Topics

Last lecture
  • Getting started with solver-aided programming.

Today
  • Going pro with solver-aided programming.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

Solver-aided programming in two parts: (1) getting started and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
How to build your own solver-aided tool or language

The classic (hard) way to build a tool
What is hard about building a solver-aided tool?

An easier way: tools as languages
How to build tools by stacking layers of languages.

Behind the scenes: symbolic virtual machine
How Rosette works so you don’t have to.

A last look: a few recent applications
Cool tools built with Rosette!
The classic (hard) way to build a tool

Recall the solver-aided programming tool chain: the tool reduces a query about program behavior to an SMT problem.

```plaintext
P(x) {
    ...
    ...
} assert safe(x, P(x))
```

Recall the solver-aided tool:

- `∃x. ¬safe(x, P(x))`
- `x = 42 ∧ safe(x, P(x))`
- `∃e. ∀x. safe(x, Pe(x))`
The classic (hard) way to build a tool

Recall the solver-aided programming tool chain: the tool reduces a query about program behavior to an SMT problem. What all queries have in common: they need to translate programs to constraints!
The classic (hard) way to build a tool
Wanted: an easier way to build tools

- verify
- solve
- synthesize

```plaintext
P(x) {
  ...
  ...
}
assert safe(x, P(x))
```

- programming
- an interpreter for the source language
Wanted: an easier way to build tools

Technical challenge: how to efficiently translate a program and its interpreter? [Torlak & Bodik, PLDI’14]

\[\text{assert \ safe}(x, P(x))\]
How to build your own solver-aided tool or language

The classic (hard) way to build a tool
What is hard about building a solver-aided tool?

An easier way: tools as languages
How to build tools by stacking layers of languages.

Behind the scenes: symbolic virtual machine
How Rosette works so you don’t have to.

A last look: a few recent applications
Cool tools built with Rosette!
Layers of classic languages: guests and hosts

guest language

library (shallow) embedding

interpreter (deep) embedding

host language

D3 → guest language → Python

JavaScript → guest language → C
Layers of solver-aided languages

solver-aided guest language

library (shallow) embedding

interpreter (deep) embedding

solver-aided host language
Layers of solver-aided languages

C (subset) → solver-aided guest language → interpreter (deep) embedding → BPF, x86 32, x86 64, ARM 32, ARM 64, RISC-V 32, RISC-V 64

library (shallow) embedding

Jitterbug (OSDI 2020): Verifying and synthesizing BPF JITs in Linux.

RSETSÆTE
A tiny example solver-aided guest language

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6
```

**BV**: A tiny assembly-like language for writing fast, low-level library functions.

1. interpreter [50 LOC]
2. verifier [free]
3. synthesizer [free]

We want to **test**, **verify**, and **synthesize** programs in the BV SDSL.
A tiny example language

def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> bvmax(-2, -1)
A tiny example language

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> bvmax(-2, -1)
```

 defin (interpret prog inputs)
 (make-registers prog inputs)
 (for ([stmt prog])
     (match stmt
         ([list out opcode in ...]
         (define op (lookup opcode))
         (define args (map load in))
         (store out (apply op args)))))
 (load (last)))
A tiny example language

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> bvmax(-2, -1)
-1
```

(\textbf{define} \texttt{bvmax}
\begin{itemize}
  \item (2 bvsge 0 1)
  \item (3 bvneg 2)
  \item (4 bvxor 0 2)
  \item (5 bvand 3 4)
  \item (6 bvxor 1 5))
\end{itemize}

\textbullet\ pattern matching
\textbullet\ first-class & higher-order procedures
\textbullet\ side effects

(\textbf{define} (interpret prog inputs)
  (make-registers prog inputs)
  (for [[stmt prog]]
    (match stmt
      [(list out opcode in ...)
        (define op (lookup opcode))
        (define args (map load in))
        (store out (apply op args))]]))
  (load (last)))
A tiny example language

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> verify(bvmax, max)
```

```plaintext
(define-symbolic x y int32?)
(define in (list x y))
(verify
(assert (equal? (interpret bvmax in) (apply max in))))
```
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> verify(bvmax, max)

(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmax in) (apply max in))))

(define (max x y)
  (if (bvsge x y) x y))
A tiny example language

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> verify(bvmax, max)
```

Creates two fresh symbolic values of type 32-bit integer and binds them to the variables x and y.

```racket
(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmax in) (apply max in))))
```
A tiny example language

```
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> verify(bvmax, max)
```

Creates two fresh symbolic values of type 32-bit integer and binds them to the variables \( x \) and \( y \).

```
(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmax in) (apply max in))))
```

Symbolic values can be used just like concrete values of the same type.
A tiny example language

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> verify(bvmax, max)
```

**query**

```
(define-symbolic x y int32?)
(define in (list x y))
(verify
    (assert (equal? (interpret bvmax in) (apply max in))))
```

Creates two fresh symbolic values of type 32-bit integer and binds them to the variables \( x \) and \( y \).

