Lecture Notes: Concolic Testing

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues*
clegoues@cs.cmu.edu

1 Introduction

We have discussed symbolic execution from two perspectives: as a method for forward verification condition generation, as well as a method that generalizes testing. We will continue to focus on this latter perspective by discussing key approaches that have allowed symbolic execution to find real bugs in practice.

1.1 Motivation

Companies today spend a huge amount of time and energy testing software to determine whether it does the right thing, and to find and then eliminate bugs. A major challenge is writing adequate test cases that cover all of the source code, as well as finding inputs that lead to difficult-to-trigger corner case defects.

Symbolic execution is a promising approach to exploring different execution paths through programs. However, it has significant limitations. For paths that are long and involve many conditions, SMT solvers may not be able to find satisfying assignments to variables that lead to a test case that follows that path. Other paths may be short but involve computations that are outside the capabilities of the solver, such as non-linear arithmetic or cryptographic functions. For example, consider the following function:

```
testme(int x, int y){
    if(bbox(x)==y){
        ERROR;
    } else {
        // OK
    }
}
```

If we assume that the implementation of `bbox` is unavailable, or is too complicated for a theorem prover to reason about, then symbolic execution may not be able to determine whether the error is reachable.

1.2 Statically modeling functions

We have several options for symbolically executing a program with functions (like the one we developed for interprocedural dataflow analysis). Inlining is somewhat more practical here as we

*These notes were developed together with Jonathan Aldrich are not computing fixpoints. We can also simply symbolically execute the called methods, too; because we are not joining abstract state over multiple possible paths, we do not immediately lose precision as we would in interprocedural abstract interpretation.

If we continue to operate in a language with pre and postconditions specified at the function level (as we assumed in Hoare-Style verification), we can also use those to model function behavior statically. Assuming pre- and post-conditions encoded in the same expression language as guards, $e_{\texttt{pre}}$ and $e_{\texttt{post}}$:

$$\frac{\langle e_{\texttt{post}}, \Sigma \rangle \Downarrow a_{\texttt{post}}}{\langle g, \Sigma, \texttt{return} \rangle \Downarrow \langle a_{\texttt{post}}, \Sigma \rangle} \textit{ big-return}$$

Question: what about function calls? Note that if the language involves heap-manipulation, this question becomes more or less difficult!

At some point, however, symbolic execution will reach the "edges" of the application: a library, system, or assembly code call. For certain libraries, a simpler version is available (such as `libc` implemented for embedded systems). Other tools allow custom code models, such as the implementation of a ramdisk to model kernel fs code. This is of course very labor intensive. Even when this code can be pulled in and executed symbolically, there are times that the code is simply too complicated to be tractably reasoned about statically, such as if it involves non-linear arithmetic.

The challenges of fully statically symbolically executing all code directly motivate *concolic testing*. Concolic testing combines **conc**rete execution (i.e. testing) with symb**olic** execution.[1]

1.3 Goals

We will consider the specific goal of automatically unit testing programs to find assertion violations and run-time errors such as divide by zero. We can reduce these problems to input generation: given a statement s in program P, compute input i such that $P(i)$ executes s.[2] For example, if we have a statement `assert x > 5`, we can translate that into the code:

```
1 if (!(x > 5))
2     ERROR;
```

Now if line 2 is reachable, the assertion is violated. We can play a similar trick with run-time errors. For example, a statement involving division `x = 3 / i` can be placed under a guard:

```
if (i != 0)
    x = 3 / i;
else
    ERROR;
```

2 Concolic execution overview

In concolic execution, symbolic execution is used to solve for inputs that lead along a certain path. However, when a part of the path condition is infeasible for the SMT solver to handle, we substitute values from a test run of the program. In many cases, this allows us to make progress

[1] The word concolic is a portmanteau of **conc**rete and symb**olic**

[2] This formulation is due to Wolfram Schulte

towards covering parts of the code that we could not reach through either symbolic execution or randomly generated tests.

Consider the `testme` example from the motivating section. Although symbolic analysis cannot solve for values of `x` and `y` that allow execution to reach the error, we can generate random test cases. These random test cases are unlikely to reach the error: for each `x` there is only one `y` that will work, and random input generation is unlikely to find it. However, concolic testing can use the concrete value of `x` and the result of running `bbox(x)` in order to solve for a matching `y` value. Running the code with the original `x` and the solution for `y` results in a test case that reaches the error.

