Lecture Notes: Symbolic Execution

17-355/17-665/17-819: Program Analysis (Spring 2019)

Jonathan Aldrich*
aldrich@cs.cmu.edu

1 Symbolic Execution Overview

Symbolic execution is a way of executing a program abstractly, so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code. The execution treats these inputs symbolically, "returning" a result that is expressed in terms of symbolic constants that represent those input values.

Symbolic execution is less general than abstract interpretation, because it doesn't explore all paths through the program. However, symbolic execution can often avoid approximating in places where AI must approximate in order to ensure analysis termination. This means that symbolic execution can avoid giving false warnings; any error found by symbolic execution represents a real, feasible path through the program, and (as we will see) can be witnessed with a test case that illustrates the error.

1.1 A Generalization of Testing

As the above discussion suggests, symbolic execution is a way to generalize testing. A test involves executing a program concretely on one specific input, and checking the results. In contrast, symbolic execution considers how the program executes abstractly on a family of related inputs. Consider the following code example, where a, b, and c are user-provided inputs:

```
1 int x=0, y=0, z=0;
2 if(a) {
3   x = -2;
4 }
5 if (b < 5) {
6   if (!a && c) { y = 1; }
7   z = 2;
8 }
9 assert(x + y + z != 3);
```

Running this code with a = 1, b = 2, and c = 1 causes the assertion to fail, and if we are good (or lucky) testers, we can stumble upon this combination and generalize to the combination of input spaces that will lead to it (and hopefully fix it!).

Instead of executing the code on concrete inputs (like a = 1, b = 2, and c = 1), symbolic execution evaluates it on *symbolic inputs*, like $a = \alpha, b = \beta, c = \gamma$, and then tracks execution in terms of those

*These notes were developed together with Claire Le Goues

symbolic values. If a branch condition ever depends on unknown symbolic values, the symbolic execution engine simply chooses one branch to take, recording the condition on the symbolic values that would lead to that branch. After a given symbolic execution is complete, the engine may go back to the branches taken and explore other paths through the program.

To get an intuition for how symbolic analysis works, consider abstractly executing a path through the program above. As we go along the path, we will keep track of the (potentially symbolic) values of variables, and we will also track the conditions that must be true in order for us to take that path. We can write this in tabular form, showing the values of the path condition g and symbolic environment E after each line:

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta \geqslant 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
9	$\neg\alpha \wedge \beta \geqslant 5 \wedge 0 + 0 + 0 \neq 3$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$

In the example, we arbitrarily picked the path where the abstract value of a, i.e. α, is false, and the abstract value of b, i.e. β, is not less than 5. We build up a path condition out of these boolean predicates as we hit each branch in the code. The assignment to x, y, and z updates the symbolic state E with expressions for each variable; in this case we know they are all equal to 0. At line 9, we treat the assert statement like a branch. In this case, the branch expression evaluates to $0 + 0 + 0 \neq 3$ which is true, so the assertion is not violated.

Now, we can run symbolic execution again along another path. We can do this multiple times, until we explore all paths in the program (*exercise to the reader: how many paths are there in the program above?*) or we run out of time. If we continue doing this, eventually we will explore the following path:

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta < 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$
9	$\neg\alpha \wedge \beta < 5 \wedge \neg(0 + 1 + 2 \neq 3)$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$

Along this path, we have $\neg\alpha \wedge \beta < 5$. This means we assign y to 1 and z to 2, meaning that the assertion $0 + 1 + 2 \neq 3$ on line 9 is false. Symbolic execution has found an error in the program!

1.2 History of Symbolic Analysis

Symbolic execution was originally proposed in the 1970s, but it relied on automated theorem proving, and the algorithms and hardware of that period weren't ready for widespread use. With recent advances in SAT/SMT solving and 4 decades of Moore's Law applied to hardware, symbolic execution is now practical in many more situations, and is used extensively in program analysis research as well as some emerging industry tools.

2 Symbolic Execution Semantics

We can write rules for evaluating programs symbolically in WHILE. We will write the rules in a style similar to the big-step semantics we wrote before, but incorporate symbolic values and keep track of the path conditions we have taken.

We start by defining symbolic analogs for arithmetic expressions and boolean predicates. We will call symbolic predicates *guards* and use the metavariable g, as these will turn into guards for paths the symbolic evaluator explores. These analogs are the same as the ordinary versions, except that in place of variables we use symbolic constants:

$$
\begin{array}{rcl@{\qquad}rcl}
g & ::= & \text{true} & a_s & ::= & \alpha \\
 & | & \text{false} & & | & n \\
 & | & \text{not } g & & | & a_{s1}\ op_a\ a_{s2} \\
 & | & g_1\ op_b\ g_2 & & & \\
 & | & a_{s1}\ op_r\ a_{s2} & & &
\end{array}
$$

