Computer-Aided Reasoning for Software

CSE507

Solver-Aided Programming II

Topics

Last lecture

- Getting started with solver-aided programming.

Today

- Going pro with solver-aided programming.

topics

A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

ROSETTE

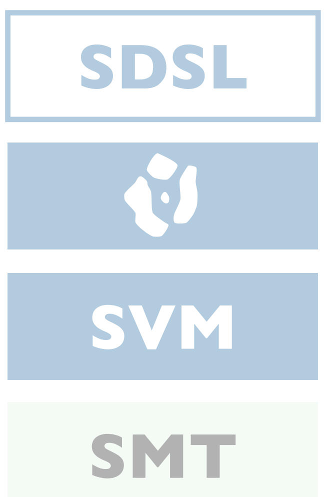
Solver-aided programming in two parts: (1) getting started and (2) going pro

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.

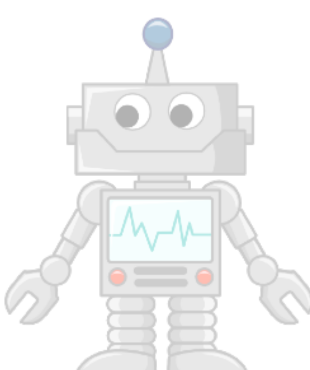
How to build your own solver-aided tool or language

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.

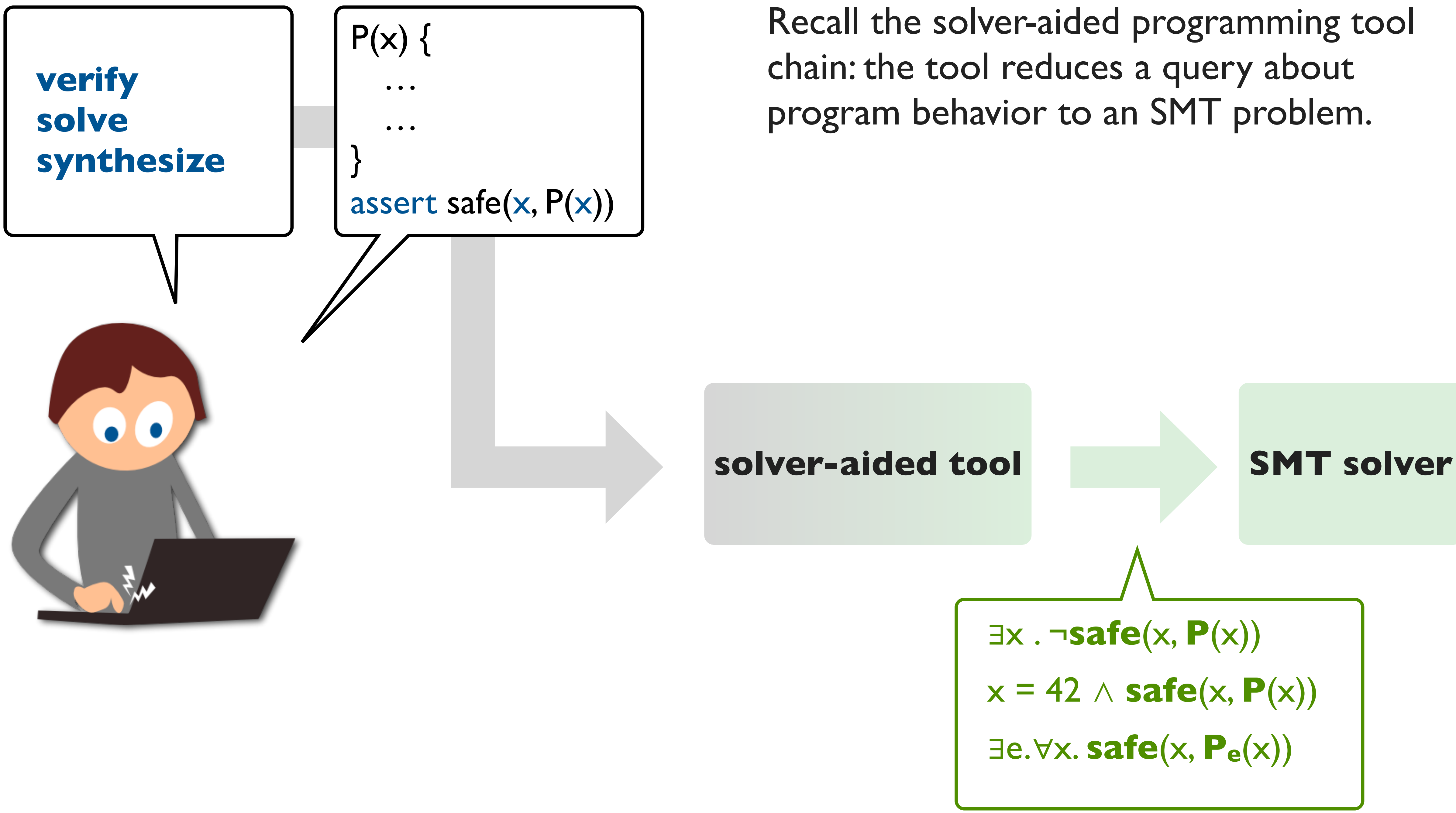
Behind the scenes: symbolic virtual machine
How Rosette works so you don't have to.



A last look: a few recent applications
Cool tools built with Rosette!

The classic (hard) way to build a tool

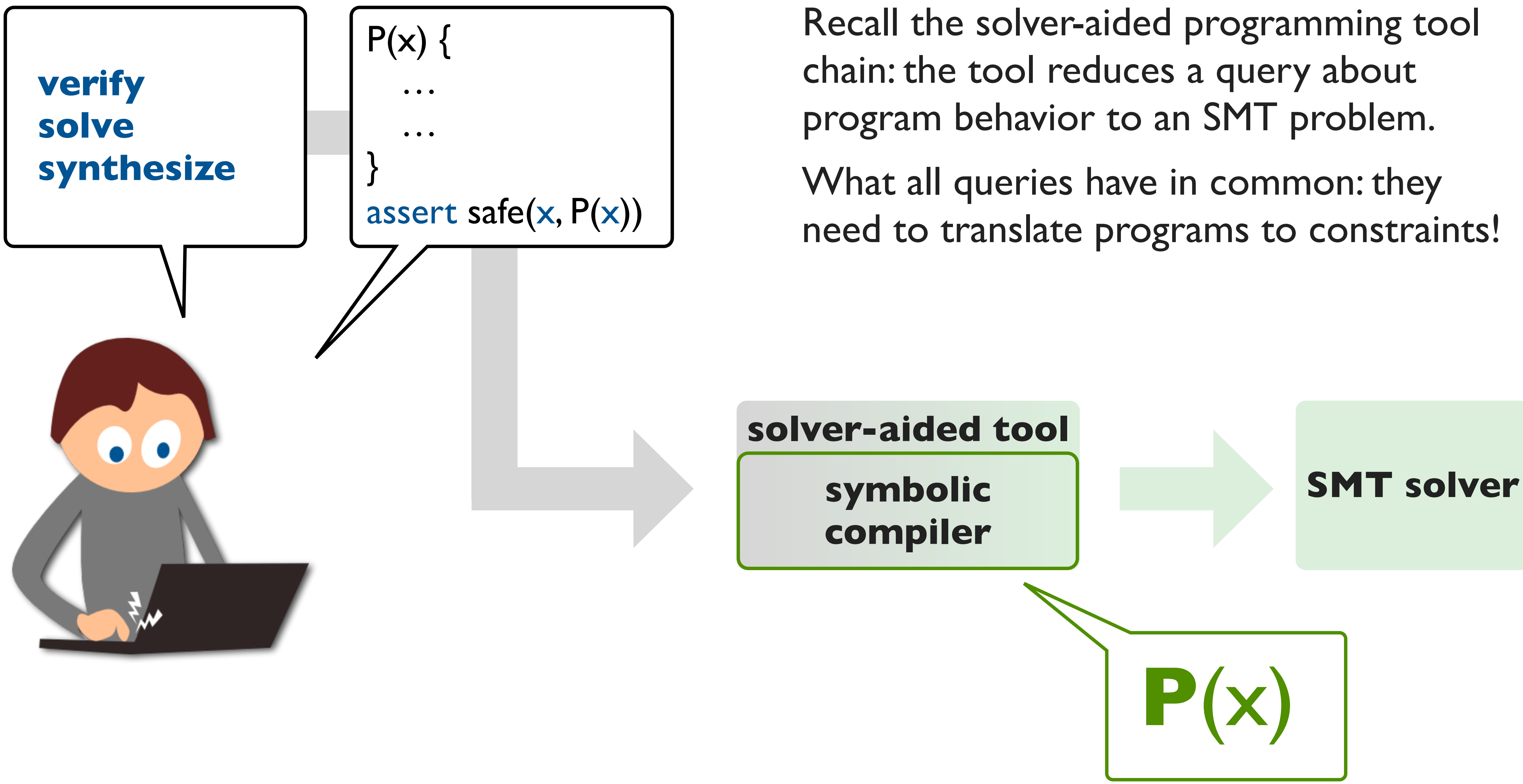
Recall the solver-aided programming tool chain: the tool reduces a query about program behavior to an SMT problem.



