Homework 7: Symbolic and Concolic Execution

17-355/17-665/17-819: Program Analysis
Claire Le Goues[*]
clegoues@cs.cmu.edu

Due: Thursday, April 2 11:59 pm

100 points total

Assignment Objectives:

- Demonstrate understanding of specifications for verifying programs.
- Understand soundness criteria for substituting subexpressions of a path condition with concrete values in concolic execution
- Understand Symbolic Execution and Implement Forward Verification Condition Generation for symbolically executed paths.

Setup. Clone the starter repository here: `https://classroom.github.com/a/EMQiv2z8`

Handin Instructions. Please submit your assignment via Gradescope by pointing it to your GitHub repository by the due date. You should replace `written-answers.pdf` with your written answers to questions 1 and 2 and update the `signed*.py` files to complete question 3.

Note: *We will not look at your homework repository directly, but will only see what you have submitted to Gradescope!* ***Make sure that you (re)submit after you have completed all parts; Gradescope does not automatically pull new commits from GitHub.***

Question 1, Verifying with Dafny, *(30 points).* Dafny is a programming language with built-in specification constructs. For example, Dafny lets you specify pre- and post conditions on methods, and will verify that your code meets the specification. Underneath the hood, Dafny discharges SMT formulas based on the program and specifications, and validates correctness using, e.g., Z3. The online tutorial for Dafny is a good resource for examples and getting started: `https://rise4fun.com/Dafny/tutorial/Guide`. *Note, however, that you do not need to write or understand much Dafny to complete this question, which primarily concerns specification/verification.*

Consider the bubble sort program written in Dafny in Figure 1 (also at `https://rise4fun.com/Dafny/1xSS` and in the `bubblesort.dfy` file in the homework repository). By writing specifications in Dafny, we can verify the correctness of bubble sort (i.e., that it always returns a sorted list). Take some time to understand the program and the existing specifications, then answer the following questions.

[*]This homework was developed together with Jonathan Aldrich

```
predicate sorted(a: array?<int>, l: int, u: int)
  reads a
  requires a ≠ null
{
  ∀ i, j • 0 ≤ l ≤ i ≤ j ≤ u < a.Length ⟹  __FIXME__
}

predicate partitioned(a: array?<int>, i: int)
  reads a
  requires a ≠ null
{
  ∀ k, k’ • 0 ≤ k ≤ i < k’ < a.Length ⟹ a[k] ≤ a[k’]
}

method BubbleSort(a: array?<int>)
  modifies a
  requires a ≠ null
  ensures sorted(a, 0, a.Length-1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant 0 < i < a.Length
    invariant sorted(a, i, a.Length-1)
    invariant partitioned(a, i)
  {
    var j := 0;
    while (j < i)
      invariant 0 < i < a.Length ∧ 0 ≤ j ≤ i
      invariant sorted(a, i, a.Length-1)
      invariant partitioned(a, i)
      invariant ∀ k • 0 ≤ k ≤ j ⟹ a[k] ≤ a[j]
    {
      if(a[j] > a[j+1])
      {
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i -1;
  }
}

method Main() {
  var a := new int[5];
  a[0], a[1], a[2], a[3], a[4] := 9, 4, 6, 3, 8;
  BubbleSort(a);
  var k := 0;
  while(k < 5) { print a[k], "\n"; k := k + 1; }
}
```

Figure 1: Incomplete Dafny bubble sort.

a) (5 points) The predicate `sorted` is incomplete. What should be substituted for `__FIXME__` on line 5?

b) (10 points) After adding the condition for part *a)*, run Dafny again. Dafny still unable to prove the program correct due to a loop invariant. It gives two errors: `This loop invariant might not hold on entry` and `This loop invariant might not be maintained by the loop.`

Correct the reported loop invariant so that Dafny no longer reports the case where the `loop invariant might not be maintained by the loop`. Write out the code/invariant you changed in your assignment, and explain in prose why the original loop invariant was insufficient.

c) (15 points) After fixing the loop invariant in part *b)*, Dafny still reports that the correct loop invariant `might not hold on entry`. Explain in prose why this is the case.

Dafny will verify the complete implementation with some changes that deal with the condition on loop entry. One way is to add an additional invariant. Another way is to change the program so that Dafny infers stronger conditions on variable(s).

