil).

The value 5 is obtained from the first attribute-grammar production which states

V(expr) = V(id)|Si(expr)

which gives us

V(j) = j|[3/i, 5/j] = 5.

The value of the right-hand side is determined by evaluating

E(inc(A[i]), [3/i, 5/j], TRUE).

A[i] is rewritten as acc(A, i) where i is the selector list. The value of i is i|[3/i, 5/j] = 3. The value of A is A|[3/i, 5/j] = A.

Next, we evaluate inc(acc(A, 3)). The precondition of inc is TRUE, so the verification condition attribute Cs is

TRUE → TRUE.

From the production for functions, the new state is

M([3/i, 5/j], v′/acc(A, 3))
= M([3/i, 5/j], ch(A, 3, v′)/A)
= [3/i, 5/j, ch(A, 3, v′)/A].

The new path condition is

TRUE & (x=#x+1 & inc=x)|S, where S =
M([3/i, 5/j], v′/x, inc′/inc, acc(A, 3)/#x)
= [3/i, 5/j, v′/x, inc′/inc, acc(A, 3)/#x].

By applying S, the path condition is simplified to

TRUE & (v′=acc(A, 3)+1 & inc′=v′).

The value of the right-hand side is inc′. The result of evaluating inc(A[i]) is

E(inc(A[i]), [3/i, 5/j], TRUE) =
(nil, inc′, [3/i, 5/j, ch(A, 3, v′)/A],
TRUE & (v′=acc(A, 3)+1 & inc′=v′), TRUE → TRUE).

Notice that the location attribute is nil, indicating that the expression cannot be the target of an assignment or a variable parameter. The verification condition results from the precondition of inc. Now, according to the rule for assignments, we must establish

M([3/i, 5/j, ch(A, 3, v′)/A], inc′/j),
TRUE & (v′ = acc(A, 3)+1 & inc′=v′) \ **confirm** Q

which simplifies to

[3/i, inc′/j, ch(A, 3, v′)/A],
TRUE & (v′ = acc(A, 3)+1 & inc′=v′) \ **confirm** Q

Thus, the statement has the effect of incrementing A[3] and assigning the new value to j.

The new conditional rule is like the first one in Section IV-B except the evaluation of expr can result in a different path condition and state (PC′ and S′), and the precondition of its function calls must be verified.

The **confirm** rule is identical to the previous one.

The loop rule is essentially unchanged. For consistency, the **modify** function is used in place of the composition operation used previously. The **exit** rule and **branch** rules are identical to the ones in Sections V-B and V-C.

The rule for procedure calls in Fig. 3 is similar to the attribute-grammar production for function references. The

1) Assignment Statement

C1
C2
M(S', V/L), PC' \ A

S, PC \ expr1 := expr2 ; A

where E(expr1, S, PC) = (L, V1, S1, PC1, C1)
and E(expr2, S1, PC1) = (L2, V, S', PC', C2)

2) Conditional Statement

C
S', PC' & V \ A2; A1
S', PC' & ~V \ A3; A1

S, PC \ **if** expr **then** A2 **else** A3 **endi**; A1

where E(expr, S, PC) = (L, V, S', PC', C).

3) Confirm Statement

PC → Q|S

S, PC \ **confirm** Q

4) Loop Rule

S, PC \ A1; **confirm** B; **end**; A3
S', PC & B|S' \ A2; A1; **confirm** B; **end**; A3

S, PC \ **loop altering** v; A1; **maintaining** B; A2 **endl**; A3

where S' is M(S, v'/v)
and v' is a new unique constant.

5) Exit Statement

S, PC \ **branch** L; A2

S, PC \ **exit** L; A1; **end**; A2

where A1 does not contain the statement **end**.