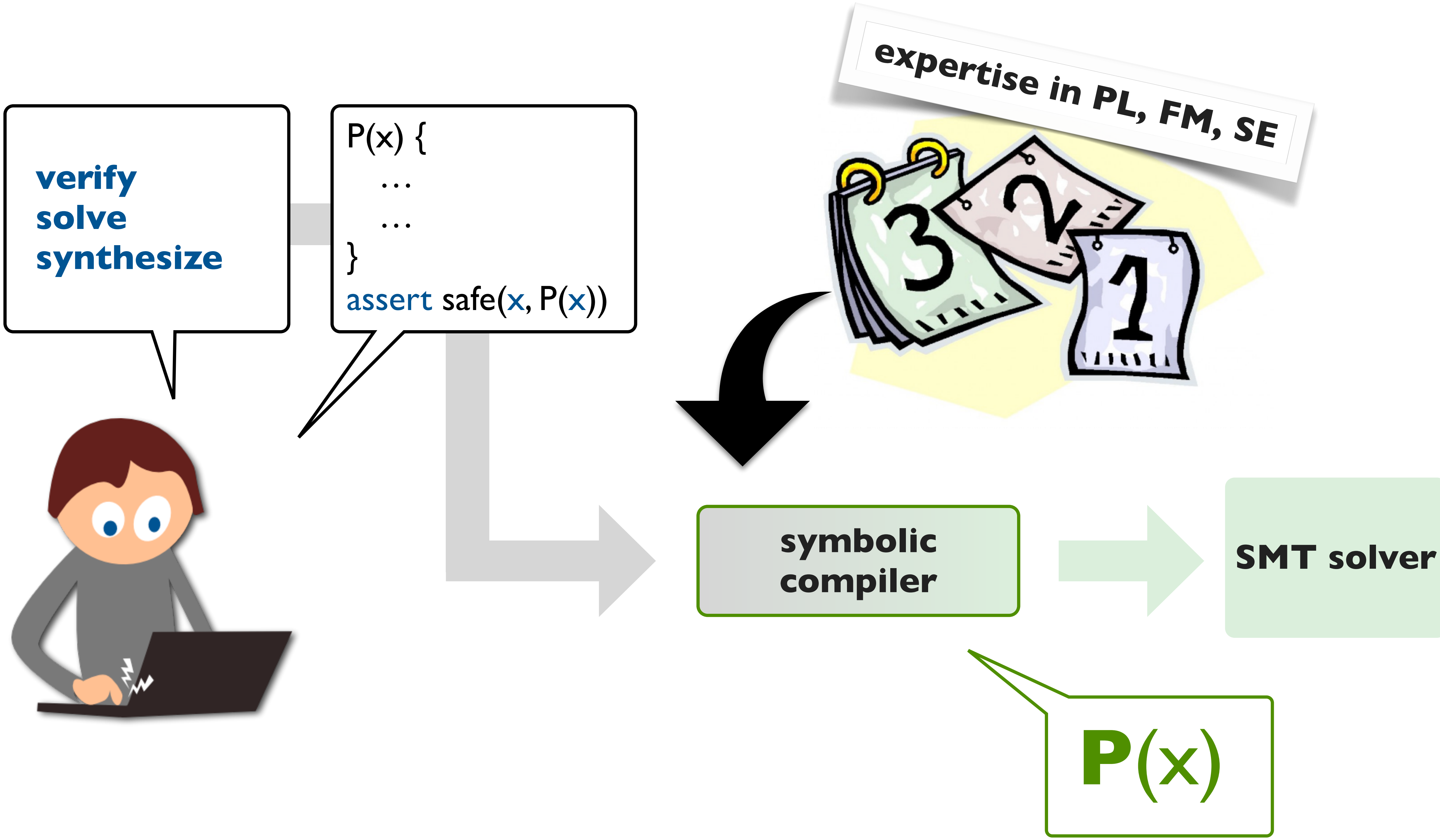
The classic (hard) way to build a tool

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

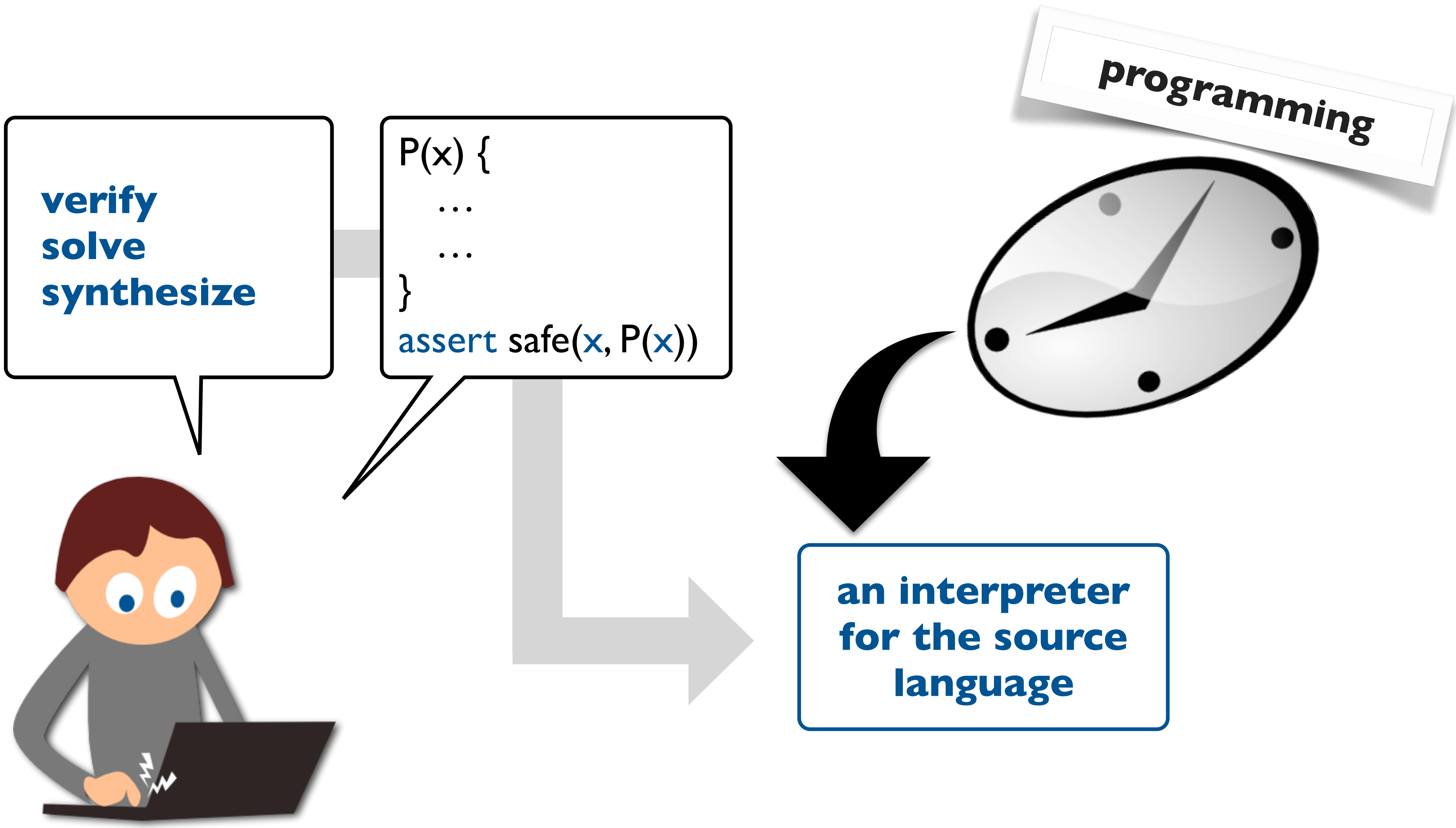
What all queries have in common: they need to translate programs to constraints!



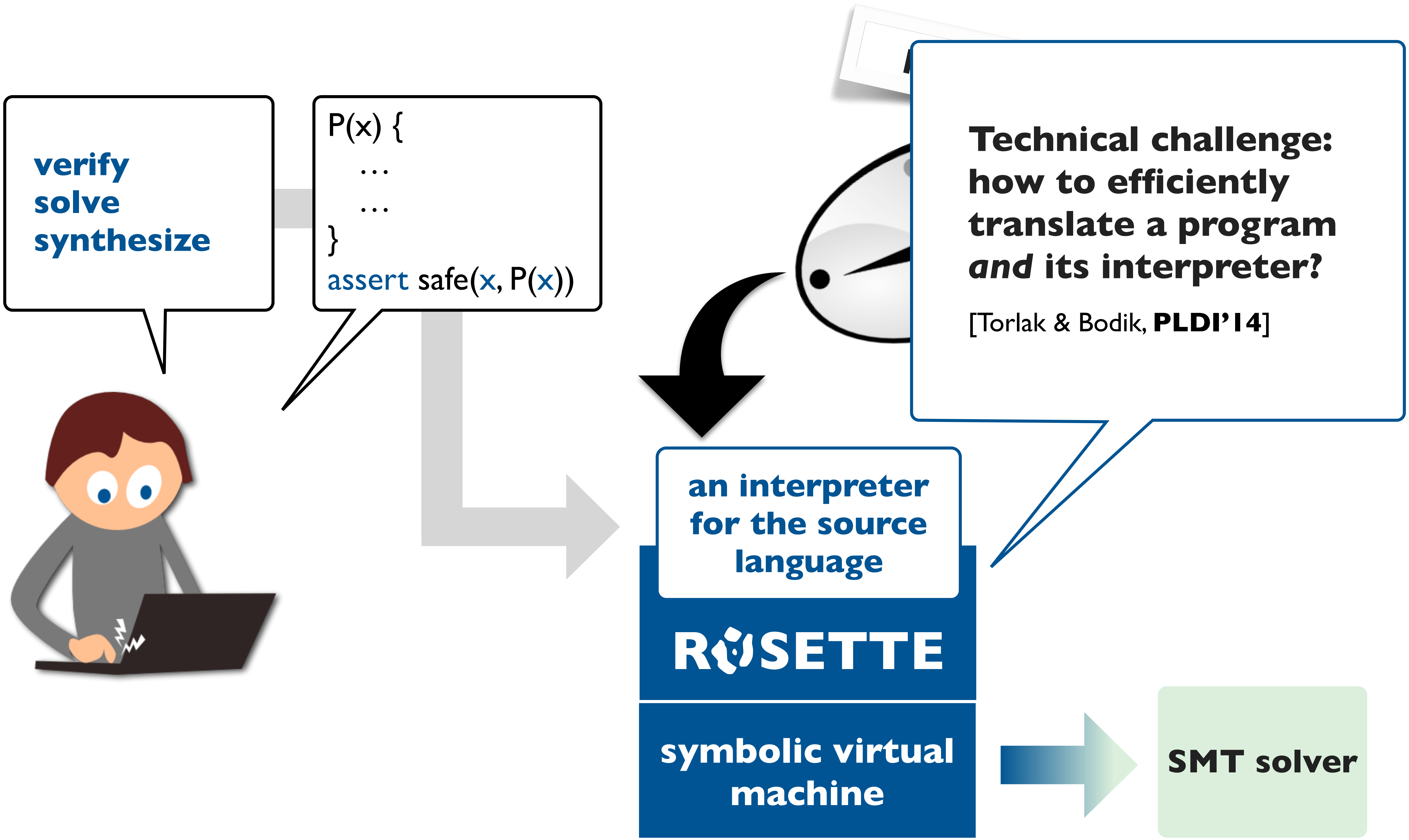
The classic (hard) way to build a tool



Wanted: an easier way to build tools



Wanted: an easier way to build tools

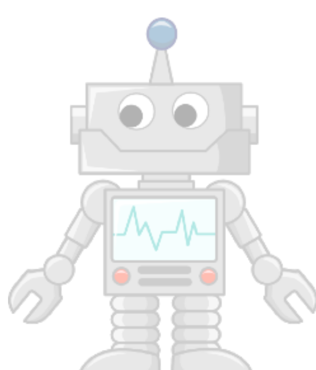


How to build your own solver-aided tool or language

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

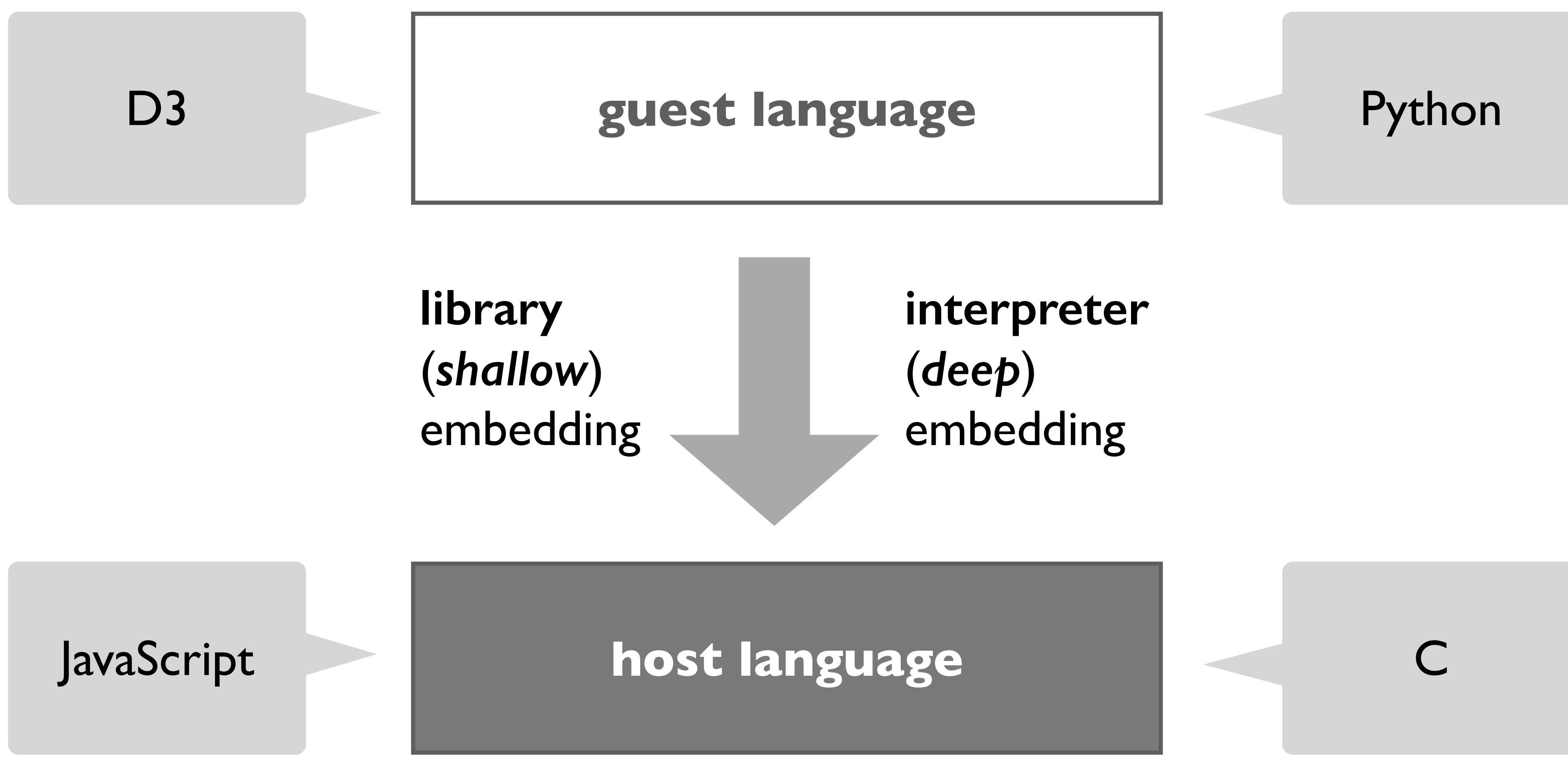
An easier way: tools as languages
How to build tools by stacking layers of languages.

