Program Analysis, University of Stuttgart, Winter 2020/21

Exercise 2: Symbolic and Concolic Execution

Deadline for uploading solutions via Ilias:
January 15, 2021, 11:59pm Stuttgart time

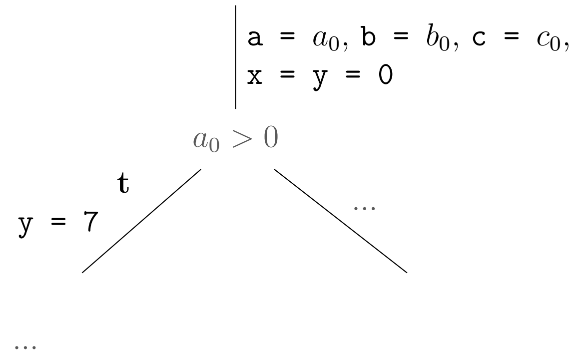
Task 1 Symbolic Execution [33 points]

For this task, you are given the following JavaScript program.

```
function foo(a, b, c) {
  let x = y = 0;

  if (a > 0) {
    y = 7;
  } else {
    y = 5;
  }

  if (b > 3) {
    x = 2;

    if (a <= 0 && c == 0) {
      y = y - 3;
    }
  }

  assert(x + y != 5);
}
```

Subtask 1.1 Program Execution Tree [12 points]

Complete the program execution tree (PET) started below, as demonstrated in the lecture. That is, nodes are conditionals and edges correspond to sequences of non-conditional statements. Mark *true* branches with a **t** and *false* branches with an **f**. Also write the assignments that were performed next to each edge. Please use a small square □ to signify when an execution ends (i.e., for leaf nodes of the PET).



Subtask 1.2 Path Conditions [9 points]

Each of the leaves in the program execution tree above corresponds to a unique path through the program. For each such unique path, there is a path condition, i.e., a logical formula that must be satisfied for the program to take this path. Collect the path condition for each leaf in your program execution tree. Write down all those path conditions below by listing them from left to right in the tree. Use syntax from mathematics for logical connectives (i.e., $\wedge$, $\neg$, etc.) to make clear that these are logical formulas. You are allowed but not required to simplify the formulas.

1. ____
2. ____
3. ____
4. ____
5. ____
6. ____

Subtask 1.3 Solve for Inputs [9 points]

An SMT solver (and you as a human) can solve path conditions, i.e., try to find an assignment of a_0, b_0, and c_0 that satisfies the formula. Write down one possible solution (there may be infinitely many) per path condition, i.e., write down values for a_0, b_0, and c_0. If no values of a_0, b_0, and c_0 can satisfy the formula, write down **UNSAT**.

1. ____
2. ____
3. ____
4. ____
5. ____
6. ____

Subtask 1.4 Assertion [3 points]

For which path does the assertion in line 18 fail, i.e., `x + y != 5` evaluates to false? Write down the path condition: ________________________.

What are the values of `x` and `y` at the assert in that case? `x` = _______, `y` = _______.

Task 2 Concolic Testing [38 points]

In this task, you will perform concolic testing on the following JavaScript program. Assume all variables are integers.

```
1  function bar(x, y) {
2    // Program state?
3    if (x > y) {
4      x = 3;
5    } else {
6      y = 3;
7    }
8    // Program state?
9    y = y * 2;
10   // Program state?
11   x = baz(x, y);
12   // Program state?
13   if (x < 0) {
14     y = y - 1;
15   }
16   // Program state?
17   assert(x + y == 0);
18 }
19 function baz(a, b) {
20   return a - b;
21 }
```

Subtask 2.1 First Concolic Execution [12 points]

We start executing the program by calling function `bar` with seed inputs `x` $= 1$ and `y` $= 2$. Complete the table below with the values of the variables `x` and `y` for the concrete and symbolic execution of the program. Write those down at each program line given in the first column. After the first branch, also write down the path condition under which the program has executed along this path.

At Line	Concrete Execution	Symbolic Execution	Path condition
2	`x` $= 1$, `y` $= 2$	`x` $= x_0$, `y` $= y_0$	N/A
8			
10			
12			
16			

Subtask 2.2 Generating New Inputs [3 points]

Since concolic execution is a test generation technique, our next goal is to generate a new set of inputs that leads the program down a different path than in the previous execution. For that, take the path condition from the previous execution and negate the conjunct that corresponds to the branch at line 13 in the program.