Symbolic values can be used just like concrete values of the same type.
A tiny example language

\[
\begin{align*}
def\ bmvax(r0, r1) : \\
  r2 &= \ bvsge(r0, r1) \\
  r3 &= \ bvneg(r2) \\
  r4 &= \ bvxor(r0, r2) \\
  r5 &= \ bvand(r3, r4) \\
  r6 &= \ bvxor(r1, r5) \\
  \text{return } r6
\end{align*}
\]

> \textbf{verify}(bmvax, max) \\
\[0, -2\]
A tiny example language

```python
def bvmax(r0, r1):
    r2...r6 = inst??(bvsge, bvneg, bvxor, bvand)
    return r6

> synthesize(bvmax, max)
```

```scheme
(define-symbolic x y int32?)
(define in (list x y))
(synthesize
 #:forall in
 #:guarantee
 (assert (equal? (interpret bvmax in) (apply max in))))
```
A tiny example language

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 =bvneg(r2)
    r4 = bvxor(r0, r1)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> synthesize(bvmax, max)
```

```scheme
(define-symbolic x y int32?)
(define in (list x y))
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in) (apply max in)))))
```
How to build your own solver-aided tool or language

The classic (hard) way to build a tool
What is hard about building a solver-aided tool?

An easier way: tools as languages
How to build tools by stacking layers of languages.

Behind the scenes: symbolic virtual machine
How Rosette works so you don’t have to.

A last look: a few recent applications
Cool tools built with Rosette!
How it all works: a big picture view

- query
- result
- program
- guest language
- theories of bitvectors, integers, reals, and uninterpreted functions
- pattern matching
- dynamic evaluation
- first-class procedures
- higher-order procedures
- side effects
- macros

RiSETTE

Symbolic Virtual Machine

SMT solver Z3
Translation to constraints by example

\[
(a, b) \quad \text{reverse and filter, keeping only positive numbers} \quad \text{constraints} \quad a > 0 \land b > 0
\]
Design space of precise symbolic encodings

solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)

symbolic execution

bounded model checking

solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)
Challenge: simple vs compact encoding (SE and BMC)

symbolic execution

concrete evaluation

Can we have both a polynomially sized encoding (like BMC) and concrete evaluation of complex operations (like SE)?
Solution: type-driven state merging

solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
Solution: type-driven state merging

solve:
    \[ ps = () \]
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)

Merge instances of:
- primitive types: symbolically
- value types: structurally
- all other types: via unions

\[ \{ a > 0 \} \]
\[ \{ b > 0 \} \]
\[ \text{true} \]
Solution: type-driven state merging

solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)

Merge instances of
    › primitive types: symbolically
    › value types: structurally
    › all other types: via unions
Solution: type-driven state merging

```
solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
```

Merge instances of

- primitive types: symbolically
- value types: structurally
- all other types: via unions
Solution: type-driven state merging

solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)

Merge instances of
- primitive types: symbolically
- value types: structurally
- all other types: via unions

\{ a > 0 \}
\{ b > 0 \}
\{ true \}
Solution: type-driven state merging

solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)

Evaluate \( \text{len} \) concretely on all lists in the union; assertion true only on the list guarded by \( g_2 \).

Execute \( \text{insert} \) concretely on all lists in the union.

\[ g_0 = a > 0 \]
SymPro (OOPSLA’18): use **symbolic profiling** to find performance bottlenecks in solver-aided code.

Solution: **type-driven state merging**

solve:

```python
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)
```

**polynomial encoding**

- $g_0 = a > 0$
- $g_1 = b > 0$
- $g_2 = g_0 \land g_1$
- $g_3 = \neg(g_0 \leftrightarrow g_1)$
- $g_4 = \neg g_0 \land \neg g_1$

$c = \text{ite}(g_1, b, a)$

**concrete evaluation**

- $c = \text{ite}(g_1, b, a)$
- $g_2 \vdash (b, a)$
- $g_3 \vdash (c)$
- $g_4 \vdash ()$

**symbolic virtual machine**

<table>
<thead>
<tr>
<th>vs</th>
<th>ps</th>
<th>$g_0$</th>
<th>$g_1$</th>
<th>$g_2$</th>
<th>$g_3$</th>
<th>$g_4$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, b)</td>
<td>()</td>
<td>$\neg g_0$</td>
<td>$g_0$</td>
<td>$g_0 \vdash (a)$</td>
<td>$\neg g_0 \vdash ()$</td>
<td>$\neg g_0 \vdash (b)$</td>
<td>$\neg g_0 \vdash ()$</td>
</tr>
<tr>
<td>(a)</td>
<td>()</td>
<td>$g_1$</td>
<td>$\neg g_1$</td>
<td>$\neg g_1 \vdash ()$</td>
<td>$\neg g_1 \vdash (a)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

SymPro (OOPSLA’18): use **symbolic profiling** to find performance bottlenecks in solver-aided code.
How to build your own solver-aided tool or language

The classic (hard) way to build a tool
What is hard about building a solver-aided tool?

An easier way: tools as languages
How to build tools by stacking layers of languages.

Behind the scenes: symbolic virtual machine
How Rosette works so you don’t have to.

A last look: a few recent applications
Cool tools built with Rosette!
30+ tools
programming languages, software engineering, systems, architecture, networks, security, formal methods, databases, education, games, …

programming languages, formal methods, and software engineering
type systems and programming models
compilation and parallelization
safety-critical systems [CAV’16]
test input generation
software diversification

education and games
hints and feedback
problem generation
problem-solving strategies

systems, architecture, networks, security, and databases
memory models
OS components
data movement for GPUs
router configuration
cryptographic protocols
Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW

- 30 years of incident-free service.
- Controlled by custom software, built by CNTS engineering staff.
- Third generation of Therapy Control software built recently.
Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW

- Prescription
- Sensors
- Therapy Control Software
- Beam, motors, etc.
Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW

Experimental Physics and Industrial Control System (EPICS) Dataflow Language

Therapy Control Software
Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW

EPICS program

safety property

EPICS verifier

bug report
Verifying a radiation therapy system

Found safety-critical defects in a pre-release version of the therapy control software.

Used by CNTS staff to verify changes to the controller.
Summary

Today
  • Going pro with solver-aided programming.

Next lecture
  • Getting started with SAT solving!