Solver-aided programming: going pro

Emina Torlak
University of Washington
emina@cs.washington.edu
homes.cs.washington.edu/~emina/
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!
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.

```
verify debug solve synthesize

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

```
P(x) {
  ...
  ...
} assert safe(x, P(x))
```
Wanted: an easier way to build tools

Programming

assert safe(x, P(x))

P(x) {
  ...
  ...
}

verify debug solve synthesize

solve

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

```
P(x) {
  ...
  ...
}
assert safe(x, P(x))
```
Wanted: an easier way to build tools

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

**domain-specific language (DSL)**

A formal language that is specialized to a particular application domain and often limited in capability.

**host language**

A high-level language for implementing DSLs, usually with meta-programming features.
Layers of classic languages: DSLs and hosts

- **domain-specific language (DSL)**
  - library *(shallow)* embedding
  - interpreter *(deep)* embedding

- **host language**

- A formal language that is specialized to a particular application domain and often limited in capability.

- A high-level language for implementing DSLs, usually with meta-programming features.
Layers of classic languages: many DSLs and hosts

- **domain-specific language (DSL)**
  - library (shallow) embedding
  - interpreter (deep) embedding
- **host language**
- **artificial intelligence**
  - Church, BLOG
- **databases**
  - SQL, Datalog
- **hardware design**
  - Bluespec, Chisel, Verilog, VHDL
- **math and statistics**
  - Eigen, Matlab, R
- **layout and visualization**
  - LaTex, dot, dygraphs, D3
- **Racket, Scala, JavaScript, …**
Layers of classic languages: why DSLs?

domain-specific language (DSL)

library (shallow) embedding

interpreter (deep) embedding

host language

$C = A \times B$

for ($i = 0; i < n; i++$)
  for ($j = 0; j < m; j++$)
    for ($k = 0; k < p; k++$)
      $C[i][k] += A[i][j] \times B[j][k]$
Layers of classic languages: why DSLs?

- **domain-specific language (DSL)**
  - library *(shallow)* embedding
  - interpreter *(deep)* embedding
  - host language

Easier for people to read, write, and get right.

```
C = A * B
```

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    for (k = 0; k < p; k++)
      C[i][k] += A[i][j] * B[j][k]
```
Layers of classic languages: why DSLs?

- **domain-specific language (DSL)**
  - library (shallow) embedding
  - interpreter (deep) embedding
  - host language

**Easier for people to read, write, and get right.**

- Eigen / Matlab
  - \[ C = A \times B \] [associativity]

**Easier for tools to analyze.**

- C / Java
  - ```cpp
  for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
  for (k = 0; k < p; k++)
  C[i][k] += A[i][j] \times B[j][k]
  ```
Layers of solver-aided languages

- **Solver-aided domain-specific language (SDSL)**
- **Library (shallow) embedding**
- **Interpreter (deep) embedding**
- **Solver-aided host language**
Layers of solver-aided languages: tools as SDSLs

education and games
Enlearn, RuleSy (VMCAI’18), Nonograms (FDG’17), UCB feedback generator (ITiCSE'17)

synthesis-aided compilation
LinkiT tools, Chlorophyll (PLDI’14), GreenThumb (ASPLOS’16)

type system soundness
Bonsai (POPL’18)

systems software
Serval (under submission)

computer architecture
MemSynth (PLDI’17)

radiation therapy control
Neutrons (CAV’16)

... and more
Layers of solver-aided languages: tools as SDSLs

- **education and games**
  - Enlearn, RuleSy (VMCAI’18),
  - Nonograms (FDG’17), UCB feedback generator (ITiCSE’17)

- **synthesis-aided compilation**
  - LinkiTools, Chlorophyll (PLDI’14),
  - GreenThumb (ASPLOS’16)

- **type system soundness**
  - Bonsai (POPL’18)

- **systems software**
  - Serval (under submission)

- **computer architecture**
  - MemSynth (PLDI’17)

- **radiation therapy control**
  - Neutrons (CAV’16)

