‘Symbolic Execution and Program Testing’ By James C. King

Marie Nesfield

31st Jan 2017

Testing	**Symbolic Execution**	**Proving**
Exercises a single program trace--Dynamic	Tries to exhaust all program traces--Dynamic	Need not be dynamic
Ensures property holds on single trace	Ensures property holds on as much as maybe executed	Ensures property holds for entire program
Fuzzing: number of test cases is determined by how many RNG produces.	Number of cases is determined by the branching behavior of the program	

Questions:
How much human involvement is needed in each of the cases? How can we eliminate humans?
Why is proving difficult? We’ve really only talked about how we can specify these things so far.
How does one go about generating good test cases?
All three cases have to have an idea of what good or bad is. The formulation and specificity needed may vary.
Problem: still need to determine what the code *should* do.

Ideal:

The authors list 3 reasons their approach is ideal:

1. The authors restrict themselves to arbitrary length integers (this impediment has been improved upon since)
2. Manny programs have infinite execution trees (this is a fundamental problem)
3. Theorem proving for modest programming languages is impossible (advancements have been made here too)

Section 9 mentions what would happen if a theorem prover said that an IF statement could be resolved if it couldn’t. The path would be taken, when it should not be, and we would have an over approximation of what could actually happen in the system, and a path condition that should evaluate to false.

Programming language:

Variables: all of type signed integer

$IF,\ THEN,\ ELSE$

$GoTo$: transfers program control to a specified label

Means of obtaining inputs (parameters, read operations, global variables)

Arithmetic expressions: integer operators: $\times,\ +,\ -$

Boolean expression: $\{arithexpr \geq 0\}$

Symbolic values (from program input): $\{\alpha_1 \ldots \alpha_i\}$

Integer polynomials: expressions formed over: $\{\alpha_1 \ldots \alpha_i\}$, integer coefficients, (,), $\times,\ +,\ -$, $\{arithexpr \geq 0\}$

Assignment : $\leftarrow$: transfers an integer polynomial to a variable

State: values of program variables, position of the program counter, pc

pc: path constraint, boolean expression over symbolic inputs $\{\alpha_1 \ldots \alpha_i\}$ (initialized to true, always satisfiable)

Takes the form R: a polynomial over the set : $\{\alpha_1 \ldots \alpha_i\}$. May be of the form $(R \geq 0)$ or $\neg(R \geq 0)$

Example: $R = \{\alpha_1 \geq 0 \wedge \alpha_1 + 2 \times \alpha_2 \geq 0 \wedge \neg(\alpha_2 \geq 0)\}$

Evaluating IF($q \geq 0$): if $pc \supset q$ take the if side, or $pc \supset \neg q$ take the else case. Otherwise, execute the IF with $pc = pc \wedge q$, and execute the ELSE with $pc = pc \wedge \neg q$.

Trees:

Node: corresponds to a single instruction

Branch: corresponds to a forking IF, one subtree corresponds to the statement being true, the other false.

Properties:

pc in distinct branches will be distinct

There exist concrete inputs to reach any branch

How if statement works:

1. FOO: PROCEDURE(A); $//pc = \{true\},\ A = \alpha_1$
2. IF(A)
3. THEN: PRINT("HAPPY TUESDAY"); $//pc = \{\alpha_1 \geq 0\}$
4. ELSE: PRINT("IT IS NOT TUESDAY"); $//pc = \{\neg(\alpha_1 \geq 0)\}$
5. RETURN;

Sum:

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

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

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

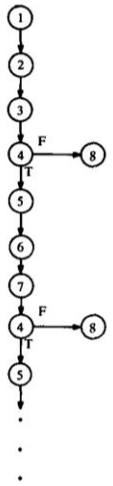
Fig. 3. Symbolic execution of SUM (α_1, α_2, α_3). Path condition is abbreviated *pc*.