Either add a single invariant *or* make a small change the program so that Dafny verifies the program. Rerun Dafny and confirm that it verifies the program with no warnings. Describe the change you made. both?

Question 2, Concolic execution soundness, *(20 points)*. In class (and in the notes) we saw an example of a path condition g and a sound concolic replacement g' for it. In particular, g was $x_0 == (y_0 * y_0) \ \% \ 50$ after negation and g' was $x_0 == 49$ after negation. This is trivially sound because the only solution is $x_0 == 49$, which when extended with $y_0 == 7$ from the original test case yields a new test input that fulfills the original path condition $x_0 == (y_0 * y_0) \ \% \ 50$.

- Give an example path condition g, test input M, and concolic path condition g' resulting from replacing a subexpression a_s of g with a concrete value $n = [M]a_s$, such that g' is *unsound*.
- Witness the unsoundness by also providing a test input M' that satisfies g' but not g.
- Give a condition on g, M, g' and/or a_s that is sufficient to ensure that g' is sound.

Question 3, Forward VCGen with Symbolic Execution, *(50 points)*.

For this task, you will implement per-path verification condition (VC) generation to prove whether a variable is possibly negative along all program paths, similar to sign analysis in previous homework.

Typical Symbolic Executors implement rules to emit verification conditions based on a program grammar (c.f., Symbolic Execution notes). In this task, we will take a shortcut and only consider concrete programs, instead of a full grammar. Your job is to manually instrument statements to generate and collect verification conditions for Python programs, just like a real symbolic executor would.

Example. Figure 2 shows a small function `signed`. We care about two variables: the input variable `x` and a local variable `y`. We want to check that `y` is nonnegative along all paths, such that it is safe for C-like array access. We introduce two symbolic variables to track the values of `x` and `y`: `x0` and `y0` on Lines 1 and 2.

Some example verification constraints have been added in red. The first constraint is simply satisfiable (Line 5). The final constraint checks whether `y0` is negative is satisfiable. If the solver finds a satisfying model where `y0` is negative, we say that an error occurs for that path. On the other hand, if the solver finds that `y0 < 0` is UNSAT, the path is safe.

VC generation has been added for the path taken by the `if` statement on Line 7. For example, Line 8 conjuncts the constraint `x0 < 0` with the `current_VC`, corresponding to the if-condition. Line 9 further updates the `current_VC` to account for the assignment `y=x`.

```
 0  x0 = Int('x0')
 1  y0 = Int('y0')
 2
 3  def signed(x):
 4    current_VC = True
 5
 6    if (x < 0):
 7        current_VC = And(current_VC, x0 < 0)
 8        y = x
 9        current_VC = And(current_VC, y0 == x0)
10    else:
11        # FIXME: add sound verification conditions to make the test pass
12        y = x
13
14    # y must be nonnegative
15    current_VC = And(current_VC, y0 < 0)
16
17    # Check: one path is safe, the other is unsafe.
```

Figure 2: Simplified `signed.py`

This path is SAT, implying it is unsafe since `y0 < 0`. However, the path on the `else` branch is in fact safe (and emitting the correct constraints should result in the solver saying UNSAT). Unfortunately, because there are no constraints generated for this path yet, we can't tell that it's safe: the solver emits SAT. Your task is to implement the missing VC generation for this branch, as well as the other programs in the homework repository, so that the signedness for `y` is tracked correctly on each path (which will make the tests pass). You should add VC generation for each statement and branch conditional. See the `README.md` file in the for more details.

Setup and Test. To test that your verification condition solution is correct, we need a way to execute along all of the paths. To do that, we're in fact going to *use* an existing Python Symbolic Executor, PyExZ3.[1] Follow the installation instructions for PyExZ3 in the homework repository `README.md`. You can then run, for example, `python3 pyexz3.py signed.py` on the test programs.

Submission. Update the `signed*.py` files in the homework repository and commit the changes. Note that passing the tests is a necessary, but not sufficient condition for credit: you must implement sound (and tight) verification condition generation for the example programs. *Make sure that you submit your repository to Gradescope when you are done.*

[1]Of course, PyExZ3 is going to generate its own verification conditions internally so that it can execute all paths for the Python programs, like the one in Figure 2. Neat huh?