# Homework 9: Symbolic and Concolic Execution

17-355/17-665/17-819O: Program Analysis
Claire Le Goues and Jonathan Aldrich
`clegoues@cs.cmu.edu, aldrich@cs.cmu.edu`

Due: Thursday, April 12 11:59 pm

100 points total

**Assignment Objectives:**

- 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.

**Handin Instructions.** Please submit the written assignment Canvas as a **PDF** by the due date. Name it **[your-andrew-id]-hw9.pdf**. Submit your solution to **Q2** in a folder called **hw9** in your GitHub repository.

**Question 1**, Concolic Path Condition Soundness, *(50 points)*.

Concolic execution is motivated by the presence of subexpressions within a path condition that are difficult for a SMT solver to reason about. The key idea of concolic execution is to replace these subexpressions with appropriate concrete values.

Where possible, we would like this replacement to be sound. Intuitively, the replacment is sound if any solution to the new path condition is also a solution to the old one. This means that even after the substitution, concolic execution will successfully drive the program down the desired path. Let's make this idea more formal.

Let $g$ be a negated path condition as defined in the lecture notes. Let $M$ be a map from symbolic constants $\alpha$ to integers $n$. We write $[M]g$ for the boolean expression we get by substituting all the symbolic constants in $g$ with the corresponding integer values given in $M$; this is only defined if the free symbolic constants $FC(g)$ are the same as *domain*$(M)$. We define $[M]a_s$ similarly for substitution of symbolic constants with values in arithmetic expressions.

Given $g$ and a map $M$ that represents the inputs to a concrete test case execution, concolic execution may replace a subexpression $a_s$ of $g$ with the concrete value $n$ achieved in testing. Note that $n = [M]a_s$. Let the new guard be $g' = [n/a_s]g$ (again, we consider this *after* negating the last constraint in the path).

We say that $g'$ is a *sound* concolic path condition if for all alternative test inputs $M'$ such that $[M']g'$ is true, we have $[extend(M', M)]g$ true. Here, the $extend$ function extends the symbolic constants in $M'$ with any that are necessary to match the domain of $M$. More precisely, $\forall \alpha' \in domain(M'), extend(M', M)[\alpha'] = M'[\alpha']$ and $\forall \alpha \in (domain(M) - domain(M')), extend(M', M)[\alpha] = M[\alpha]$.

The notes gave 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 $x_0 == (y_0 * y_0)\%50$.

*a) (10 points)* 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 $n = [M]a_s$, wuch that $g'$ is *unsound*.

*b) (10 points)* Witness the unsoundness by also providing a test input $M'$ that satisfies $g'$ but not $g$.

*c) (10 points)* Give a condition on $g, M, g'$ and/or $a_s$ that is sufficient to ensure that $g'$ is sound.

*d) (20 points)* Prove that your condition is sufficient for soundness.

**Question 2**, 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 1 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`.

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 this folder, 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 online README 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 Install Instructions for PyExZ3, and run `python3 run_tests.py`

[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 1. Neat huh?

```python
x0 = Int('x0')
y0 = Int('y0')

def signed(x):
  current_VC = True

  if (x < 0):
      current_VC = And(current_VC, x0 < 0)
      y = x
      current_VC = And(current_VC, y0 == x0)
  else:
      # FIXME: add sound verification conditions to make the test pass
      y = x

  # y must be nonnegative
  current_VC = And(current_VC, y0 < 0)

  # Check: one path is safe, the other is unsafe.
```

Figure 1: Simplified `signed.py`

`test` to make sure it works. You can then run, for example, `python3 pyexz3.py signed.py` on the test programs.

**Submission.** Copy the test files into your **hw9** directory, and modify them so that they pass the test when run with `python3 pyexz3.py <filename>.py`. 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.