In order to understand how concolic testing works in detail, consider a more realistic and more complete example:

```
int double (int v) {
    return 2*v;
}

void bar(int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
              ERROR;
        }
    }
}
```

We want to test the function `bar`. We start with random inputs such as $x = 22, y = 7$. We then run the test case and look at the path that is taken by execution: in this case, we compute $z = 14$ and skip the outer conditional. We then execute symbolically along this path. Given inputs $x = x_0, y = y_0$, we discover that at the end of execution $z = 2 * y_0$, and we come up with a path condition $2 * y_0 \neq x_0$.

In order to reach other statements in the program, the concolic execution engine picks a branch to reverse. In this case there is only one branch touched by the current execution path; this is the branch that produced the path condition above. We negate the path condition to get $2 * y_0 == x_0$ and ask the SMT solver to give us a satisfying solution.

Assume the SMT solver produces the solution $x_0 = 2, y_0 = 1$. We run the code with that input. This time the first branch is taken but the second one is not. Symbolic execution returns the same end result, but this time produces a path condition $2 * y_0 == x_0 \wedge x_0 \leqslant y_0 + 10$.

Now to explore a different path we could reverse either test, but we've already explored the path that involves negating the first condition. So in order to explore new code, the concolic execution engine negates the condition from the second `if` statement, leaving the first as-is. We hand the formula $2 * y_0 == x_0 \wedge x_0 > y_0 + 10$ to an SMT solver, which produces a solution $x_0 = 30, y_0 = 15$. This input leads to the error.

The example above involves no problematic SMT formulas, so regular symbolic execution would suffice. The following example illustrates a variant of the example in which concolic execution is essential:

```
int foo(int v) {
    return v*v%50;
}

```

```
void baz(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
               ERROR;
        }
    }
}
```

Although the code to be tested in `baz` is almost the same as `bar` above, the problem is more difficult because of the non-linear arithmetic and the modulus operator in `foo`. If we take the same two initial inputs, $x = 22, y = 7$, symbolic execution gives us the formula $z = (y_0 * y_0)\%50$, and the path condition is $x_0 \neq (y_0 * y_0)\%50$. This formula is not linear in the input y_0, and so it may defeat the SMT solver.

We can address the issue by treating `foo`, the function that includes nonlinear computation, concretely instead of symbolically. In the symbolic state we now get $z = foo(y_0)$, and for $y_0 = 7$ we have $z = 49$. The path condition becaomse $foo(y_0) \neq x_0$, and when we negate this we get $foo(y_0) == x_0$, or $49 == x_0$. This is trivially solvable with $x_0 == 49$. We leave $y_0 = 7$ as before; this is the best choice because y_0 is an input to $foo(y_0)$ so if we change it, then setting $x_0 = 49$ may not lead to taking the first conditional. In this case, the new test case of $x = 49, y = 7$ finds the error.

3 Implementation

Ball and Daniel [?] give the following pseudocode for concolic execution (which they call dynamic symbolic execution):

```
i = an input to program P
while defined(i):
   p = path covered by execution P(i)
   cond = pathCondition(p)
   s = SMT(Not(cond))
   i = s.model()
```

Broadly, this just systematizes the approach illustrated in the previous section. However, a number of details are worth noting:

First, when negating the path condition, there is a choice about how to do it. As discussed above, the usual approach is to put the path conditions in the order in which they were generated by symbolic execution. The concolic execution engine may target a particular region of code for execution. It finds the first branch for which the path to that region diverges from the current test case. The path conditions are left unchanged up to this branch, but the condition for this branch is negated. Any conditions beyond the branch under consideration are simply omitted. With this approach, the solution provided by the SMT solver will result in execution reaching the branch and then taking it in the opposite direction, leading execution closer to the targeted region of code.

Second, when generating the path condition, the concolic execution engine may choose to replace some expressions with constants taken from the run of the test case, rather than treating those expressions symbolically. These expressions can be chosen for one of several reasons. First, we may choose formulas that are difficult to invert, such as non-linear arithmetic or cryptographic hash functions. Second, we may choose code that is highly complex, leading to formulas that are too large to solve efficiently. Third, we may decide that some code is not important to test, such as low-level libraries that the code we are writing depends on. While sometimes these libraries could be analyzable, when they add no value to the testing process, they simply make the formulas harder to solve than they are when the libraries are analyzed using concrete data.

4 Concolic Path Condition Soundness

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, a replacement 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. Let M be a map from symbolic constants α 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 $\forall \alpha \in (domain(M) - domain(M')), extend(M', M)[\alpha] = M[\alpha]$.

In class 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 $x_0 == (y_0 * y_0)\%50$.

As an exercise:

- 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.
- Prove that your condition is sufficient for soundness.

5 Acknowledgments

The structure of these notes and the examples are adapted from a presentation by Koushik Sen.