Formal Methods and Specification (SS 2021)
Lecture 8:
Symbolic Program Execution

Stefan Ratschan

Katedra číslicového návrhu
Fakulta informačních technologií
České vysoké učení technické v Praze

Evropský sociální fond Praha & EU: Investujeme do vaší budoucnosti

Today's Program

Technique that

- already helped everybody, who has ever used Microsoft Windows and that
- helped Microsoft to save millions of dollars, and
- that Microsoft has been running on $>$ 100 computers since 2008.

[Godefroid et al., 2012]

Until now, primary goal: correctness proofs, found bugs only by-product

Now, primary goal: finding bugs

Prelude: Validity vs. Satisfiability

For example: real numbers

$$\models x + 1 \geq x$$

or, equivalently

$$\neg(x + 1 \geq x) \text{ that is } x + 1 < x \text{ unsatisfiable}$$

and, in general:

$$\models \phi$$
iff
formula $\neg\phi$ is unsatisfiable

Analogy:

Prove a formula ϕ
or, equivalently
assume $\neg\phi$ and find a contradiction

Intuition: satisfiability = solvability

Potential confusion:

we want to prove $\models \phi$, but algorithms usually decide satisfiability.

Today's Main Question

We want to test programs as completely as possible.

For example, as complete coverage of code with test cases as possible
(viz coverage criteria)

Execution path: sequence of program lines.

Main question:
How to cover with test cases
as many execution paths as possible?

Example:

```
1:  while x ≥ 0 do
2:      input x
3:      if 2x − 1 ≥ 3 then
4:          x ← x − 2
5:      else
6:          x ← x − 1
```

How to follow the execution path $1, 2, 3, 4, 1, 2, 3, 6$?

$$x \geq 0 \wedge 2x - 1 \geq 3 \wedge x = x - 2 \wedge x \geq 0 \wedge 2x - 1 < 3$$

Initial value and inputs for x may differ!

$$x_1 \geq 0 \wedge 2x_2 - 1 \geq 3 \wedge x_3 = x_2 - 2 \wedge x_3 \geq 0 \wedge 2x_4 - 1 < 3$$

Conjunction of all assignments and conditions along the path + variables indices

A solver can then find a test case or prove that none exists (demo).

Formalization

Program P:

1: **while** $x \geq 0$ **do**
2:     **input** x
3:     **if** $2x - 1 \geq 3$ **then**
4:         $x \leftarrow x - 2$
5:     **else**
6:         $x \leftarrow x - 1$

How to extract condition along execution path $1, 2, 3, 4, 1, 2, 3, 6$?

Notation: $BP_{1,2,3,4,1,2,3,6}(P)$

Variable indices? SSA

Corresponding basic path:

1: **assume** $x \geq 0$
2: **input** x
3: **assume** $2x - 1 \geq 3$
4: $x \leftarrow x - 2$
1: **assume** $x \geq 0$
2: **input** x
3: **assume** $\neg 2x - 1 \geq 3$
@ $\perp$

Formalization

Program P:	$BP_{1,2,3,4,1,2,3,6}(P)$	SSA
1: **while** $x \geq 0$ **do**	**assume** $x \geq 0$	**assume** $x_1 \geq 0$
2: **input** x	**input** x	**input** x_2
3: **if** $2x - 1 \geq 3$ **then**	**assume** $2x - 1 \geq 3$	**assume** $2x_2 - 1 \geq 3$
4: $x \leftarrow x - 2$	$x \leftarrow x - 2$	$x_3 \leftarrow x_2 - 2$
5: **else**	**assume** $x \geq 0$	**assume** $x_3 \geq 0$
6: $x \leftarrow x - 1$	**input** x	**input** x_4
	assume $\neg 2x - 1 \geq 3$	**assume** $\neg 2x_4 - 1 \geq 3$
	@ $\perp$	@ $\perp$

$$x_1 \geq 0 \land 2x_2 - 1 \geq 3 \land x_3 = x_2 - 2 \land x_3 \geq 0 \land 2x_4 - 1 < 3$$

$$X_{l_1,\dots,l_n}(P) :\Leftrightarrow F_{pre}(SSA(BP_{l_1,\dots,l_n}(P)))$$

where

$$F_{pre}(c_1; \dots; c_1; @\perp) := \bigwedge_{i \in \{1,\dots,n\}, F(c_i) \neq \top} F(c_i)$$

with F as defined for extracting verification conditions.

Summary of Formalization

The lines $l_1, \ldots, l_n$ of a program P can be executed in this order iff $X_{l_1,\ldots,l_n}(P)$ is satisfiable.