... and more
A tiny example SDSL

\[
\text{def bvmax(r0, r1) :}
\]
\[
r2 = \text{bvsge}(r0, r1)
\]
\[
r3 = \text{bvneg}(r2)
\]
\[
r4 = \text{bvxor}(r0, r2)
\]
\[
r5 = \text{bvand}(r3, r4)
\]
\[
r6 = \text{bvxor}(r1, r5)
\]
\[
\text{return } r6
\]

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

```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.

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

```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 [10 LOC]
2. verifier [free]
3. debugger [free]
4. synthesizer [free]

We want to **test**, **verify**, **debug**, and **synthesize** programs in the BV SDSL.
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 SDSL

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

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

```scheme
(define bvmax
  `((2 bvsge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))
```
A tiny example SDSL

```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)
```

```lisp
(define bvmax
  `(2 bvsge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))

(out opcode in ...)
```
A tiny example SDSL

```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)
```

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
 `(-2 -1)
```

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

```
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)
```

```
(define bvmax
 `((2 bvsge 0 1)
  (3 bvneg 2)
  (4 bvxor 0 2)
  (5 bvand 3 4)
  (6 bvxor 1 5)))

(define (interpret prog inputs)
 (make-registers prog inputs)
 (for ([stmt prog])
    (match stmt
      [(list out opcode in ...)
       (define op (eval opcode))
       (define args (map load in))
       (store out (apply op args)))]))

(load (last))
```
A tiny example SDSL

```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)
```

```scheme
(define bvmax
  `(0 -2
    `(2 bvsge 0 1)
    `(3 bvneg 2)
    `(4 bvxor 0 2)
    `(5 bvand 3 4)
    `(6 bvxor 1 5)))

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

```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)
```

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

**Interpret**

```Scheme
(define (interpret prog inputs)
  (make-registers prog inputs)
  (for ([stmt prog])
    (match stmt
      [(list out opcode in ...)]
      (define op (eval opcode))
      (define args (map load in))
      (store out (apply op args)]))
  (load (last)))
```

```
(define bvmax
  `((2 bvsge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))

0 -2
1 -1
2
3
4
5
6
```
A tiny example SDSL

```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)
```

```
(define bvmax
  '((2 bvsge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))
```

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

```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)
```

(\begin{verbatim}
(define bvmax
  `((2 bvsge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))

(define (interpret prog inputs)
  (make-registers prog inputs)
  (for ([stmt prog])
    (match stmt
      [(list out opcode in ...)]
      ([define op (eval opcode)])
      ([define args (map load in)])
      ([store out (apply op args)]))
  (load (last)))
```

-interpreter
A tiny example SDSL

```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)
```

```plaintext
(define bvmax
  `((2 bvsge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))
```

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

```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
```

(define bvmax
  `((2 bvsge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))

 interprets:

```scheme
(define (interpret prog inputs)
  (make-registers prog inputs)
  (for ([stmt prog])
    (match stmt)
      [(list out opcode in ...)]
        (define op (eval opcode))
        (define args (map load in))
        (store out (apply op args)))
  (load (last)))
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>-2</td>
<td>-1</td>
<td></td>
<td></td>
<td>-2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>-1</td>
</tr>
</tbody>
</table>
A tiny example SDSL

```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
```

(define bvmax
  `((2 bvsge 0 1)
     (3 bvneg 2)
     (4 bvxor 0 2)
     (5 bvand 3 4)
     (6 bvxor 1 5)))

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

```
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)
                  (interpret max in))))
```
A tiny example SDSL

```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)
                  (interpret max in))))
```

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

```
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) (interpret max in)))
```

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

```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)
```

(verify `expr`) searches for a concrete interpretation of symbolic values that causes `expr` to fail.

Symbolic values can be used just like concrete values of the same type.