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

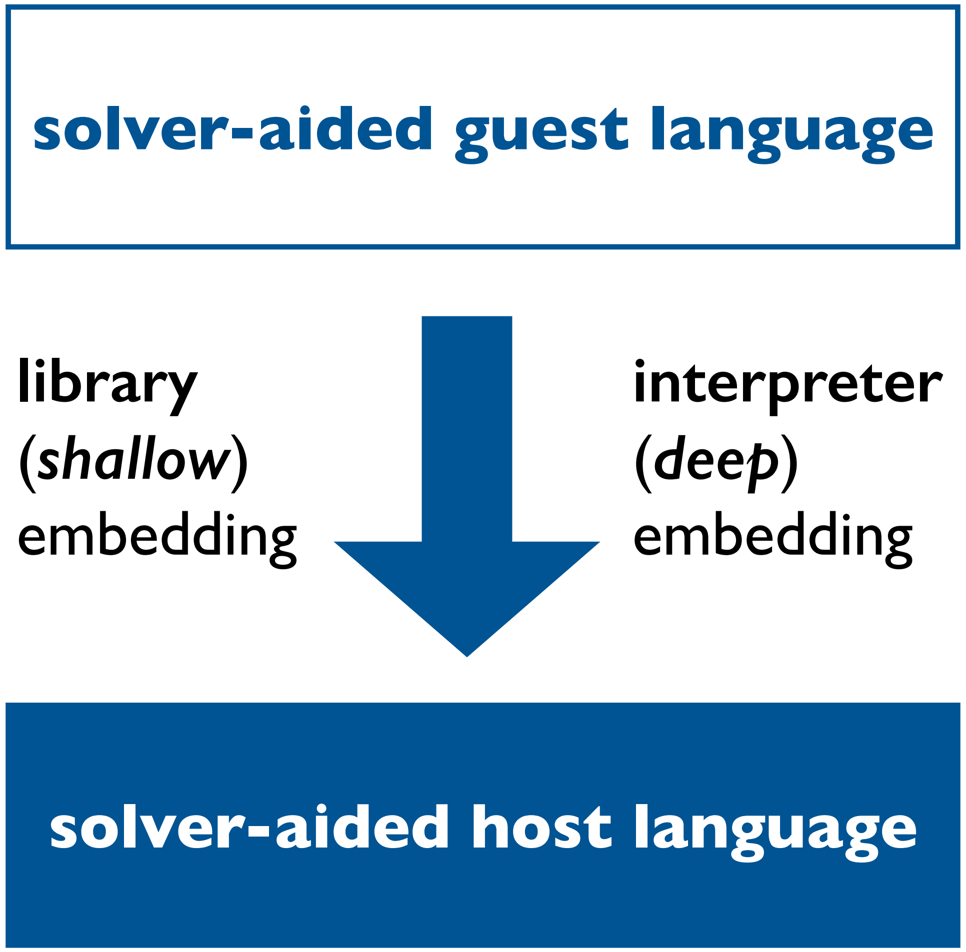


A last look: a few recent applications
Cool tools built with Rosette!

Layers of classic languages: guests and hosts



Layers of solver-aided languages



Layers of solver-aided languages

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

C (subset)

solver-aided guest language

BPF, x86 32, x86 64, ARM 32, ARM 64, RISC-V 32, RISC-V 64

library (*shallow*) embedding

interpreter (*deep*) embedding

ROSETTE

A tiny example solver-aided guest 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
```

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

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

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

A tiny example language

ROSETTE

```
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)
```

parse

```
(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 language

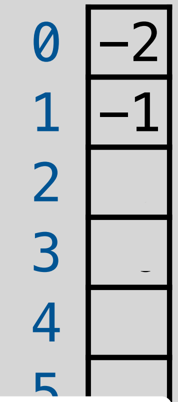
ROSETTE

```
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)
```

interpret

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




```
`(-2 -1)
```

```
(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

ROSETTE

```
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)))
```

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

```
(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

ROSETTE

```
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))))
```

A tiny example language

ROSETTE

```
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 (max x y)
 (if (bvsge x y) x y))
```

query

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

A tiny example language

ROSETTE

```
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

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))))
```

A tiny example language

ROSETTE

```
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

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

ROSETTE

```
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

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))))
```

(**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.

A tiny example language

ROSETTE

```
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]
```

query

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

A tiny example language

ROSETTE

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

```
> synthesize(bvmax, max)
```

query

```
(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

ROSETTE

```
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)
```

query

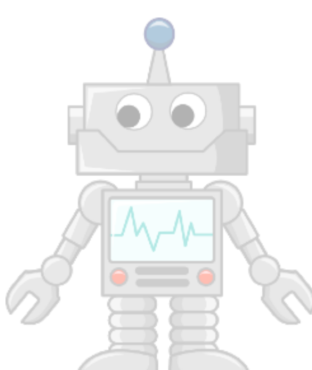
```
(define-symbolic x y int32?)
(define in (list x y))
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in)
                  (apply max in))))
```

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?

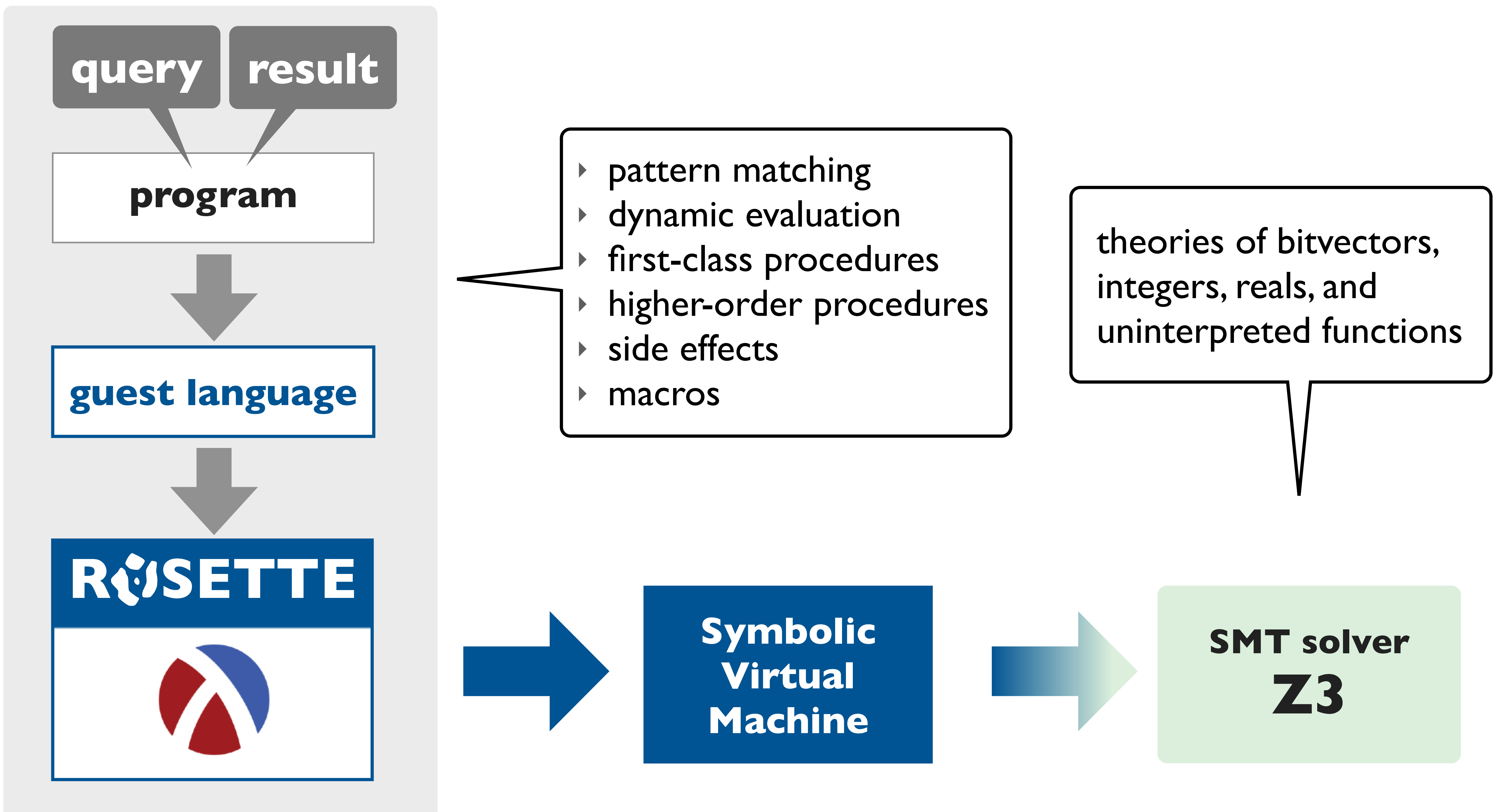
An easier way: tools as languages
How to build tools by stacking layers of languages.

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

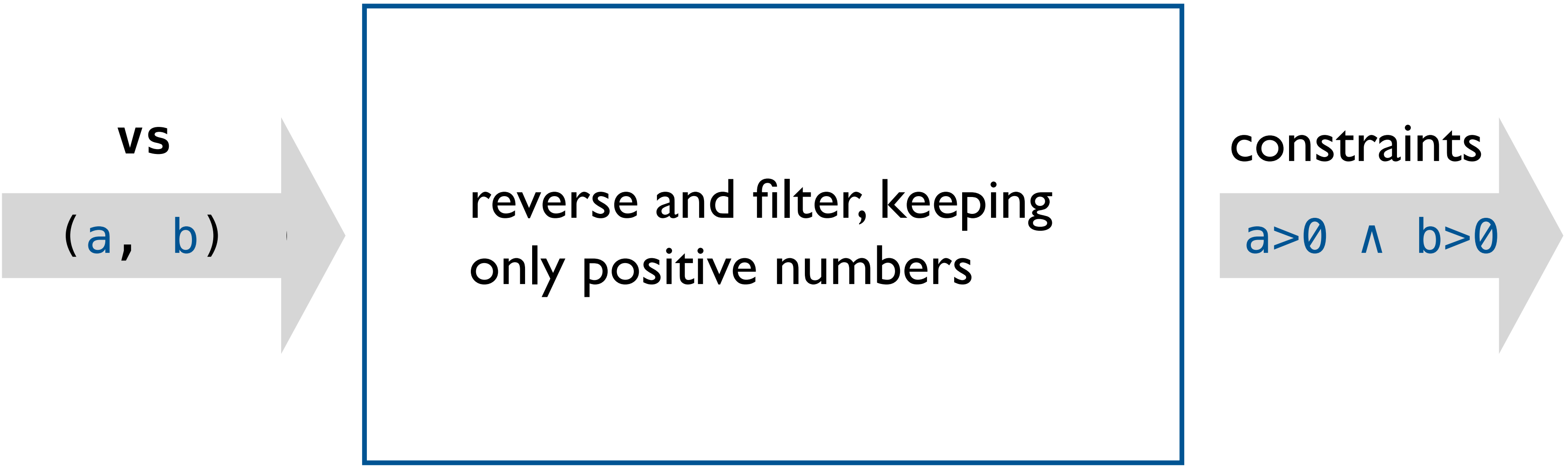


A last look: a few recent applications
Cool tools built with Rosette!