Now we generalize the notion of the environment E, so that variables refer not just to integers but to symbolic expressions:

$$E \in \mathit{Var} \to a_s$$

Now we can define big-step rules for the symbolic evaluation of expressions, resulting in symbolic expressions. Since we don't have actual values in many cases, the expressions won't evaluate, but variables will be replaced with symbolic constants:

$$\frac{}{\langle n, E\rangle \Downarrow n}\ \textit{big-int}$$

$$\frac{}{\langle x, E\rangle \Downarrow E(x)}\ \textit{big-var}$$

$$\frac{\langle a_1, E\rangle \Downarrow a_{s1} \quad \langle a_2, E\rangle \Downarrow a_{s2}}{\langle a_1 + a_2, E\rangle \Downarrow a_{s1} + a_{s2}}\ \textit{big-add}$$

We can likewise define rules for statement evaluation. These rules need to update not only the environment E, but also a path guard g:

$$\frac{}{\langle g, E, \texttt{skip}\rangle \Downarrow \langle g, E\rangle}\ \textit{big-skip}$$

$$\frac{\langle g, E, s_1\rangle \Downarrow \langle g', E'\rangle \quad \langle g', E', s_2\rangle \Downarrow \langle g'', E''\rangle}{\langle g, E, s_1; s_2\rangle \Downarrow \langle g'', E''\rangle}\ \textit{big-seq}$$

$$\frac{\langle a, E\rangle \Downarrow a_s}{\langle g, E, x := a\rangle \Downarrow \langle g, E[x \mapsto a_s]\rangle}\ \textit{big-assign}$$

$$\frac{\langle P, E\rangle \Downarrow g' \quad g \wedge g' \texttt{SAT} \quad \langle g \wedge g', E, s_1\rangle \Downarrow \langle g'', E'\rangle}{\langle g, E, \texttt{if } P \texttt{ then } s_1 \texttt{ else } s_2, \rangle \Downarrow \langle g'', E'\rangle}\ \textit{big-iftrue}$$

$$\frac{\langle P, E\rangle \Downarrow g' \quad g \wedge \neg g' \texttt{SAT} \quad \langle g \wedge \neg g', E, s_2\rangle \Downarrow \langle g'', E'\rangle}{\langle g, E, \texttt{if } P \texttt{ then } s_1 \texttt{ else } s_2, \rangle \Downarrow \langle g'', E'\rangle}\ \textit{big-iffalse}$$

The rules for skip, sequence, and assignment are compositional in the expected way, with the arithmetic expression on the right-hand side of an assignment evaluating to a symbolic expression rather than a value. The interesting rules are the ones for if. Here, we evaluate the condition to a symbolic predicate g'. In the true case, we use a SMT solver to verify that the guard is satisfiable when conjoined with the existing path condition. If that's the case, we continue by evaluating the true branch symbolically. The false case is analogous.

We leave the rule for `while` to the reader, following the principles behind the `if` rules above.

3 Heap Manipulating Programs

We can extend the idea of symbolic execution to heap-manipulating programs. Consider the following extensions to the grammar of arithmetic expressions and statements, supporting memory allocation with *malloc* as well as dereferences and stores:

$$\begin{array}{rcl} a & ::= & \ldots \mid *a \mid malloc \\ S & ::= & \ldots \mid *a := a \end{array}$$

Now we can define memories as a basic memory μ that can be extended based on stores into the heap. The memory is modeled as an array, which allows SMT solvers to reason about it using the theory of arrays:

$$m \quad ::= \quad \mu \mid m[a_s \mapsto a_s]$$

Finally, we extend symbolic expressions to include heap reads:

$$a_s \quad ::= \quad \ldots \mid m[a_s]$$

Now we can define extended version of the arithmetic expression and statement execution semantics that take (and produce, in the case of statements) a memory:

$$\frac{\alpha \notin E, m}{\langle malloc, E, m\rangle \Downarrow \alpha}\ \textit{big-malloc}$$

$$\frac{\langle a, E, m\rangle \Downarrow a_s}{\langle *a, E, m\rangle \Downarrow m[a_s]}\ \textit{big-deref}$$

$$\frac{\langle a, E, m\rangle \Downarrow a_s \quad \langle a', E, m\rangle \Downarrow a'_s}{\langle g, E, m, *a := a'\rangle \Downarrow \langle g, E, m[a_s \mapsto a'_s]\rangle}\ \textit{big-store}$$

4 Symbolic Execution Implementation and Industrial Use

Of course programs with loops have infinite numbers of paths, so exhaustive symbolic execution is not possible. Instead, tools take hueristics, such as exploring all execution trees down to a certain depth, or limiting loop iteration to a small constant, or trying to find at least one path that covers each line of code in the program. In order to avoid analyzing complex library code, symbolic executors may use an abstract model of libraries.

Symbolic execution has been used in industry for the last couple of decades. One of the most prominent examples is the use of the PREfix to find errors in C/C++ code within Microsoft [1].

References

[1] W. R. Bush, J. D. Pincus, , and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30:775–802, 2000.