Symbolic Evaluation

Richard A. Kemmerer

Department of Computer Science
University of California
Santa Barbara, California

References

"An Introduction to Proving the Correctness of Programs," S. Hantler and J. King, *Computing Surveys*, **Vol. 8, No. 3, September 1976.**

"Unisex: A Unix-Based Symbolic Executor for Pascal," R. Kemmerer and S. Eckmann, *Software Practice and Experience,* **Vol. 15, No. 5, May 1985.**

Verification Condition Generator

Symbolic Execution

Symbolic Execution uses a standard mathematical technique of letting symbols represent arbitrary program inputs.

What we need is a machine that performs algebraic manipulation on the symbolic expressions.

Inputs **are represented symbolically**
$\alpha_1, \alpha_2, \alpha_3, \ldots$

Variables **may have values of:**

- **signed integer constants**
- α_i
- **expressions formed from** α_i **and constants**

Program *state* includes

- **value of variables**
- **program counter**
- **path condition (PC)**

<u>Need a rule for each statement in the language</u>

Assignment: **X := E**

Replace all variables in the expression E with their current values. The resulting expression becomes the new value of variable X.

Program counter is set to the next instruction and path condition is unchanged.


```
1  PROCEDURE TEST (A, B: INTEGER;
                        VAR X, Y, Z: INTEGER);
2  BEGIN

3    X := A + B;

4    Y := A - B;

5    Z := X + Y

6  END;
```


	PC	A	B	X	Y	Z
1	true	$\alpha 1$	$\alpha 2$	$\alpha 3$	$\alpha 4$	$\alpha 5$
3						
4						
5						

Assume (<Boolean Expression>)

Variables in the boolean expression are replaced by their current symbolic values. Let B represent this expression.

$PC_{new} = PC_{old}$ & B

Prove (<Boolean Expression>)

Let B represent the symbolic expression that results from replacing all the variables in the boolean expression with their current values.

If PC $\longrightarrow$ B

then print "Verified"

else print "Not verified"


```
1    PROCEDURE TEST (A, B: INTEGER;
                          VAR X, Y, Z: INTEGER);
2    BEGIN
2.5      ASSUME ( A > B )
3      X := A + B;
4      Y := A - B;
5      Z := X + Y
5.5      PROVE ( X = A' + B' & Y = A' - B'
                          & Z = 2A' & Y > 0 )
6    END;
```


	PC	A	B	X	Y	Z
1	true	α_1	α_2	α_3	α_4	α_5
2.5						
3						
4						
5						
5.5						

Verification Condition (VC)

$\alpha_1 > \alpha_2 \rightarrow \alpha_1 + \alpha_2 = \alpha_1 + \alpha_2$

$\& \ \alpha_1 - \alpha_2 = \alpha_1 - \alpha_2$

$\& \ \alpha_1 + \alpha_2 + \alpha_1 - \alpha_2 = 2\alpha_1$

$\& \ \alpha_1 - \alpha_2 > 0$


```
1  PROCEDURE ABSOLUTE ( X : INTEGER;
                          VAR Y: INTEGER );
2  BEGIN
3    ASSUME ( TRUE );
4    IF X< 0
5       THEN  Y := -X
6       ELSE  Y := X;
7    PROVE (( Y = X' |  Y = - X' ) &
                 Y ≥ 0  & X = X' )
8  END;
```


IF (< bool. expr.>) THEN $stat_1$ ELSE $stat_2$

Let B represent evaluated boolean expression

- If PC $\rightarrow$ B then execute $stat_1$
- If PC $\rightarrow$ ~B then execute $stat_2$
- If PC $\not\rightarrow$ B and PC $\not\rightarrow$ ~B

  then two cases to be considered

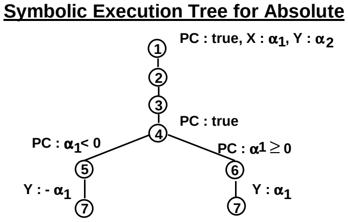
  Case 1: (B is true)
  $PC_{new} = PC_{old}$ & B
  execute $stat_1$