How it all works: a big picture view

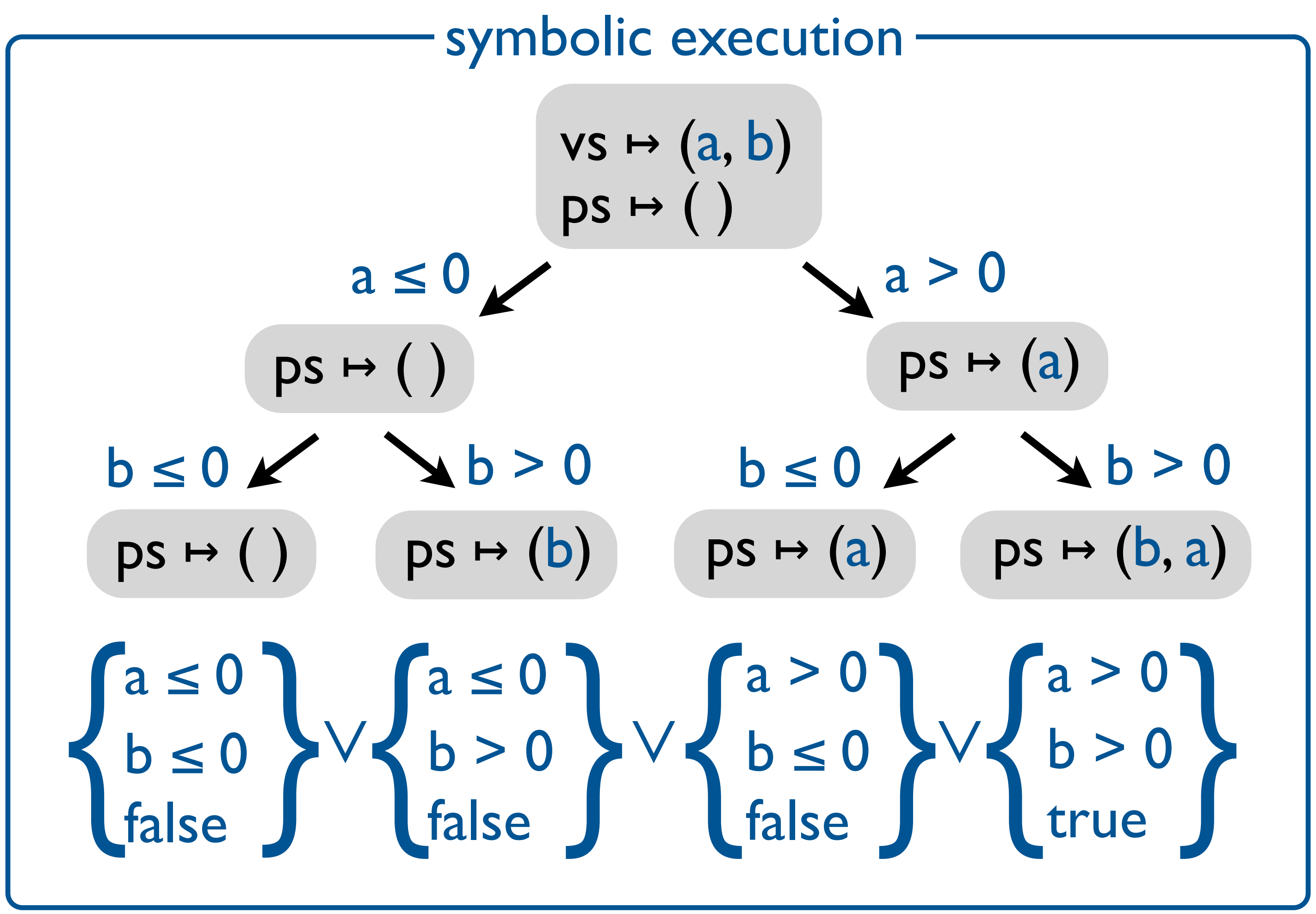


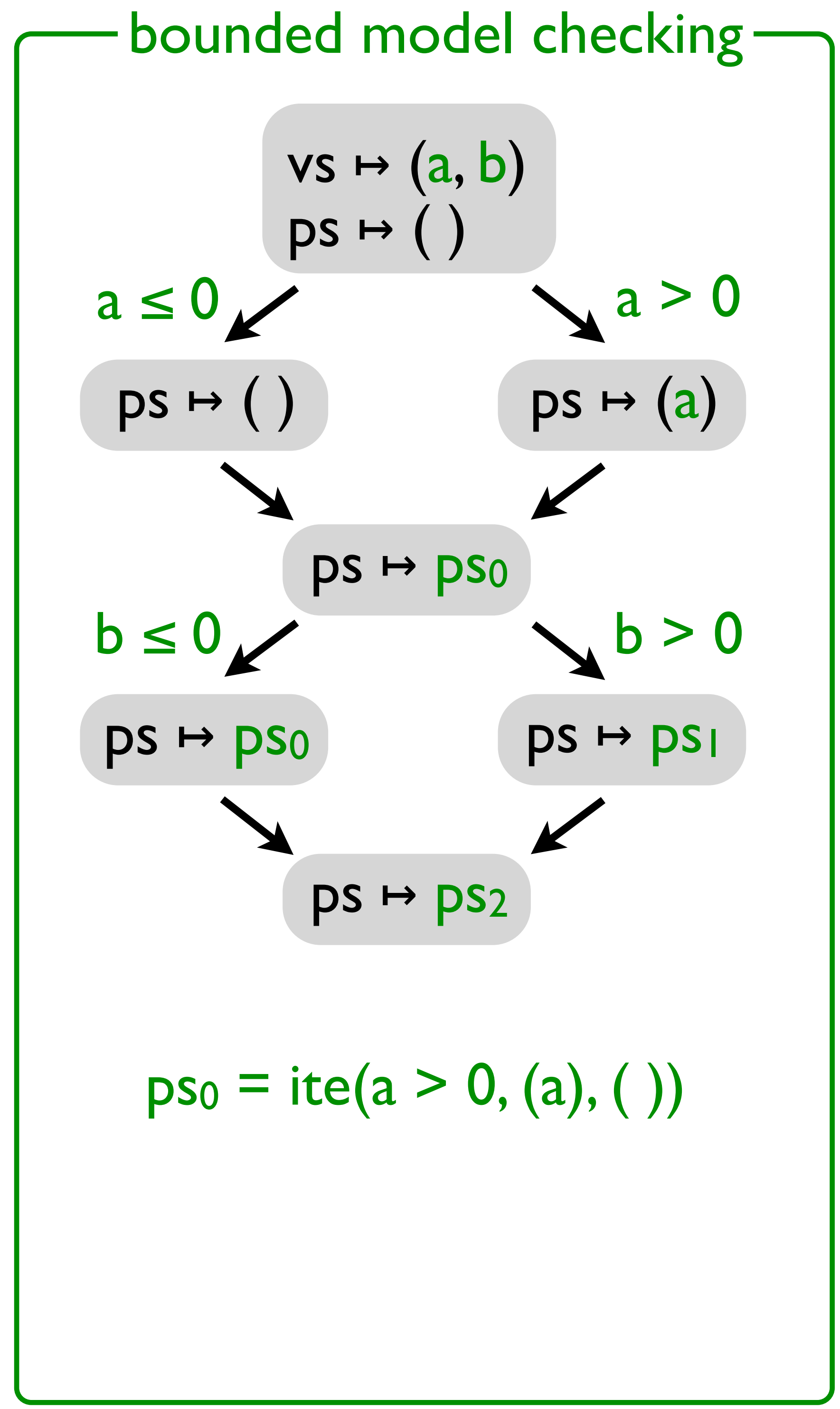
Translation to constraints by example



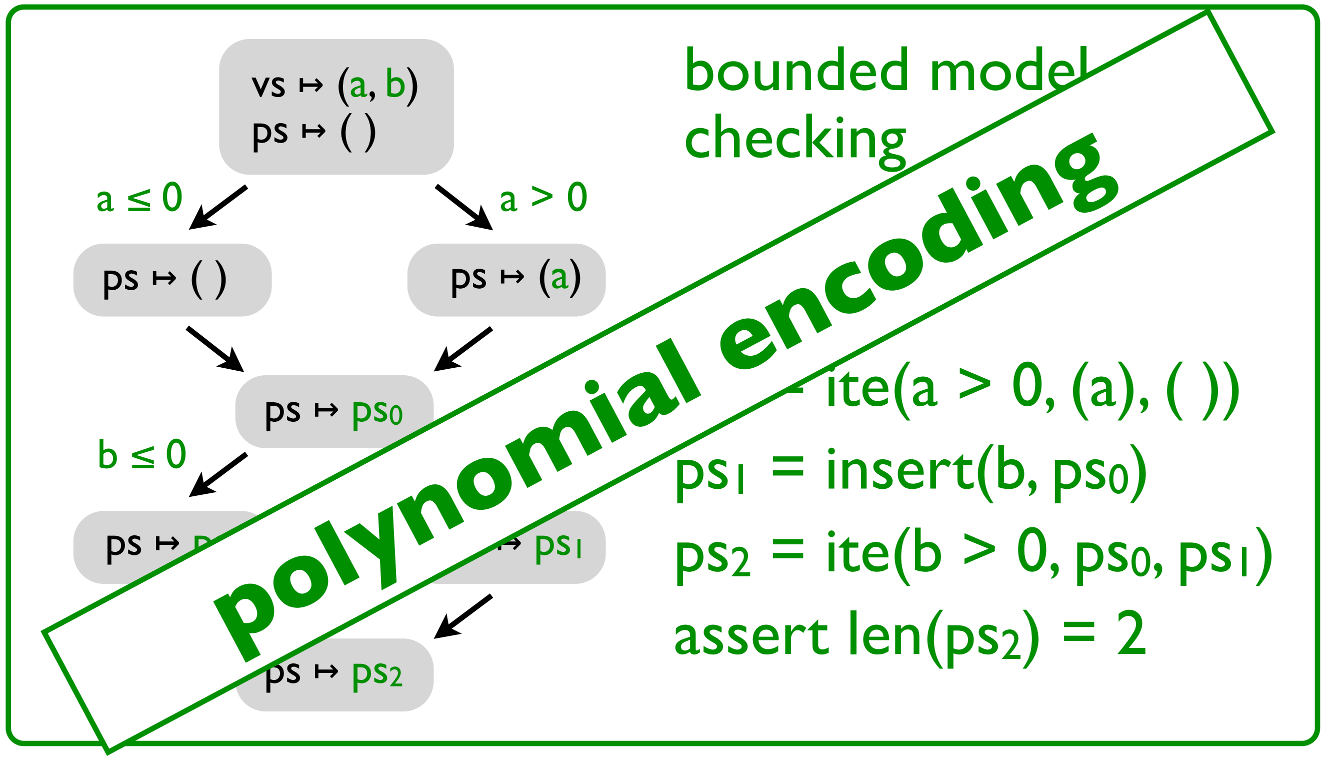
Design space of precise symbolic encodings

