Symbolic Execution

Lecture 23

Dr. Ichul Yoon (icyoon@sung.korea.ac.kr)

Sources: Foundations of Software Testing (textbook)

Symbolic Execution

● Insight
   - *Generate test cases by symbolically executing the code*

   ● In specific,
     - Use symbols to represent input values (eventually mapped to variables) - e.g., \( \alpha_1, \alpha_2, \ldots \)
     - Execute a program symbolically for a set of classes of input symbols.
     - A test with a set of specific input values is a special case.

Symbolic Execution: An Example

● A simple statement sequence

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

   - If test input is \( \{a=1, b=3, c=5\} \), the outcome is 9, and this represent only one test case
   - In symbolic execution, the symbolic execution result of PROCEDURE \( \alpha_1, \alpha_2, \alpha_3 \) is \( \alpha_1 + \alpha_2 + \alpha_3 \)

Testing vs. Program Proving

● Testing
   - Program quality assurance is based on running the input samples and verifying the outcomes.
   - No guarantee for the inputs not in the sample

● Program proving
   - Prove the conformance to the requirement without being required to execute the program
   - Use precise specification of the correct program behavior, followed by formal proof procedure
   - The accuracy of the specification and the proof steps has to be guaranteed.
Symbolic Execution: Details

- The language syntax and the individual programs written in the language need not be changed
  - Symbolic data objects are used only as inputs to the program
- Evaluation rules for arithmetic expressions in assignment and branch statement must be extended to handle symbolic values
  - Assignment statement
    - Evaluate the RHS of the statement to a polynomial over symbols
  - Branch statement
    - Requires path condition (pc) – the accumulator of properties (a Boolean expression over the symbolic inputs \( \{\alpha_i\} \))
    - Never contains program variables

Symbolic Execution: Details

- Path condition
  - A conjoined list of expressions of the form: \( R \geq 0 \) or \( \neg R \geq 0 \), where \( R \) is a polynomial over \( \{\alpha_i\} \)
  - Example: \( m \geq 0 \land m + 2 \times m \geq 0 \land \neg (m \geq 0) \)
- Initialized to true
- Consider two situations
  - \( pc \Rightarrow q \) (\( q \) is a condition expression)
  - \( pc \Rightarrow \neg q \)

Symbolic Execution: Details

- Branch statement
  - Non-forking execution
    - If \( pc \Rightarrow q \), execute THEN path
    - If \( pc \Rightarrow \neg q \), execute ELSE path
  - Forking execution
    - Neither expressions are true and each alternative is possible
    - Explore both control paths – fork into two parallel executions
    - For the THEN path, the inputs are assumed to satisfy \( q \).
    - So, update pc to \( pc \land q \)
    - Similarly for the ELSE path update pc to \( pc \land \neg q \).

Symbolic Execution Tree

- An "execution tree" characterizing the execution paths followed during the symbolic execution of a procedure.
- Build procedure
  - Associate a node with:
    - Each statement executed
    - Each transition between statements a directed arc connecting the associated nodes
  - For each forking IF statement execution, the associated node has two arcs leaving the node:
    - Labeled "T" and "F" for the true and false path, respectively
  - Associate the complete current execution state, i.e. variable values, statement counter, and pc with each node.
Symbolic Execution Tree: Example

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

Symbolic Execution Tree: Example

- Symbolic Execution Tree

  - Properties
    - For each terminal leaf in the tree, there does exist a particular non-symbolic input to the program
    - pc's associated with any two terminal leaves are distinct.
      - At forking node, some q was added to one pc while ¬q was added to the other.
    - Commutativity
      - The operation of instantiating the symbols \( \alpha_i \) with specific integers, say \( \{j_i\} \), and the operation of executing the program are interchangeable.

Symbolic Execution Tree

- An Interactive Symbolic Executer – EFFIGY
  - Debugging
    - Supported tracing, breakpoints, and state saving
    - Symbolic execution where specific input values are mixed with symbols
    - Systematically explore execution tree, interacting with users
  - Testing
    - Check test case results against output assertions if they are supplied
    - Program verifier which uses symbolic execution and user supplied assertions to generate the verification conditions
Program Correctness and Symbolic Exec.