  Case 2 : (B is false)
  $PC_{new} = PC_{old}$ & ~B
  execute $stat_2$

Can use tree structure to represent the symbolic execution

- each node represents a statement in the program
- each branch point corresponds to a forking IF

Symbolic Execution Tree for Absolute



Prove :

$\alpha_1 < 0 \rightarrow (- \alpha_1 = \alpha_1 \ | \ - \alpha_1 = - \alpha_1)$

$\& \ - \alpha_1 \geq 0 \ \& \ \alpha_1 = \alpha_1$

Prove :

$\alpha_1 \geq 0 \rightarrow (\alpha_1 = \alpha_1 \ | \ \alpha_1 = - \alpha_1)$

$\& \ \alpha_1 \geq 0 \ \& \ \alpha_1 = \alpha_1$


```
1  FUNCTION FACT ( N : INTEGER ) : INTEGER;
2  VAR X,Y : INTEGER;
3  BEGIN
4     ASSUME ( N ≥ 0 );
5     X := 0;
6     Y := 1;
7     WHILE X < N DO BEGIN
8        X := X + 1;
9        Y := Y * X
10       END;
11    PROVE ( Y = N! );
12    FACT := Y
13 END;
```

WHILE <bool. expr.> DO <stat>

Let B represent the evaluated boolean expression

If PC $\longrightarrow$ B then execute the statement list followed by the same WHILE statement

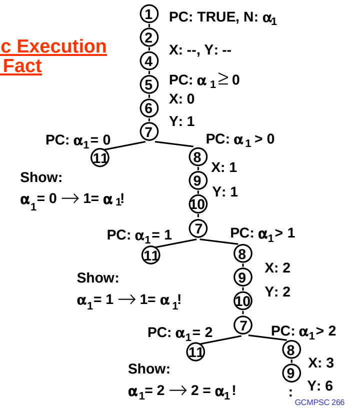
If PC $\longrightarrow$ ~B then execute the statement following the WHILE construct

If PC $\not\longrightarrow$ B and PC $\not\longrightarrow$ ~B then two cases to be considered

Case1:(B is true)
PCnew = PCold & B
execute the statement list followed by the same WHILE statement

Case 2: (B is false)
PCnew = PCold & ~B
execute the statement following the WHILE construct

Symbolic Execution Tree for Fact



Inductive Assertions

**ENTRY {INITIALIZATION; LOOP_INVARIANT
WHILE LOOP_TEST DO LOOP_BODY;
FINALIZATION} EXIT**

ENTRY {INITIALIZATION} LOOP_INVARIANT

**LOOP_INVARIANT & LOOP_TEST
{LOOP_BODY}
LOOP_INVARIANT**

**LOOP_INVARIANT & ~LOOP_TEST
{FINALIZATION}
EXIT**

Need an Assert Statement

ASSERT (<boolean expression>)

Let B be the evaluated boolean expression

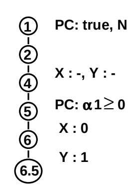
**IF PC $\longrightarrow$ B
THEN previous sequential code is verified
ELSE not verified**

Assign fresh symbolic values to all of the variables and let B be the boolean expression evaluated with these values

PCnew = B (to be used to verify the following piece of sequential code)