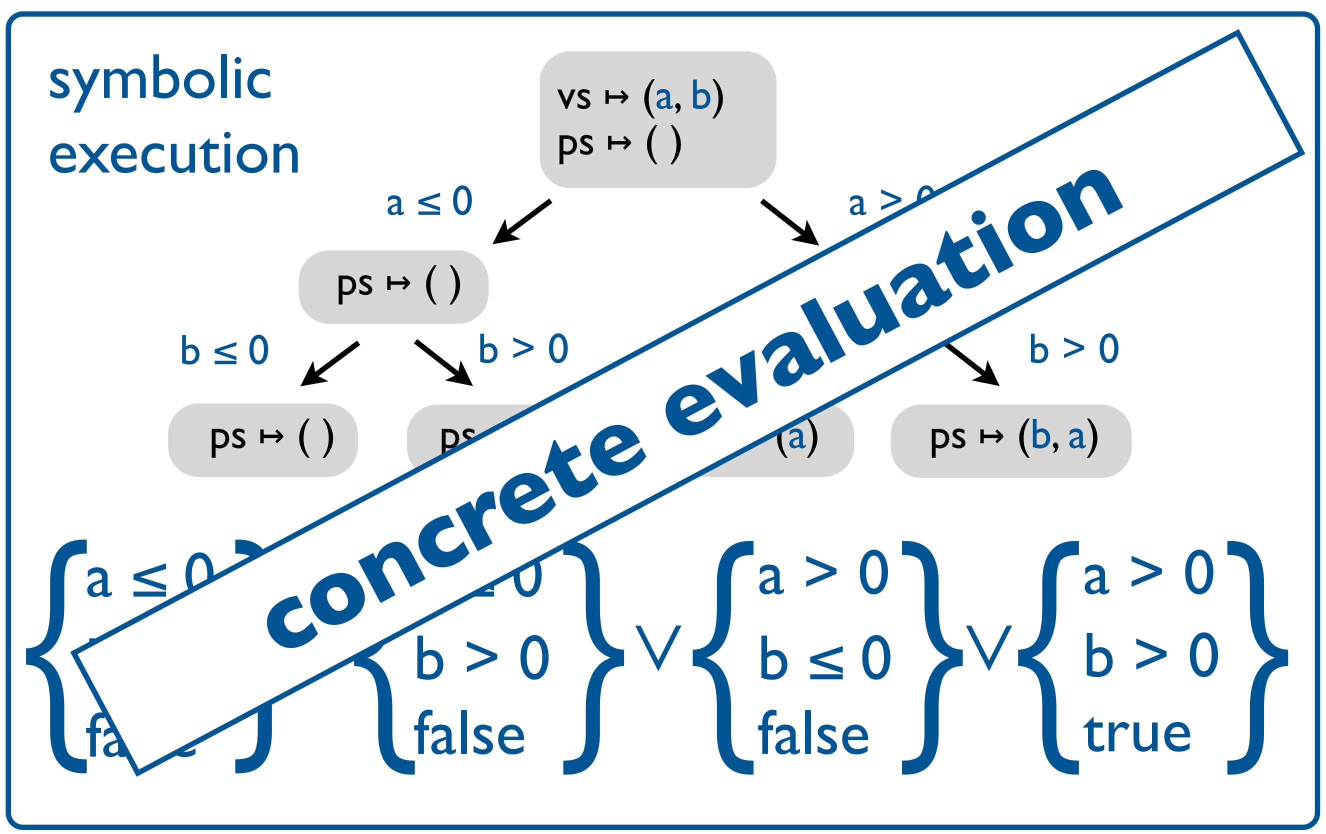
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```





Challenge: simple vs compact encoding (SE and BMC)

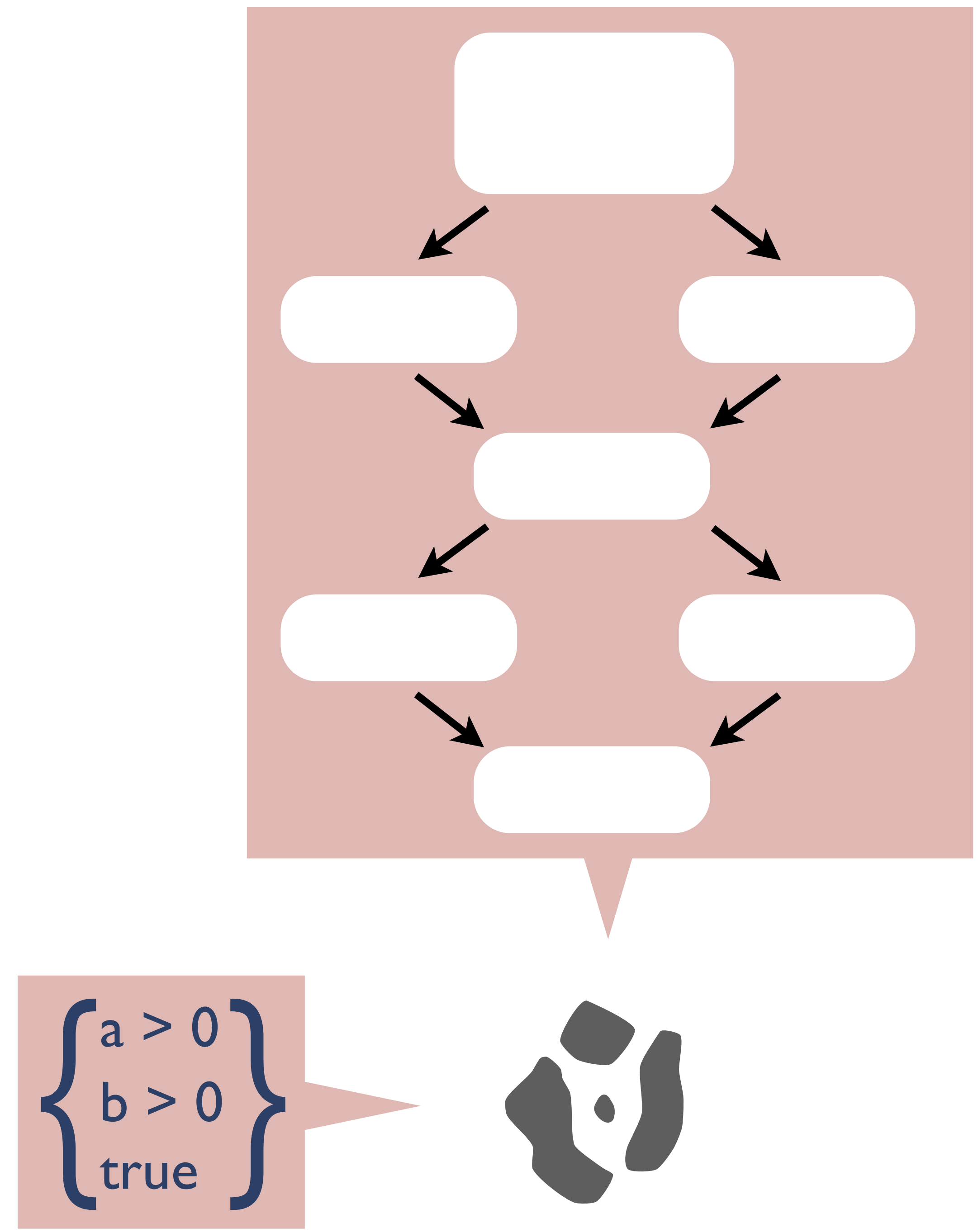




Can we have *both* a polynomially sized encoding (like BMC) and concrete evaluation of complex operations (like SE)?

Solution: type-driven state merging

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

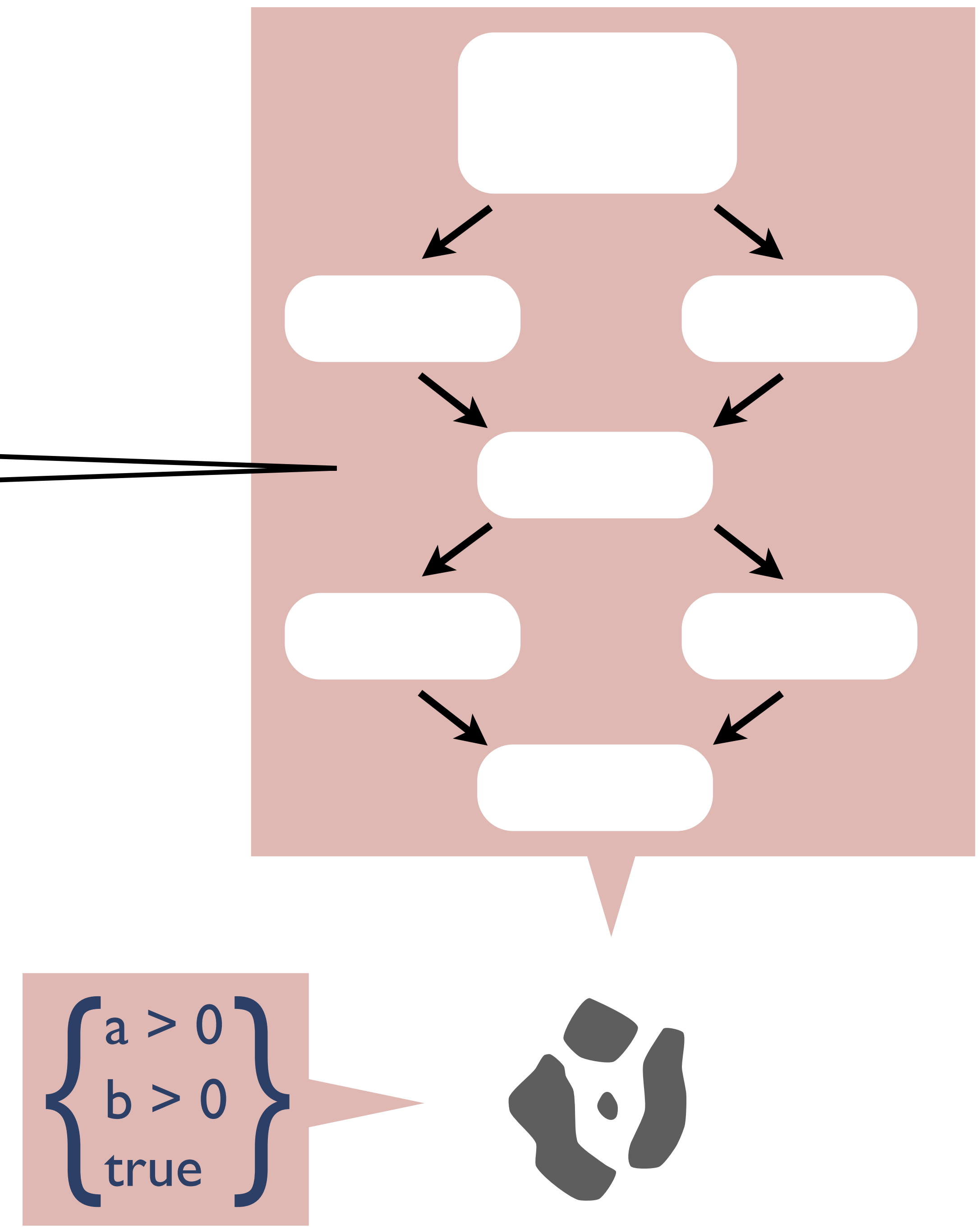


Solution: type-driven state merging

```
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

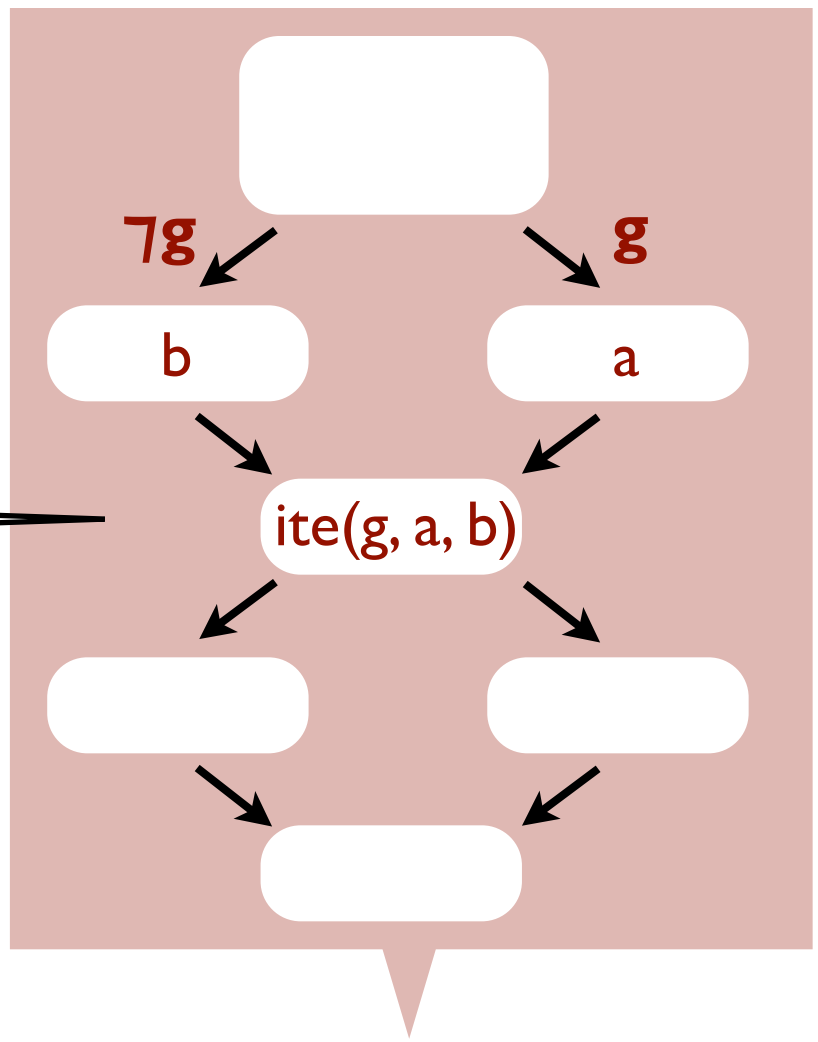


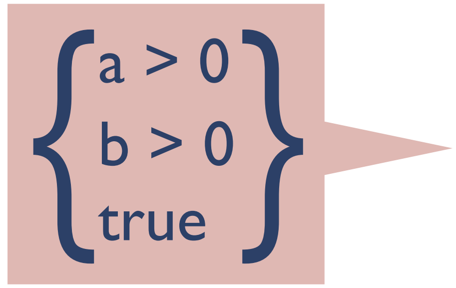
Solution: type-driven state merging