6) Branch Statement

S, PC \ A1; A3

S, PC \ **branch** L; **select** L: A1 **enda**; A2 **ends**; A3

S, PC \ **branch** L1; **select** A2 **ends**; A3

S, PC \ **branch** L1; **select** L2: A1 **enda**; A2 **ends**; A3

7) Procedure Call Statement

Cv
Ce
PCe → P|M(Se, Vv/x, Ve/y)
M(Se, g'/g, v'/Lv), PCe & Q1|S1 \ **branch** L1; A
M(Se, g'/g, v'/Lv), PCe & Q2|S1 \ **branch** L2; A

S, PC \ p(v, e); A

where E(v, S, PC) = (Lv, Vv, Sv, PCv, Cv),
E(e, Sv, PCv) = (Le, Ve, Se, PCe, Ce),

S1 = M(Se, v'/x, Ve/y, g'/g, Vv/#x, (g|Se)/#g),

g' and v' are new unique constants,
v is the actual variable parameter,
e is the actual constant parameter,
x is the formal variable parameter,
y is the formal constant parameter,
g is the global altered by p,
L1, L2 are exit labels,
P is the precondition of p,
Q1, Q2 are the postconditions of q,
and sharped (#) symbols refer to initial values.

8) Procedure Declaration

Each procedure declaration like Fig. 1 must be proved correct by verifying

[x/#x, g/#g], P|[x/#x, g/#g] \
A; **select** L1: **confirm** Q1 **enda**
L2: **confirm** Q2 **enda ends**

9) Function Declaration

Each function declaration like Fig. 2 must be proved correct by verifying

[x/#x, g/#g], P|[x/#x, g/#g] \ A; **confirm** Q

Fig. 3. Rules of inference.

changes to the state are the same except the constant f', representing the value of the function, is not necessary. This is also true of the substitution applied to the postcondition(s). A verification condition is generated for each exit label as in the previous procedure-call rule.

As before, the correctness of a procedure declaration is established by assuming its precondition, executing its body, and confirming its postcondition. A select statement is used to select the appropriate component of the postcondition for the exit that is taken. The initial state contains the values of #'d variables.

A function declaration is verified in a similar manner as shown in Fig. 3. Recall that functions can only have a single exit. All procedures and functions called from the body of a procedure or function must be correct by the same definition.

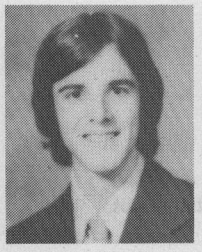
IX. Summary

The notion of symbolic execution facilitates the writing and understanding of formal rules of inference for proving program correctness. Using these rules, a language with side effects, multiple-exit loops, and procedures with multiple exits has been formally described. A similar set of rules has been used as the basis of a mechanical verification condition generator [1]. In our experience, the implementation of a verification condition generator is relatively straightforward once the verification rules have been precisely stated using the notation developed in this paper. This is particularly true of the symbolic execution of expressions. The details of the attribute grammer in Section VII are quite intricate, but the implementation of the attribute grammar is very similar to its formal specification.

References

[1] R. B. Dannenberg, "An extended verification condition generator," Dep. Comput. Sci., Case Western Reserve Univ., Tech. Rep. CES-79-3, 1979.
[2] S. L. Hantler and J. C. King, "An introduction to proving the correctness of programs," *Comput. Surveys*, vol. 8, pp. 331-353, Sept. 1976.
[3] C.A.R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576-580, Oct. 1969.
[4] D. E. Knuth, "Semantics of context-free languages," *Math Syst. Theory*, vol. 2, no. 2, pp. 127-145, 1968, and vol. 5, no. 1, pp. 95-96, 1971.