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) (interpret max in)))))

Query
A tiny example SDSL

```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)
[0, -2]
```

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

```
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)
[0, -2]

> bvmax(0, -2)
-1
```

(\texttt{define-symbolic \textit{x y int32}?})
(\texttt{define \textit{in (list x y)})
(\texttt{verify})
  (\texttt{assert (equal? (interpret bvmax in) (interpret max in))))
A tiny example SDSL

```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

> debug(bvmax, max,[0, -2])
```

Query:

```scheme
(define in (list (int32 0) (int32 -2)))
(debug [register?]
  (assert (equal? (interpret bvmax in)
      (interpret 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

> debug(bvmax, max,[0, -2])
A tiny example SDSL

```python
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(??, ??)
    r5 = bvand(r3, ??)
    r6 = bvxor(??, ??)
    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) (interpret max in)))))
```
A tiny example SDSL

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)
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
program
SDSL
ROSETTE
Symbolic Virtual Machine
SMT solver Z3
How it all works: a big picture view

query → result

program

SDSL

ROSETTE

Symbolic Virtual Machine

SMT solver Z3
How it all works: a big picture view

- R*SETTE
  - program
    - SDSL
      - query
      - result
  - Symbolic Virtual Machine
  - SMT solver Z3
  - theories of bitvectors, integers, reals, and uninterpreted functions
  - pattern matching
  - dynamic evaluation
  - first-class procedures
  - higher-order procedures
  - side effects
  - macros
Translation to constraints by example

\[
\begin{align*}
\text{solve:} & \\
\text{ps} & = () \\
\text{for} & \\
v & \text{in} \\
\text{vs} & \\
\text{if} & \\
v & > 0 \\
\text{ps} & = \text{insert}(v, \text{ps}) \\
\text{assert} & \\
\text{len}(\text{ps}) & = \text{len}(\text{vs}) \\
\end{align*}
\]

reverse and filter, keeping only positive numbers

\[
\text{vs} \quad (3, 1, -2) \quad \rightarrow \quad \text{reverse and filter, keeping only positive numbers} \quad \rightarrow \quad \text{ps} \quad (1, 3)
\]
Translation to constraints by example

\[
\text{vs} = (3, 1, -2)
\]

\[
\begin{align*}
\text{ps} & = () \\
\text{for } v \text{ in vs:} & \\
\quad & \text{if } v > 0: \\
\quad & \quad \text{ps} = \text{insert}(v, \text{ps})
\end{align*}
\]

\[
\text{ps} = (1, 3)
\]
Translation to constraints by example

solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
Translation to constraints by example

solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
Translation to constraints by example

solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
Design space of precise symbolic encodings

solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)
Design space of precise symbolic encodings

solve:

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

symbolic execution:

- $\text{vs} \mapsto (a, b)$
- $\text{ps} \mapsto ()$

- $a > 0$
  - $\text{ps} \mapsto (a)$
- $b \leq 0$
  - $\text{ps} \mapsto (a)$