```
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



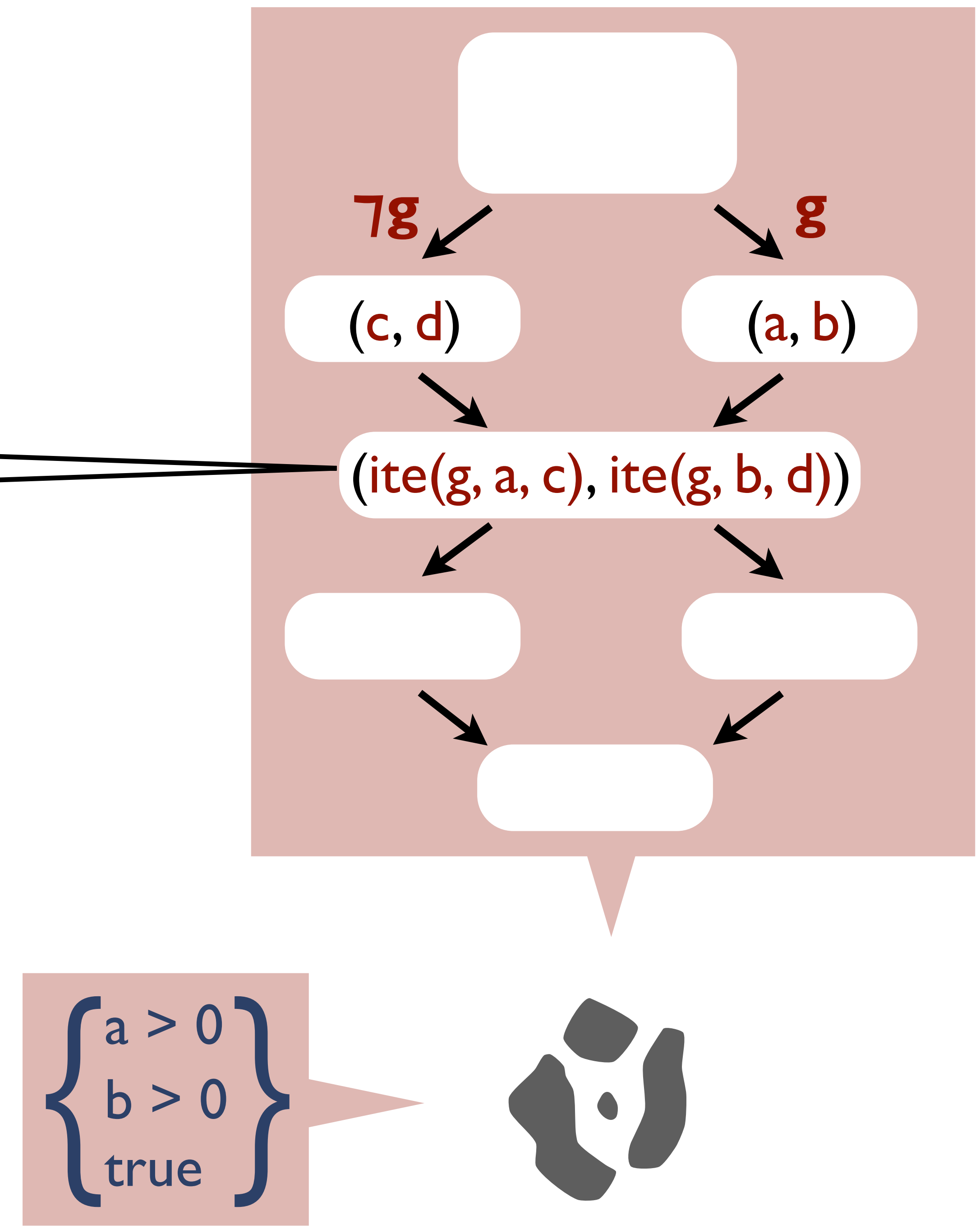


Solution: type-driven state merging

```
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

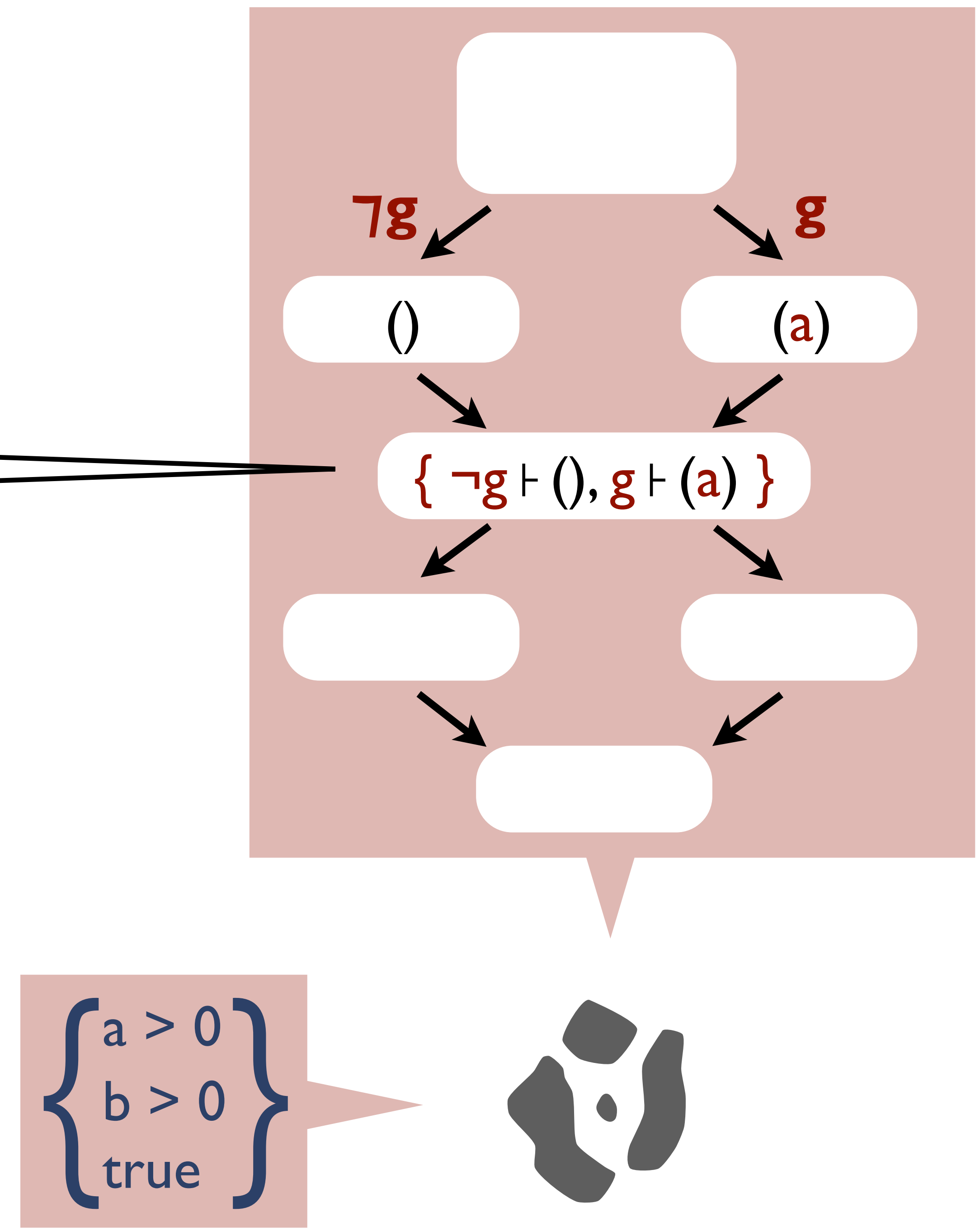


Solution: type-driven state merging

```
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



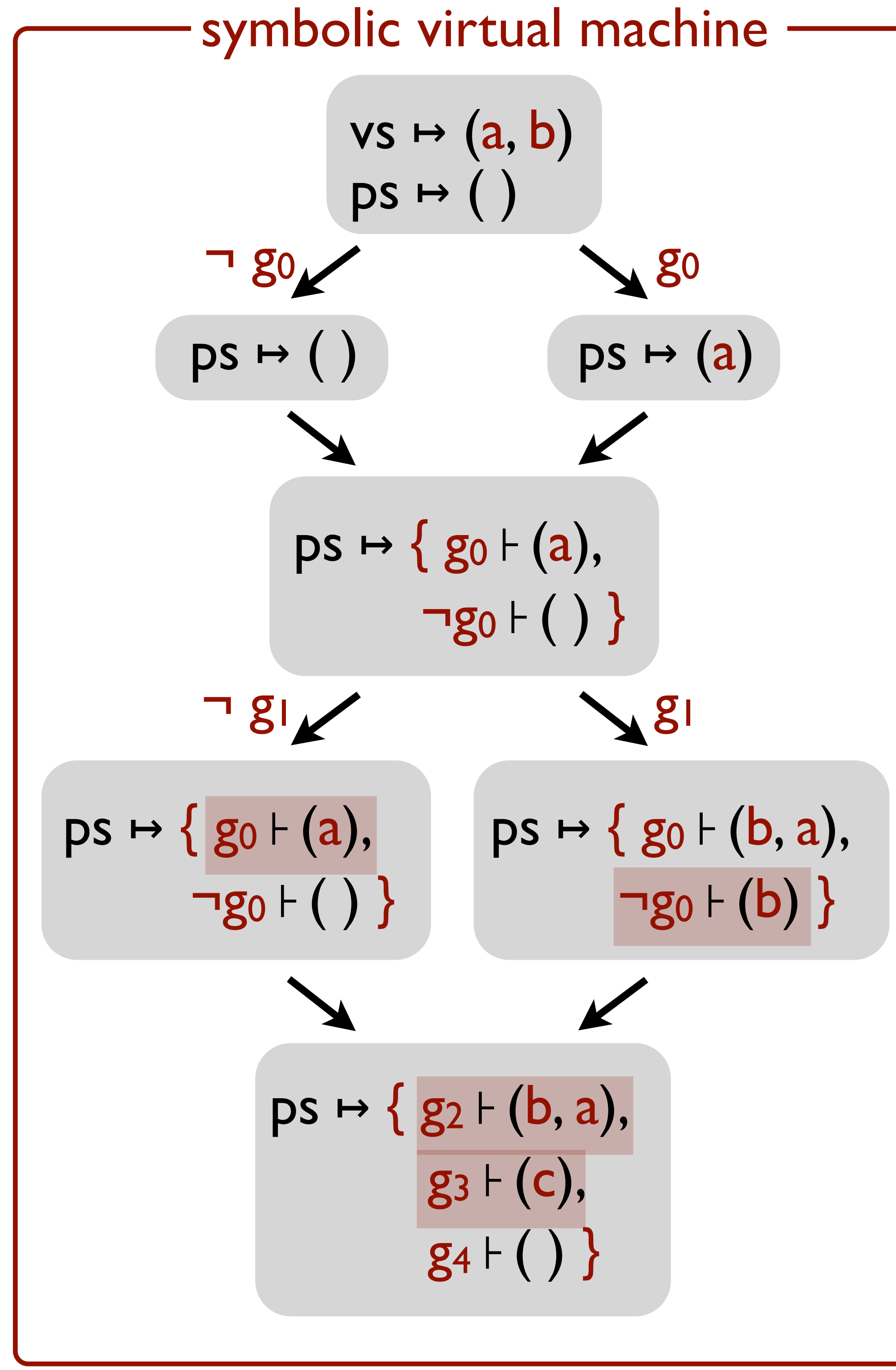
Solution: type-driven state merging

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

Execute `insert` concretely on all lists in the union.