[5] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Revised report on the programming language Euclid," Xerox Res. Center, Tech. Rep. CSL 78-2, 1978; An earlier version appeared in *SIGPLAN Notices*, vol. 12, Feb. 1977.
[6] D. C. Luckham and N. Suzuki, "Verification of array, record, and pointer operations in Pascal," *ACM Trans. Programming Languages Syst.*, vol. 1, pp. 226-244, Oct. 1979.
[7] P. McCarthy and J. A. Painter, "Correctness of a compiler for arithmetic expressions," in *Proc. Symp. Appl. Math.*, vol. 19. Providence, RI: Amer. Math. Soc., 1967, pp. 33-41.
[8] W. F. Ogden, personal communication.
[9] N. Wirth, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35-63, 1971.
[10] C. T. Zahn, "A control statement for natural top-down structured programming," in *Proc. Programming Symp.*, Paris, Apr. 9-11, 1974, *Lecture Notes in Computer Science*, vol. 19. Springer-Verlag, 1974, pp. 170-179.



Roger B. Dannenberg was born in Houston, TX, in 1955. He received the B.S.E.E. degree from Rice University, Houston, in 1977, the M.S. degree in computer engineering from Case Western Reserve University, Cleveland, OH, in 1979, and the M.S. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1980.

From 1978 to 1981, Mr. Dannenberg held an NSF fellowship. He is currently a Hertz fellow at Carnegie-Mellon, pursuing the Ph.D. degree in computer science. His research interests include distributed systems, parallel processing, computer languages, and computer music.

Mr. Dannenberg is a member of Phi Beta Kappa, Tau Beta Pi, and the Association for Computing Machinery.

George W. Ernst was born in St. Marys, PA, in 1939. He received the Ph.D. degree in electrical engineering from Carnegie Institute of Technology in 1966.

Upon completion of his Ph.D., he joined the faculty of Case Institute of Technology, Case Western Reserve University, Cleveland, OH, where he is currently an Associate Professor of Computer Engineering and Science. He coauthored a book with Allen Newell entitled *GPS: A Case Study in Generality and Problem Solving* (New York: Academic, 1969). He has contributed to leading journals in the areas of artificial intelligence, mechanical theorem proving, and program verification. He was an ACM National Lecturer and is a Past Chairman of the ACM Special Interest Group on Artificial Intelligence (SIGART).

Analysis of a Hybrid Access Scheme for Buffered Users-Probabilistic Time Division

ANTHONY EPHREMIDES, SENIOR MEMBER, IEEE, AND OSAMA A. MOWAFI, MEMBER, IEEE

Abstract—**A new multiple access scheme is proposed and evaluated. The proposed scheme combines desirable features of the ordinary time-division (TDMA) and the random access (RA) schemes. It is shown that by adjusting the value of a single parameter *a*, the proposed access method can vary continuously from one extreme (TDMA) to the other (RA). The average delay per packet and the throughput can be improved for intermediate values of the load factor. Furthermore, the method can control the channel instability.**

Index Terms—**Delay, multiple access, packet switching, stability, throughput.**

Manuscript received August 10, 1980; revised May 29, 1981. This work was supported in part by the National Science Foundation under Grant ENG-7722752 and the Naval Research Laboratory under Task Area RR021-05-41.

A. Ephremides is with the Department of Electrical Engineering, University of Maryland, College Park, MD 20742.

O. A. Mowafi is with the Networks Architecture Department, Computer Sciences Corporation, Falls Church, VA 22046.

I. Introduction

In packet-switched broadcast channels the problem of multiple access has received several solutions ranging from schemes of the dedicated type, like TDMA and FDMA, through demand assignment and reservation methods, to the completely random methods of contested access like the Aloha scheme.

On the one end of the spectrum we have the class of dedicated methods. The chief representative of this class is the widely known TDMA method which has been extensively used in data communications, but only recently analyzed in the context of satellite or packet-radio communications [1], [2]. It is known that this scheme performs satisfactorily in terms of channel utilization (throughput) and average packet delay if the traffic is heavy (high load factor) or if the user terminals produce data on a regular (almost periodic) basis. Otherwise the performance of TDMA tends to become unsatisfactory.

0098-5589/82/0100-0052$00.75