- To prove the correctness of a program,
  - Supply an "input predicate" and an "output predicate" with the program. Define the "correct" program behavior.
  - The program is correct if for all inputs which satisfy the input predicate, the results produced by the program satisfy the output predicate.

- To prove the correctness of each path
  - ASSUME(B) at the beginning of the path
  - Execute the path symbolically
  - PROVE(B) at the end of the path

Dynamic Symbolic Execution Techniques

- Capability of executing concretely and symbolically in a mixed fashion.

- Let's see another example of symbolic execution

  ```plaintext
  1  int twice (int x) {
  2      return 2*x;
  3  }
  4  
  5  void main (int x, int y) {
  6      if (x > y) {
  7        if (x >= y) {
  8          if (x >= y) {
  9            if (x >= y) {
 10              if (x >= y) {
 11              }
 12          }
 13          }
 14          }
 15        }
 16      };
  17      y = sym.input();
  18      main(y, y);
  19      return 0;
  20  }
  ```

Dynamic Symbolic Execution Techniques

- A key disadvantage of classical symbolic execution
  - It cannot generate an input if the symbolic path constraint along an execution path contains formulas that cannot be (efficiently) solved by a constraint solver.

  - Variant 1 – modify `twice` function to return `(y, y, y) % 50`.
    - Symbolic execution will generate the path constraints `x0 ≠ (y, y, y) % 50` and `x0 = (y, y, y) % 50`. After the execution of the first conditional statement.

  - Variant 2 – assume that the function `twice` is not available.
    - Symbolic execution will try to generate the path constraints `x0 ≠ twice(y0)` and `x0 = twice(y0)`. But, the constraint solver will fail to solve any of these constraints, because the function is not available.
Concolic Testing

- Perform symbolic execution dynamically, while the original program is executed on concrete input values.
  - Maintain a concrete state and a symbolic state
    - Concrete state maps all variables to their concrete values; Needs initial concrete values for its inputs. (random is an option)
    - Symbolic state only maps variables that have non-concrete values.
- Execute a program starting with initial input values and gather symbolic constraints at conditional statements
- Use a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative execution path.
- Repeat until all execution paths are explored, a user-defined coverage criteria is met.

Example

Start with some random input, say \( \{x = 22, y = 7\} \)

Execute the program both concretely and symbolically.

At line 7, the concrete execution will take the "else" branch. The symbolic execution will generate the path constraint \( x_0 \neq 2y_0 \) along the concrete execution path.

Concolic testing negates a conjunct in the path constraint and solves \( x_0 = 2y_0 \) to get the test input \( \{x = 2, y = 1\} \).

New input will force the program execution along a different execution path.

Example (cont.)

Next, concolic testing performs symbolic execution along the concrete execution and

Generates the path constraint \( x_0 = 2y_0 \) \( \land \) \( x_0 \leq y_0 + 10 \).

Concolic testing will generate a new test input by negating the conjunct \( x_0 \leq y_0 + 10 \) – i.e., solve the constraint \( x_0 = 2y_0 \) \( \land \) \( x_0 > y_0 + 10 \) to get test input (e.g., \( \{x = 30, y = 15\} \)).

Execution-Generated Testing

- Make a distinction between the concrete and symbolic state of a program.
  - Dynamically check before every operation if the values involved are all concrete. If so, the operation is executed just as in the original program.
  - Otherwise, the operation is performed symbolically, by updating the path condition for the current path.
Execution-Generated Testing

Example
- If line 17 is changed to $y = 10$, then line 6 will simply call function twice() with the concrete argument 20.
  - The function twice will be executed as in the original program.
    - This is common in real software (e.g., calling libraries that are not instrumented for symbolic execution, or issuing OS system calls.)
  - The branch on line 7 will become if ($20 == x$), and forking symbolic execution will follow by adding constraint ($x = 20$) or ($x \neq 20$).