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

$g_0 = a > 0$



Solution: type-driven state merging

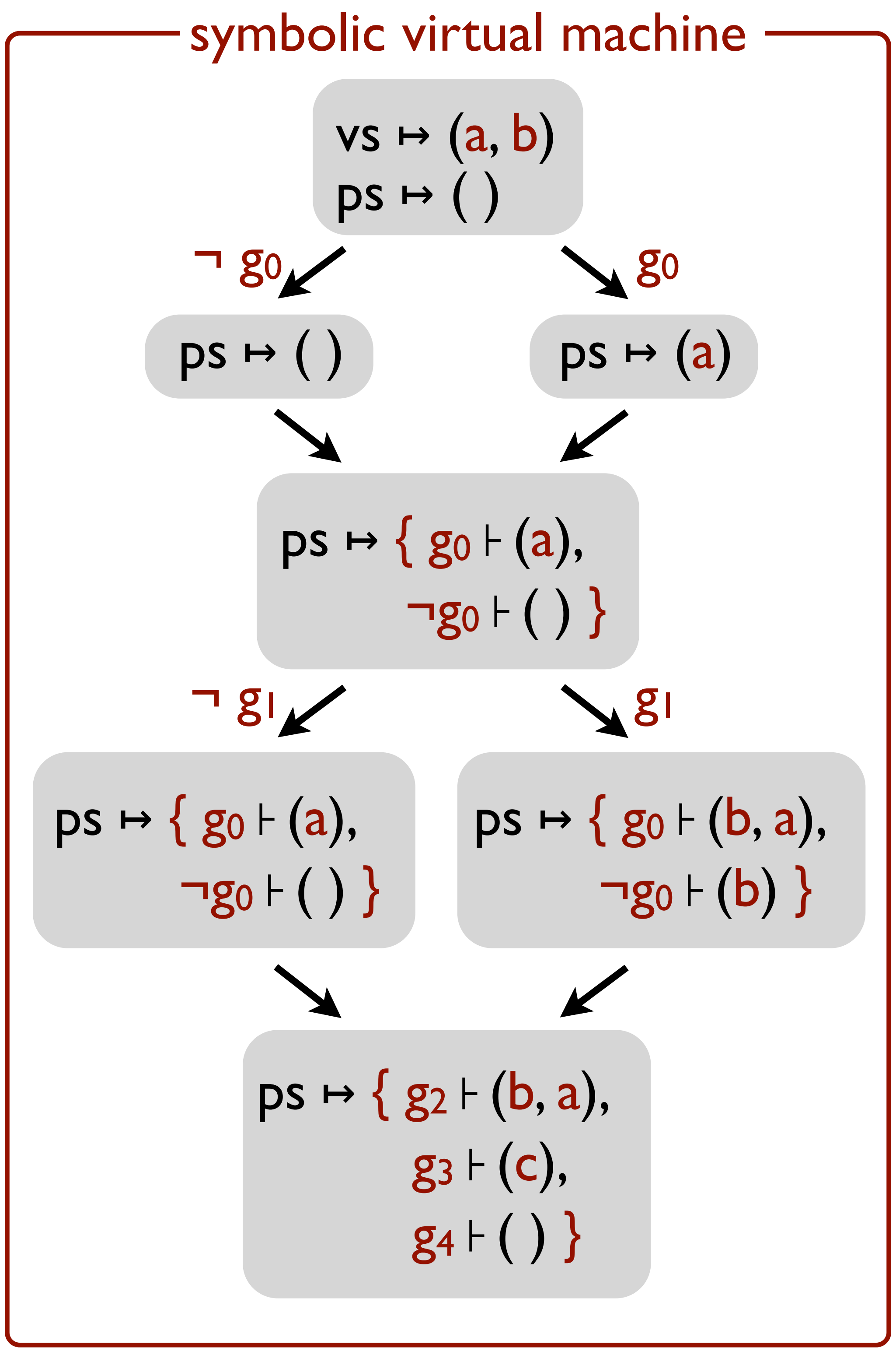
SymPro (OOPSLA'18): use **symbolic profiling** to find performance bottlenecks in solver-aided code.

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

polynomial encoding

concrete evaluation

$g_0 = a > 0$
$g_1 = b > 0$
$g_2 = g_0 \wedge g_1$
$g_3 = \neg(g_0 \Leftrightarrow g_1)$
$g_4 = \neg g_0 \wedge \neg g_1$
$c = \text{ite}(g_1, b, a)$
assert g_2

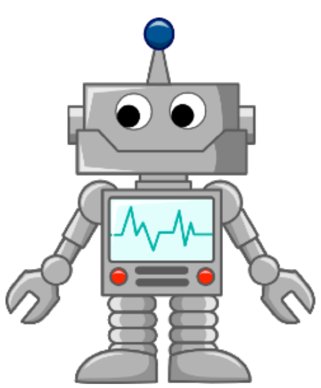


How to build your own solver-aided tool or language

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.

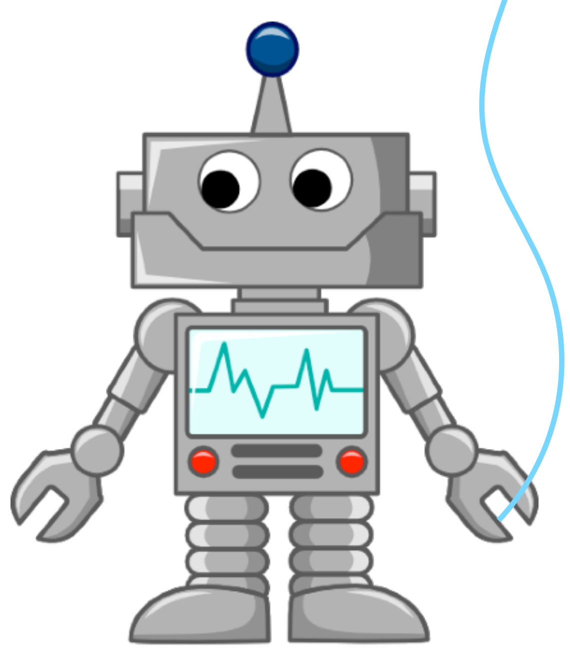
Behind the scenes: symbolic virtual machine
How Rosette works so you don't have to.



A last look: a few recent applications
Cool tools built with Rosette!

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

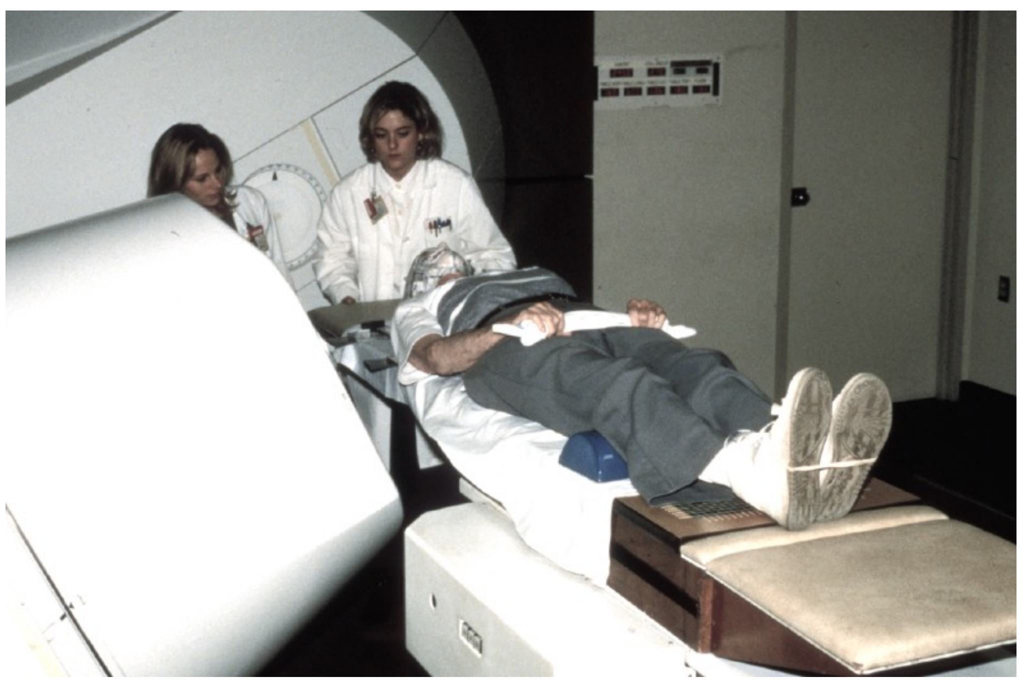
hints and feedback
problem generation
problem-solving strategies

systems, architecture, networks, security, and databases

memory models
OS components
data movement for GPUs
router configuration
cryptographic protocols

Verifying a radiation therapy system

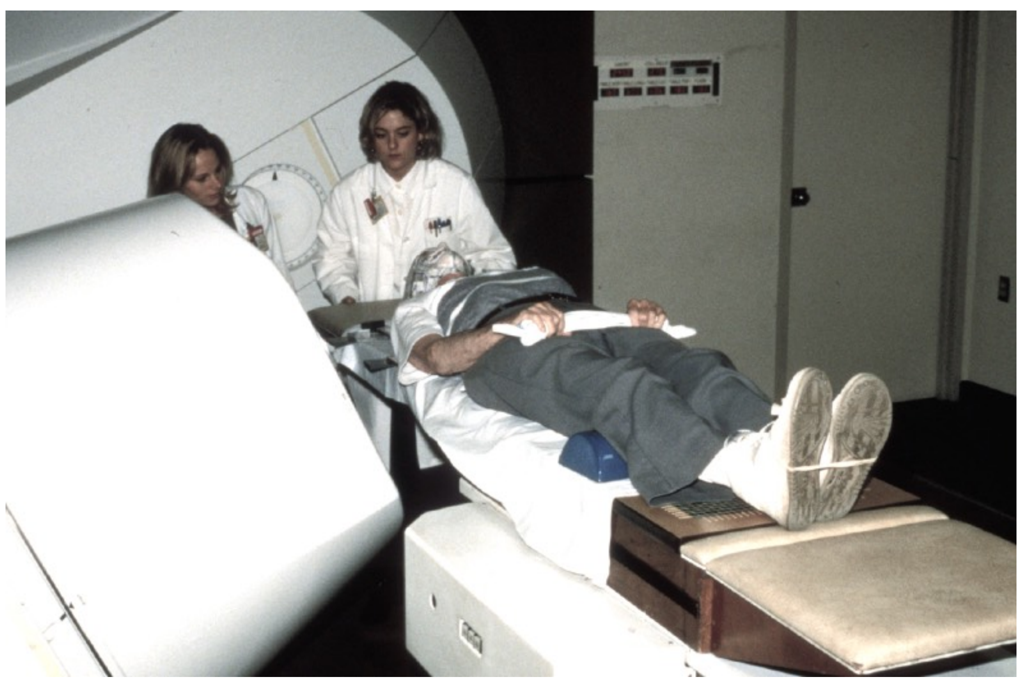
Clinical Neutron Therapy System (CNTS) at UW