This results in the following path condition: ________________________________

Please solve this path condition for x_0 and y_0 to get test inputs for the program that take a new path (there are infinitely many correct solutions): $x_0 =$ _______, $y_0 =$ _______.

Subtask 2.3 Second Concolic Execution [15 points]

We now execute the program a second time, taking the new concrete values for `x` and `y` that you generated in the previous subtask as concrete inputs. Please fill the table below again with the concrete and symbolic state of the program and the path condition.

At Line	Concrete Execution	Symbolic Execution	Path condition
2			N/A
8			
10			
12			
16			

At the end of this second execution, does the assertion in line 17 fail? __________

Subtask 2.4 Generate More Inputs [8 points]

Concolic execution can successively generate more inputs for `x` and `y` until ultimately each path through the program has been taken once.

Please list two more solutions for `x` and `y` that each cover a new path through the program. For each pair of inputs, also list the final path condition when the program executes on these inputs.

1. `x` = _______, `y` = _______, path condition: ________________________________
2. `x` = _______, `y` = _______, path condition: ________________________________

Task 3 Limitations and Understanding [19 points]

In this task, you should demonstrate a deeper understanding of symbolic and concolic execution by explaining some examples that potentially cause problems. For each of the JavaScript programs below, answer the question in prose with at most 80 words per question. Please write legibly. Assume the argument `x` is an integer in every example.

Subtask 3.1 [4 points]

```
function f(x) {
  let r = fs.readFileSync("test.txt", "utf8");
  if (r == "some string") {
    x = 0;
  }
  assert(x == 0);
}
```

Assume `fs` is the file system API from Node.js. Why is this example program difficult for symbolic execution? What are possible solutions in symbolic execution frameworks?

How does concolic execution handle this issue?

Subtask 3.2 **[7 points]**

```
function f(x) {
  let s = 0;

  let i = 0;
  while (i < x) {
    s = s + i*i;
  }

  assert(s > 0);
}
```

Why is this program problematic in general (Hint: The problem is relevant irrespective of symbolic/concolic execution)?

What issue can arise when symbolically executing the program?

Is the same issue present for concolic execution? What happens when performing concolic execution of the program?

Are there still challenges for symbolic execution, when a statement `i++;` is added after line 6? If so, explain why.

Subtask 3.3 [4 points]

```
function f(x) {
  let y = randomInt();
  if (x > y) {
    x = 0;
  }
  assert(x != 0);
}
```

Assume concolic testing executes the program with seed input `x` = 1 and `randomInt()` returns 42 this time, and a pseudo-random integer in general. Assume `x` = 43 is generated as the next input. Which issue could arise in the next concolic execution? What is the proper term for this behavior?

Besides random number generators, can there be other causes for this issue? Name an example if so, explain why if not.

Subtask 3.4 [4 points]

```
function f(x) {
  let h = SHA256(x);
  if (h == 0xDEADBEEF) {
    assert(false);
  }
}
```

Assume SHA256 computes the cryptographic SHA-256 hash function on its input. Why is this program difficult for symbolic and concolic execution?

____

____

____

____

What output can SMT and SAT solvers produce, besides a model (i.e., an assignments of values to variables that satisfies the given formula) and **UNSAT**?

____

Task 4 Multiple Choice [10 points]

Which of the following statement(s) is true? Check the correct statement(s).

- ☐ Symbolic execution is a static program analysis technique.
- ☐ Concolic execution is a static program analysis technique.
- ☐ Symbolic execution is a dynamic program analysis technique.
- ☐ Concolic execution is a dynamic program analysis technique.
- ☐ Path explosion can be an issue for symbolic execution.
- ☐ Path explosion can be an issue for concolic execution.
- ☐ SAT solvers can solve formulas involving floating point numbers.
- ☐ SMT solvers can solve every conditional in a program.
- ☐ Symbolic execution requires more "heavy-weight" program analysis than fuzzing.
- ☐ Symbolic execution is not deployed in practice because it is too "heavy-weight".