Here

$$X_{l_1,\ldots,l_n}(P) :\Leftrightarrow F_{pre}(SSA(BP_{l_1,\ldots,l_n}(P)))$$

where

$$F_{pre}(c_1; \ldots; c_1; \texttt{@}\bot) := \bigwedge_{i \in \{1,\ldots,n\}, F(c_i) \neq \top} F(c_i)$$

with

$$F(\textbf{assume } \phi) := \phi$$
$$F(\textbf{input } v) := \top$$
$$F(v \leftarrow t) := v = t$$

If satisfiable, then satisfying assignment results in test case

Assumption:

$l_1, \ldots, l_n$ is in a agreement with the control structures of the program

Execution Paths and Verification Conditions

```
1:  while x ≥ 0 do
2:      input x
3:      if 2x − 1 ≥ 3 then
4:          x ← x − 2
5:      else
6:          x ← x − 1
```

can execute lines $1, 2, 3, 4, 1, 2, 3, 6$

iff

assume $x \geq 0$
input x
assume $2x - 1 \geq 3$
$x \leftarrow x - 2$
assume $x \geq 0$
input x
assume $\neg 2x - 1 \geq 3$
@ $\perp$

is not correct.

iff its verification condition

$$[x_1 \geq 0 \wedge 2x_2 - 1 \geq 3 \wedge x_3 = x_2 - 2 \wedge x_3 \geq 0 \wedge 2x_4 - 1 < 3] \Rightarrow \perp$$

$$\neg[x_1 \geq 0 \wedge 2x_2 - 1 \geq 3 \wedge x_3 = x_2 - 2 \wedge x_3 \geq 0 \wedge 2x_4 - 1 < 3]$$

does not hold iff

$$x_1 \geq 0 \wedge 2x_2 - 1 \geq 3 \wedge x_3 = x_2 - 2 \wedge x_3 \geq 0 \wedge 2x_4 - 1 < 3$$

is satisfiable.

Symbolic Execution Based on Verification Conditions

The lines $l_1, \dots, l_n$ of a program P can be executed in this order

iff

$X_{l_1,\dots,l_n}(P)$ is satisfiable

iff

$F_{pre}(SSA(BP_{l_1,\dots,l_n}(P)))$ is satisfiable

iff

verification condition $VC(BP_{l_1,\dots,l_n}(P))$ does not hold

iff

$\neg VC(BP_{l_1,\dots,l_n}(P))$ is satisfiable

So, alternative definition:

$$X_{l_1,\dots,l_n}(P) \Leftrightarrow \neg VC(BP_{l_1,\dots,l_n}(P)) \Leftrightarrow \neg F(SSA(BP_{l_1,\dots,l_n}(P)))$$

Practical Usage of Symbolic Execution

Main question:

How to cover with test cases
as many execution paths as possible?

Method:

- Choose a set of execution paths
- For every execution path $l_1, \ldots, l_n$, find the corresponding test case by computing a satisfiable assignment of $F_{pre}(SSA(BP_{l_1,\ldots,l_n}(P)))$

Problems:

- undecidable theories (e.g., due to non-linear operations on integers)
- usage of external functions with unknown source code
  (e.g., from the operating system)

Example

$\ldots$
$x \leftarrow 2x + 1$
$r \leftarrow \text{ext}(x)$
if $r > 7$ **then**
    $y \leftarrow xy + z$
    if $y > 3$ **then**
        $\ldots$
    else
        $\ldots$
else
    $\ldots$

Condition for execution path leading into both **if** branches:

$$x_2 = 2x_1 + 1 \wedge r = \text{ext}(x_2) \wedge r > 7 \wedge y_2 = x_2 y_1 + z \wedge y_2 > 3$$

where ext() can, for example read a sensor in a nuclear power plant, call a function on super-computer, install some software on 2000 computers etc.

Hence no solver can handle this. What to do? approximation!

Handling Complex Functions: Over-Approximation

$$x_2 = 2x_1 + 1 \wedge r = \mathrm{ext}(x_2) \wedge r > 7 \wedge y_2 = x_2 y_1 + z \wedge y_2 > 3$$

$$x_2 = 2x_1 + 1 \wedge \top \wedge r > 7 \wedge y_2 = x_2 y_1 + z \wedge y_2 > 3$$

What does this mean? weaker constraint

The result "unsatisfiable" implies unsatisfiability of the original formula.

Satisfying assignment (i.e., a solution) is
not necessarily a satisfying assignment of the original formula.

In verification, where unsatisfiability means absence of an error,
this is what we want.

In symbolic execution: probably does not follow the wanted lines

Hence over-approximation is not what we want here

Handling Complex Functions: Under-Approximation

$$x_2 = 2x_1 + 1 \wedge r = \mathrm{ext}(x_2) \wedge r > 7 \wedge y_2 = x_2 y_1 + z \wedge y_2 > 3$$

$$x_2 = 2x_1 + 1 \wedge \perp \wedge r > 7 \wedge y_2 = x_2 y_1 + z \wedge y_2 > 3$$

unsatisfiable formula

better under-approximation?

For example: $x_2 = 2x_1 + 1 \wedge r = \sin(x_2) \wedge r > 7 \wedge y_2 = x_2 y_1 + z \wedge y_2 > 3$

We can execute the function for arbitrary inputs!