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

POWER:

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

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



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

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

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

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

After statement	J	X	Y	Z	*pc*
	Values would be				
1	?	α_1	α_2	?	*true*
2	—	—	—	1	—
3	1	—	—	—	—

4 execution in detail:
- (a) evaluate $Y \geq J$ getting $\alpha_2 \geq 1$.
- (b) use path condition and check:
  - (i) *true* $\supset \alpha_2 \geq 1$
  - (ii) *true* $\supset \neg(\alpha_2 \geq 1$
- (c) neither are theorems, so fork.

Case $\neg(\alpha_2 \geq 1)$:

4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$

8 this case completed. (returns 1 when $\alpha_2 < 1$.)

Case $\alpha_2 \geq 1$:

4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	—	—	—	α_1	—
6	2	—	—	—	—
7	—	—	—	—	—

4 execution in detail:
- (a) evaluate $Y \geq J$, getting $\alpha_2 \geq 2$
- (b) use *pc*:
  - (i) $\alpha_2 \geq 1 \supset \alpha_2 \geq 2$
  - (ii) $\alpha_2 \geq 1 \supset \neg(\alpha_2 \geq 2)$
- (c) neither *true*, so fork.

Case $\neg(\alpha_2 \geq 2)$:

4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \neg(\alpha_2 \geq 2)$ (or simply $\alpha_2 = 1$)

8 this case completed. (returns α_1 when $\alpha_2 = 1$.)

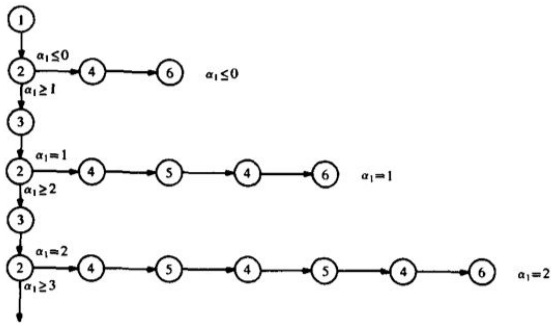
Case $\alpha_2 \geq 2$:

4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \alpha_2 \geq 2$ (or simply $\alpha_2 \geq 2$)
⋮	⋮				

In this example the symbolic execution will continue indefinitely

TWOLOOPS: this is interesting because the first loop essentially concretizes N, so the following loop does not result in a fork. They added DO semantics on the fly--could be done with GOTO and IFs.

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



Communitivity:

Instantiation: for all leaf nodes in a tree, replace symbolics in the pc with the associated values we would like to evaluate for. The results are the values in the terminal node who's pc evaluates to true. These nodes return identical results to actually running the program.

Implementation:

ASSUME statements: like gdb's set, but for constraints over the pc.

The power of the system comes down to the underlying formula simplification manipulation and evaluation capabilities.

Example:

Binary search from L to U (inclusive), shows how symbolics converge. Note: array must not be symbolic, there is a difference between a variable with symbolic assignment, and an undefined number of variables.

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

```
10.              THEN U = J - 1
11.              ELSE L = J + 1
12.     END;
13.    IF FOUND = 0 THEN J = L - 1
14. END
```

Finite subtree determined by an array with elements 1 to 5 has 11 nodes.

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

Unconstrained array bounds show shrinking bounds. SEARCH(A, “L”, “U”, “X”, FOUND, J)

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

Program Correctness, Proofs, and Symbolic Execution

Input and output predicates define the correct behavior of the program. Verify that for all inputs that satisfy the input predicate, the resulting outputs (if anything) satisfy the output predicate.

These connect the predicates to the program:
ASSUME: imposes constraints on symbolic values (the constraint is added to the pc)
PROVE(B): asks if $pc \supset$ B
ASSERT: may be a PROVE or ASSERT depending on context
Inductive predicates: the proof is broken down into inductive predicates, which are placed such that the program is broken into segments of finite length. These allow the proof of correctness to be broken down into a proof of finitely many finite paths. These are expressed with ASSERTS.

Strategy employing symbolic execution: for every adjacent pair of ASSERTS in the program, change the first to an ASSUME, the latter to a PROVE, and symbolically execute the finite length code between, to verify that the PROVE holds, given the assume (pc is initialized to true, program variables are all symbolic).