- $\{a > 0, b \leq 0, \text{false}\}$
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

vs $\mapsto (a, b)$
ps $\mapsto ()$

a \leq 0
ps $\mapsto ()$
a > 0
ps $\mapsto (a)$

b \leq 0
ps $\mapsto ()$
b > 0
ps $\mapsto (b)$

\{ a \leq 0 \} \lor \{ a > 0 \}
\{ b \leq 0 \} \lor \{ b > 0 \}
\{ \text{false} \} \lor \{ \text{true} \}

bounded model checking
Design space of precise symbolic encodings

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

\[
\begin{align*}
ps_0 &= \text{ite}(a > 0, (a), ( )) \\
ps_1 &= \text{insert}(b, ps_0)
\end{align*}
\]
Design space of precise symbolic encodings

solve:

\[
\begin{align*}
ps &= () \\
\text{for } v \text{ in } vs: \quad &
\begin{align*}
&\text{if } v > 0: \\
&\quad ps = \text{insert}(v, ps) \\
&\text{assert } \text{len}(ps) == \text{len}(vs)
\end{align*}
\end{align*}
\]

symbolic execution

bounded model checking

\[
\begin{align*}
ps_0 &= \text{ite}(a > 0, (a), ( )) \\
ps_1 &= \text{insert}(b, ps_0) \\
ps_2 &= \text{ite}(b > 0, ps_0, ps_1) \\
&\text{assert } \text{len}(ps_2) = 2
\end{align*}
\]
A new design: type-driven state merging

solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
A new design: 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}
A new design: 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 \}
A new design: 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 new design: 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
solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)

A new design: type-driven state merging
solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
solve:
\[
\begin{align*}
    ps &= () \\
    \text{for } v \text{ in } vs: \\
    &\quad \text{if } v > 0: \\
    &\quad \quad ps = \text{insert}(v, ps) \\
    &\quad \text{assert } \text{len}(ps) == \text{len}(vs)
\end{align*}
\]

Symbolic union: a set of guarded values, with disjoint guards.

\[g_0 = a > 0\]

\[g_1 = b > 0\]

\[g_2 = g_0 \land g_1\]

\[g_3 = \neg(g_0 \lor g_1)\]

\[g_4 = \neg g_0 \land \neg g_1\]

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

assert \[\text{len}(ps) == \text{len}(vs)\]

A new design: type-driven state merging
A new design: type-driven state merging

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

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

symbolic virtual machine

\[
\text{vs} \mapsto (a, b)
\]
\[
\text{ps} \mapsto ()
\]
\[
\neg g_0 \mapsto ()
\]
\[
g_0 \mapsto (a)
\]
\[
\text{ps} \mapsto ()
\]
\[
\text{ps} \mapsto (a)
\]
\[
\text{ps} \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash () \}
\]
\[
\text{ps} \mapsto \{ g_0 \vdash (b, a), \neg g_0 \vdash (b) \}
\]
\[
g_0 = a > 0
\]
\[
g_1 = b > 0
\]
A new design: type-driven state merging

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

g₀ = a > 0
g₁ = b > 0
A new design: type-driven state merging

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

g₀ = a > 0
g₁ = b > 0
g₂ = g₀ ∧ g₁

\[ g₃ = \neg(g₀ \iff g₁) \]
\[ g₄ = \neg g₀ \land \neg g₁ \]
c = ite(g₁, b, a)
A new design: type-driven state merging

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

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