We compute the result for random input, e.g., $x_2 = 0$, so $r = 0$.

$$x_2 = 2x_1 + 1 \wedge x_2 = 0 \wedge r = 0 \wedge r > 7 \wedge y_2 = x_2 y_1 + z \wedge y_2 > 3$$

Problem?

Then $x_2 = 2x_1 + 1$ does not have a solution (if x_1 is an integer)

First we search for a solution of $x_2 = 2x_1 + 1$,
then we compute $\mathrm{ext}(x_2)$ and so on

Example

$$
\begin{array}{l}
x \leftarrow 2x+1 \\
r \leftarrow \mathrm{ext}(x) \\
\textbf{if } r > 7 \textbf{ then} \\
\quad y \leftarrow xy + z \\
\quad \textbf{if } y > 3 \textbf{ then} \\
\quad\quad \ldots \\
\quad \textbf{else} \\
\quad\quad \ldots \\
\textbf{else} \\
\quad \ldots
\end{array}
$$

$$
\begin{array}{l}
x_2 = 2x_1 + 1 \wedge \\
x_2 = 5 \wedge r = 13 \wedge \\
r > 7 \wedge \\
y_2 = x_2 y_1 + z \wedge \\
y_2 > 3
\end{array}
$$

we arrive at the external function ext(), for its execution we need input (x_2)

we solve $x_2 = 2x_1 + 1$ e.g., $x_1 \mapsto 2$, $x_2 \mapsto 5$

we execute ext(5), the result can, for example, be 13

instead of $r = \mathrm{ext}(x_2)$ we add $x_2 = 5, r = 13$ to the formula.

the resulting formula is in a solvable class, so we can continue.

Dynamic Test Generation/Concolic Testing

Execute the both program concretely and symbolically.

Create the logical formula $X_{l_1,\dots,l_n}(P)$ line by line.

As soon as formula creation
arrives at a function that the solver cannot handle,
let the program run until we receive the result of the function.

If, for execution, the function needs inputs,
we compute them by solving the symbolic formula
(of part, that corresponds to the program before the external function)

Use the result of the concrete execution to
replace the external function call by corresponding equalities

Completeness

Does the method always find a test case following a given path, if it exists?

(of course) not, we only solve the formula approximately:

$x \leftarrow 2x + 1$
$r \leftarrow \mathrm{ext}(x)$
if $r > 7$ **then**
  $y \leftarrow xy + z$
  if $y > 3$ **then**
    $\ldots$
  else
    $\ldots$
else
  $\ldots$

$$\begin{aligned} & x_2 = 2x_1 + 1 \wedge \\ & x_2 = 5 \wedge r = 6 \wedge \\ & r > 7 \wedge \\ & y_2 = x_2 y_1 + z \wedge \\ & y_2 > 3 \end{aligned}$$

Assume that the result of executing ext(5) is 6, and of ext(7) it is 13.

Due to our choice of a solution of $x_2 = 2x_1 + 1$ we add $x_2 = 5, r = 6$, and the resulting formula does not have a solution.

Symbolic Execution for Test Case Generation

Cover tree of paths

(either if or else, either we stay in loop, or we leave it)

... up to certain depth

formula is built and checked incrementally

$x \leftarrow 2x + 1$
$r \leftarrow \text{ext}(x)$
if $r > 7$ **then**
  $y \leftarrow xy + z$
  if $y > 3$ **then**
    ...
  else
    ...
else
  ...

$x_2 = 2x_1 + 1 \wedge$
$x_2 = 5 \wedge r = 13 \wedge$
$r > 7 \wedge$
$y_2 = x_2 y_1 + z \wedge$
$\neg y_2 > 3$

Path Explosion Problem

@ $\forall i \in \{1, \ldots, 1000\} \,.\, a[i] \in \{0, 1\}$
for $i \leftarrow 1$ **to** 1000 **do**
    if $a[i] = 0$ **then**
        $\ldots$
    else
        $\ldots$

How many execution paths? 2^{1000}

The coverage of the full tree of execution paths
up to a practically relevant depth is often unrealistic.

Possible solutions:

- choice of paths
- merging paths

SAGE: Whitebox Fuzzing [Godefroid et al., 2012]

Instead of depth-first search, local search:

Using test cases from a different source,
cover neighboring paths by negating the individual conditions.

Technological issues (e.g., works on machine code)

Regularily finds security bugs (e.g., buffer overflow),
which saves security updates, that Microsoft would have to
send to millions of computers, worldwide.

https://en.wikipedia.org/wiki/OneFuzz

Symbolic Execution: Tools and Literature

Many tools available.

Some for specific programming languages,
some for LLVM intermediate representation.

see Wikipedia "Symbolic execution", "Concolic testing"

Survey articles: [Cadar and Sen, 2013], [Baldoni et al., 2018]

MI-TES: symbolic execution of timed automata
(program line $\sim$ location of timed automaton)

Literature I

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300.

Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408795.

Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, March 2012. doi: 10.1145/2093548.2093564.