Symbolic Execution Formally Explained

Overview

- The goal of the lecture is provide a formal explanation of symbolic execution in terms of a symbolic transition system and outlines its correctness

The language: expressions

$Var \coloneqq x, y, z, \dots$. A set of program variable

Ops a set of operation symbols op with their arity.

$Values$ (ranged over by v are nullary operators. We also assume to have *symbolic values*.

The set f programming expressions e is defined by the following grammar.

$$Exp \ ::= x \mid op(e_1, \dots, e_n)$$

Expressions e consist of program variables x and operators op applied to expressions.

The language: statements

$$
\begin{array}{rll}
S ::= & x := e & \text{assignment} \\
\mid & S;\ S & \text{sequential composition} \\
\mid & \mathit{if}\ b\ \{S_1\}\{S_2\} & \text{choice} \\
\mid & \mathit{while}\ b\ \{S\} & \text{iteration}
\end{array}
$$

Substitution

A **substitution** σ is a **mapping** from **variables** to **expressions**

$$\sigma : Var \to \ Exp$$

$$\sigma = \{\ x_1 = e_1, \ldots, x_k = e_k\}$$

Applying a substitution

Given an expression e, the **application** of a substitution σ, written $e\sigma$, means **replacing every occurrence** of each variable x_i in e by the corresponding expression e_i.

$$e = f(x, y) \qquad \sigma = \{x = a, y = g(b)\}$$

$$e\sigma = f(a, g(b))$$

Applying a substitution

The formal definition

$$x\sigma = \sigma(x)$$

$$op(e_1, \ldots, e_n)\sigma = op(e_1\sigma, \ldots, e_n\sigma)$$

Composing substitutions

Substitutions can be **composed**:

$$(\sigma_1 \circ \sigma_2)(x) = \big(\sigma_1(x)\big)\sigma_2$$

In logic programming , substitutions are used to **instantiate variables** so that expressions (or formulas) can **match** or become **identical**.

Composition update

$\sigma[x = e]$ is a substitution.

It is the update of the substitution σ defined as follows

$$\sigma[x = e](y) = e \quad if\ y = x$$

$$\sigma[x = e](y) = \sigma(y) \quad if\ x \neq y$$

The operational semantics

- A symbolic configuration is a triple
$$\langle S, \sigma, \phi \rangle$$
- where S denotes the statement to be executed, σ denotes the current substitution, and the logical condition ϕ denotes the path condition.

Assignment

$$\frac{}{\langle x := e, \sigma, \phi \rangle \to \langle \sigma[x = e], \phi \rangle}$$

Sequential composition

$$\frac{\langle S_1, \sigma, \phi \rangle \rightarrow \langle S', \sigma', \phi' \rangle}{\langle S_1; S_2, \sigma, \phi \rangle \ \rightarrow \langle S'; S_2, \sigma', \phi' \rangle}$$

$$\frac{\langle S_1, \sigma, \phi \rangle \rightarrow \langle \sigma', \phi' \rangle}{\langle S_1; S_2, \sigma, \phi \rangle \rightarrow \langle S_2, \sigma' \phi' \rangle}$$

Choice (conditional)

$$\frac{}{\langle if\,(b)\{S_1\}\{S_2\}, \sigma, \phi\rangle \rightarrow \langle S_1, \sigma, \phi \land b\sigma \rangle}$$

$$\frac{}{\langle if\,(b)\{S_1\}\{S_2\}, \sigma, \phi\rangle \rightarrow \langle S_2, \sigma, \phi \land \neg\, b\sigma \rangle}$$

While

$$\frac{}{\langle while(b)\ \{S\}, \sigma, \phi\rangle \rightarrow \langle S; while\ (b)\{S\}, \sigma, \phi \land b\sigma\rangle}$$

$$\frac{}{\langle while(b)\ \{S\}, \sigma, \phi\rangle \rightarrow \langle \sigma, \phi \land \neg b\sigma\rangle}$$

Correctness proof

- The formalization and the proof of correctness with respect to a concrete semantics is based on the notion of *memory* M
- The memory M is a function $M: Var \rightarrow Values$
- where $Values$ is a set of values (including the Boolean values).

A basic lemma

- In the proof the *basic substitution lemma* is crucial
- *The* lemma states that evaluating an expression *e* in the composition $M \circ \sigma$, gives the same result as evaluating in M the expression $e\sigma$ which results from first applying the substitution.

Array

- Expressions

$$a[e]$$

- Statements

$$a[e] \coloneqq e'$$

Special Notation

$$a[e] \coloneqq e'$$

$$\Downarrow$$

$$a \coloneqq (a[e] \coloneqq e')$$

The expression $a[e] \coloneqq e'$ denotes the array update defined by

$$(a[e] ::= e')(e'') = if\ e = e''then\ e'else\ a[e'']$$

Special predicate

$$
\begin{aligned}
\delta(x) &= true \\
\delta(a[e]) &= 0 \leq e \leq| a | \wedge \delta(e) \\
\delta(op(e_1, \ldots, e_n)) &= \delta(e_1) \wedge \cdots \wedge \delta(e_n)
\end{aligned}
$$

Special statements

We indicate the occurrence of an array-out-of-bound error by a statement **array-out-of-bound.**

This statement then
can be further evaluated in the context of error-handling constructs.

Assignment

$$\frac{}{\langle x := e, \sigma, \phi \rangle \ \rightarrow \langle \sigma[x = e], \phi \ \wedge \delta(e\sigma) \rangle}$$

$$\frac{}{\langle x := e, \sigma, \phi \rangle \ \rightarrow \langle ArrayOutOfBound, \phi \ \wedge \neg\delta(e\sigma) \rangle}$$

Array Assignment

$$\frac{}{\langle a[e] \coloneqq e', \sigma, \phi \rangle \ \rightarrow \langle \sigma[a \coloneqq (a[e\sigma] \coloneqq e'\sigma)], \phi \ \wedge \delta(a[e\sigma]) \wedge \delta(e', \sigma) \rangle}$$

$$\frac{}{\langle a[e] \coloneqq e', \sigma, \phi \rangle \ \rightarrow \langle ArrayOutOfBound, \phi \ \wedge \neg(\delta(a[e\sigma]) \wedge \delta(e', \sigma)) \rangle}$$

Other constructs

Recursion: requires the symbolic handling of closure

Classes and Objects: the symbolic execution is based on

- symbolic execution traces
- a weakest preconditon calculus.

Thread and concurrency ....

(my personal) Concluding Remarks

- Despite the popularity and success of symbolic execution techniques, the foundations of symbolic execution are still missing.
  - The foundations must cover in an uniform manner mainstream programming features
- Most existing tools for symbolic execution lack an explicit formal specification and justification.