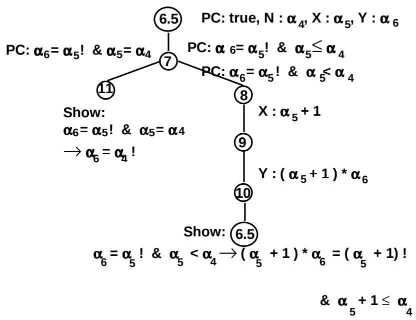

```
1   FUNCTION FACT ( N : INTEGER ) : INTEGER;
2   VAR X,Y : INTEGER;
3   BEGIN
4     ASSUME ( N ≥ 0 );
5     X := 0;
6     Y := 1;
6.5   ASSERT ( Y = X! & X ≤ N )
7     WHILE X < N DO BEGIN
8        X := X + 1;
9        Y := Y * X
10       END;
11    PROVE ( Y = N! );
12    FACT := Y
13  END;
```


New Symbolic Execution Tree for Fact



Show:
$\alpha_1 \geq 0 \longrightarrow 1 = 0! \ \& \ 0 \leq \alpha_1$

New Symbolic Execution Tree for Fact (cont.)




```
1   FUNCTION FACT ( N : INTEGER ) : INTEGER;
2   VAR X,Y : INTEGER;
3   BEGIN
4     ASSUME ( N ≥ 0 );
5     X := N;
6     Y := 1;
6.5   ASSERT (                    )
7     WHILE X >0 DO BEGIN
8       Y := Y * X;
9       X := X - 1
10      END;
11    PROVE ( Y = N! );
12    FACT := Y
13  END;
```


For every statement in the language need to define effect on the state

(i.e., the effect on the variables, program counter, and path condition)

Consider Case Statement

case
<boolean$_1$>: statement$_1$;
<boolean$_2$>: statement$_2$;
.
.
<boolean$_n$>: statement$_n$;
else: statement$_{n+1}$
esac

Pascal Case Statement

case <expression> of
<case label list$_1$>: statement$_1$;
<case label list$_2$>: statement$_2$;
.
.
<case label list$_n$>: statement$_n$
end

If the current value of the selector (i.e., the expression) is not equal to any of the values in any of the case label lists, then the effect is undefined

Subroutines and Functions

Could just place the subprocedure inline after assigning actual parameters to formals

Subroutines and Functions

Could just place the subprocedure inline after assigning actual parameters to formals

Problems

Need to reprove across statements each time the subprocedure is used


```
PROCEDURE GCD2 ( M,N : INTEGER;
                         VAR RESULT: INTEGER);
VAR B,D: INTEGER;
BEGIN
  ASSUME ( M>0 & N>0 );
  RESULT := M;
  B := N;
  ASSERT ( (RESULT,B) = (M,N) );
  WHILE RESULT<>B DO BEGIN
        ABSOLUTE ( RESULT-B, D);
        IF RESULT>B
           THEN RESULT :=D;
           ELSE B := D;
   END;
   PROVE ( RESULT = (M,N) );
END;
```



```
PROCEDURE ABSOLUTE ( X : INTEGER;
                         VAR Y: INTEGER );
BEGIN
  ASSUME ( TRUE );
  IF X< 0
     THEN Y := -X
     ELSE Y := X;
  PROVE (( Y = X' | Y = - X' ) &
               Y ≥ 0 & X = X' )
END;
```


Want a method to prove a subprocedure is consistent with its specifications and to use this information as a lemma to prove the calling procedure

Proof Method

- Change the called procedure's initial *assume* to *prove*
- Change called procedure's final *prove* to *assume*
- Replace code body of called procedure by assignment statements (one for each variable that can be altered)

When the procedure is called actuals are assigned to formals

Abbreviated Absolute Procedure

```
PROCEDURE ABSOLUTE ( X : INTEGER;
                         VAR Y: INTEGER );
BEGIN
  PROVE ( TRUE );
  Y := NEWSYMBOL;
  ASSUME (( Y = X' | Y = - X' ) &
                   Y ≥ 0 & X = X' )
END;
```

```
PROCEDURE TWOLOOPS (N: INTEGER);
VAR  J,K: INTEGER;
BEGIN
    J := 1;
    WHILE  J ≤ N
        J := J+1;
    K := 1;
    WHILE  K ≤ N
        K := K+1
END;
```