```
g_0 = a > 0
g_1 = b > 0
g_2 = g_0 \land g_1
g_3 = \neg (g_0 \iff g_1)
g_4 = \neg g_0 \land \neg g_1
c = \text{ite}(g_1, b, a)
assert g_2
```
A new design: type-driven state merging

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

g₀ = a > 0
g₁ = b > 0
g₂ = g₀ ∧ g₁
g₃ = ¬(g₀ ↔ g₁)
g₄ = ¬g₀ ∧ ¬g₁
c = ite(g₁, b, a)
assert g₂
A new design: type-driven state merging

(Sym)Pro tip: use **symbolic profiling** to find and repair performance bottlenecks in your Rosette programs.

---

**concrete evaluation**

solve:

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

---

**symbolic virtual machine**

<table>
<thead>
<tr>
<th>ps</th>
<th>vs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(a, b)</td>
</tr>
<tr>
<td>(</td>
<td>a</td>
</tr>
</tbody>
</table>


---

**polynomial encoding**

- \( g_0 = a > 0 \)
- \( g_1 = b > 0 \)
- \( g_2 = g_0 \land g_1 \)
- \( g_3 = \neg(g_0 \leftrightarrow g_1) \)
- \( g_4 = \neg g_0 \land \neg g_1 \)
- \( c = \text{ite}(g_1, b, a) \)
- \( \text{assert } g_2 \)
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!
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
Verifying a radiation therapy system

EPICS program

safety property

EPICS verifier

bug report

Prototyped in a few days and found bugs.

Calvin Loncaric
Verifying a radiation therapy system

EPICS program

safety property

EPICS verifier

bug report

[Perrestein et al., CAV’16, ICALEPCS’17]

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

Used by CNTS staff to verify changes to the controller.
Synthesizing strategies for games and education

**Nonograms** game mechanics:
The numbered hints describe how many contiguous blocks of cells are filled with *true*. Cells filled with *true* are marked as a black square and cells filled with *false* as a red X.
Nonograms game mechanics:
The numbered hints describe how many contiguous blocks of cells are filled with true. Cells filled with true are marked as a black square and cells filled with false as a red X.

A computer solves puzzles by reducing the game mechanics to backtracking search, but human players solve puzzles by using multiple strategies to make progress without guessing.

Finding these strategies is a key challenge in game design, and is usually done through human testing.
Synthesizing strategies for games and education

A computer solves puzzles by reducing the game mechanics to backtracking search, but human players solve puzzles by using multiple strategies to make progress without guessing.

Finding these strategies is a key challenge in game design, and is usually done through human testing.

The “big hint” strategy.
Synthesizing strategies for games and education

A computer solves puzzles by reducing the game mechanics to backtracking search, but human players solve puzzles by using multiple strategies to make progress without guessing.

Finding these strategies is a key challenge in game design, and is usually done through human testing.
Synthesizing strategies for games and education

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Game mechanics

Game states for training and testing

Strategy DSL synthesizer

An optimal set of most concise, general, and sound strategies
Synthesizing strategies for games and education

Game mechanics

Game states for training and testing

Strategy DSL synthesizer

An optimal set of most concise, general, and sound strategies

Eric Butler

Prototyped in a few weeks and synthesized real strategies.
Synthesizing strategies for games and education

A strategy DSL synthesizer

Game mechanics

Game states for training and testing

An optimal set of most concise, general, and sound strategies

[Butler et al., FDG’17, VMCAI’18]

Synthesized strategies that outperform documented strategies for Nonograms, both in terms of coverage and quality.

Also used to synthesize strategies for solving K-12 algebra and proofs for propositional logic, recovering and outperforming textbook strategies for these domains.
Verifying systems software

An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
<td>security monitors</td>
</tr>
<tr>
<td>OS kernel</td>
<td></td>
</tr>
<tr>
<td>hardware</td>
<td></td>
</tr>
</tbody>
</table>
Verifying systems software

An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
</tr>
</tbody>
</table>

| OS kernel |

| hardware |

Bugs in OS components are bad news for reliability, security, and performance of computer systems.
Verifying systems software

An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
<td>security monitors</td>
</tr>
<tr>
<td>file systems</td>
<td>device drivers</td>
</tr>
<tr>
<td>OS kernel</td>
<td></td>
</tr>
<tr>
<td>hardware</td>
<td></td>
</tr>
</tbody>
</table>

Bugs in OS components are bad news for reliability, security, and performance of computer systems.

Verifying OS components is hard: e.g., the seL4 kernel took 11 person-years to prove, with a proof-to-implementation ratio of 20:1.
An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
</tr>
<tr>
<td>security monitors</td>
</tr>
<tr>
<td>file systems</td>
</tr>
<tr>
<td>device drivers</td>
</tr>
<tr>
<td>OS kernel</td>
</tr>
<tr>
<td>hardware</td>
</tr>
</tbody>
</table>

Bugs in OS components are bad news for reliability, security, and performance of computer systems.

### System specification

**Serval verifiers** (RISCV, x86, ...)

**System (binary) implementation**

**Bug report or correctness guarantee**
Verifying systems software

System specification

System (binary) implementation

Serval verifiers (RISCV, x86, ...)

Bug report or correctness guarantee

Each verifier took a couple of weeks to build!

Luke Nelson
Verified three existing security monitors (CertiKOS, Komodo, Keystone) fully automatically and with low proof burden.

Found and patched 15 bugs in the Linux BPF JITs for RISCV64 and x86-32.
thanks
thanks

ROSETTE

symbolic virtual machine

your SDSL

verify
debbug
slove
synthesize