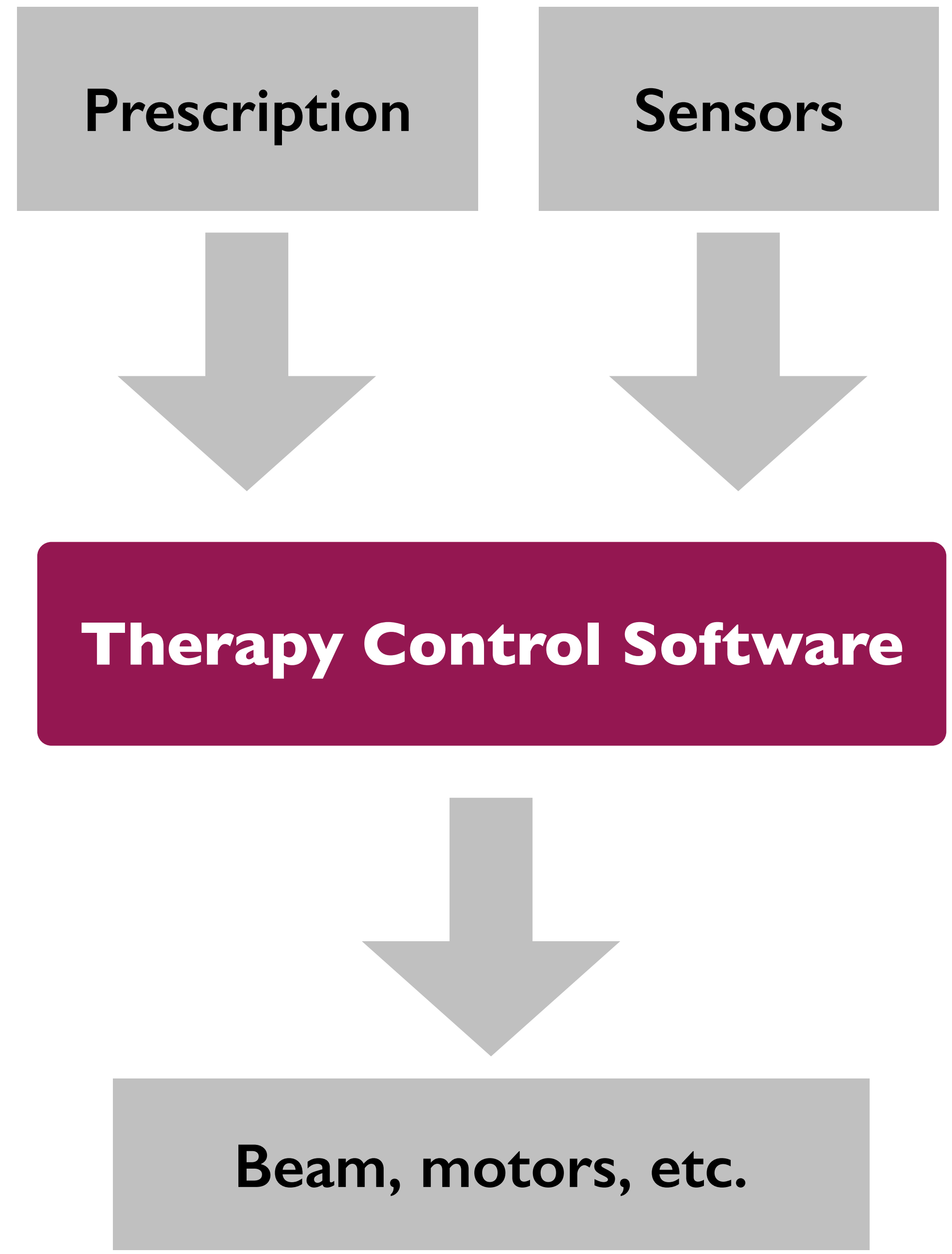


- 30 years of incident-free service.
- Controlled by custom software, built by CNTS engineering staff.
- Third generation of Therapy Control software built recently.

Verifying a radiation therapy system

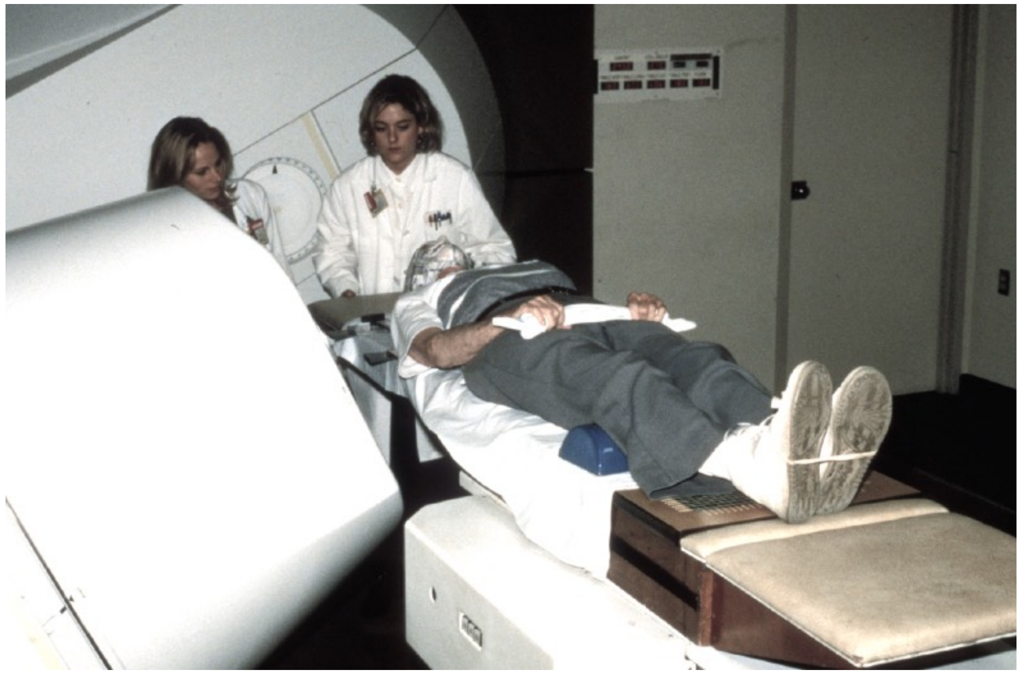
Clinical Neutron Therapy System (CNTS) at UW





Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW

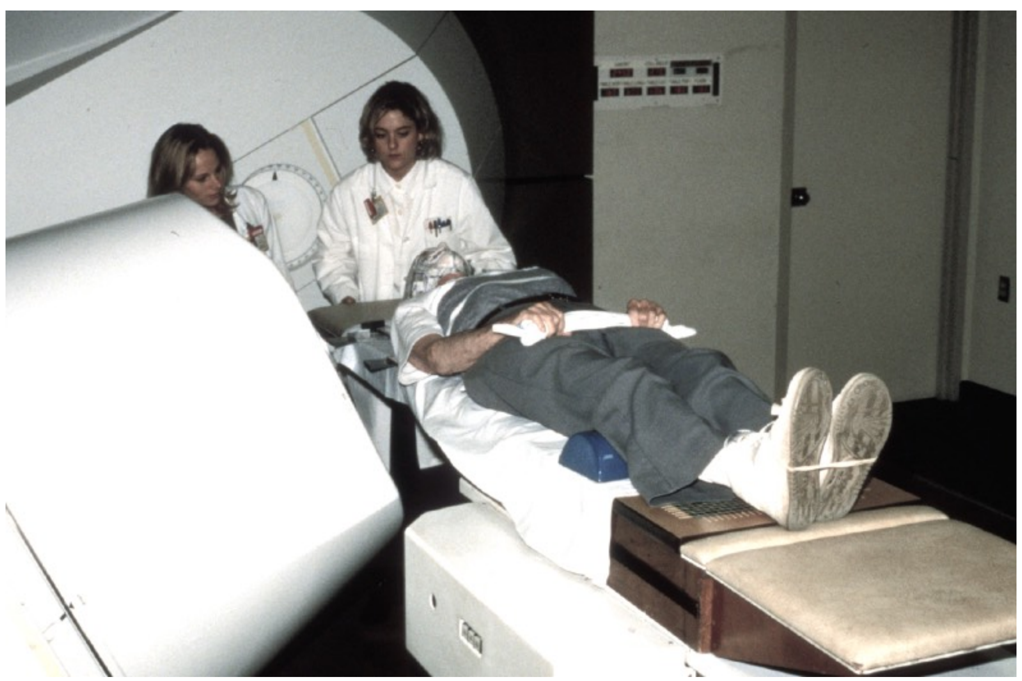


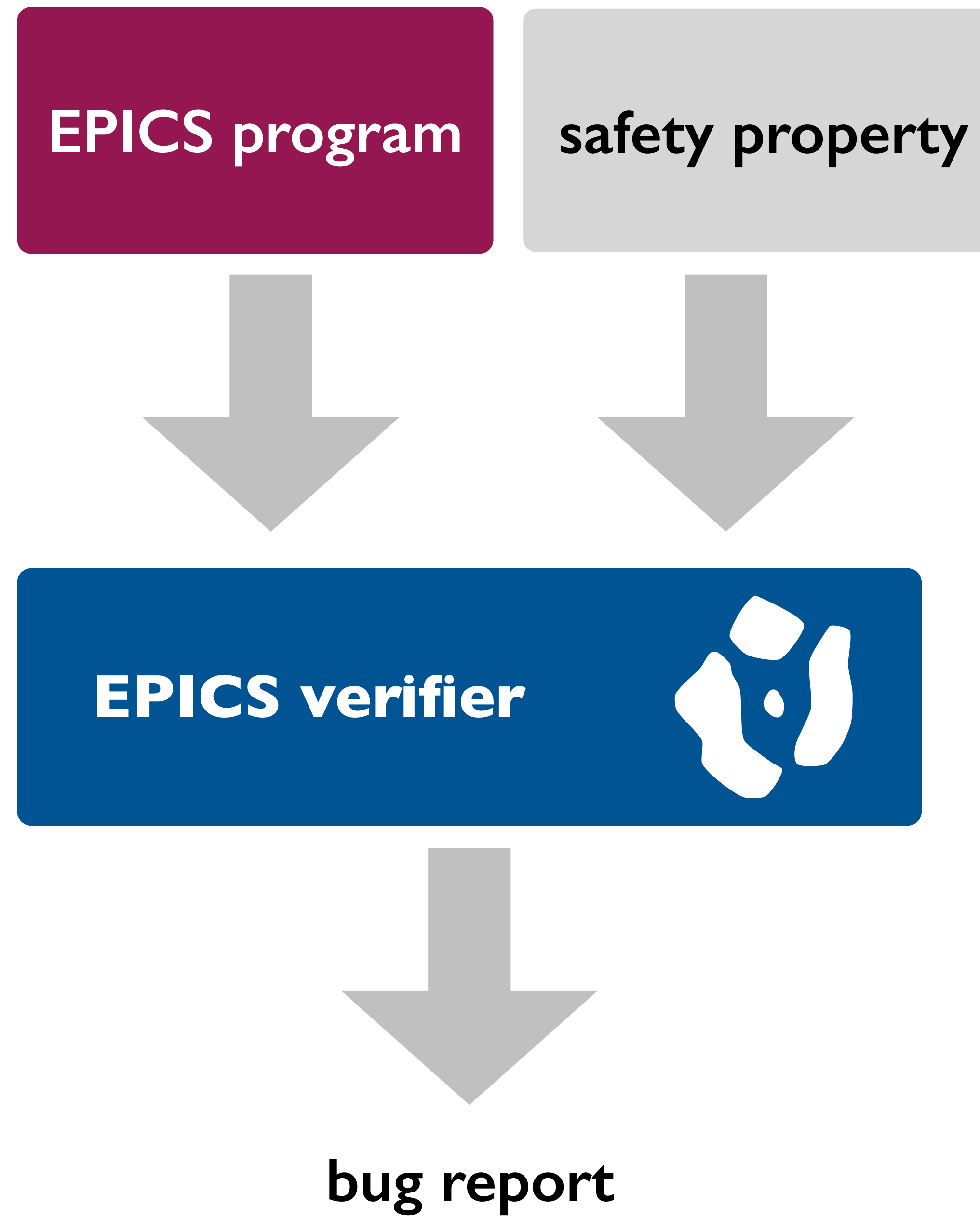
Experimental Physics and Industrial Control System (EPICS) Dataflow Language

Therapy Control Software

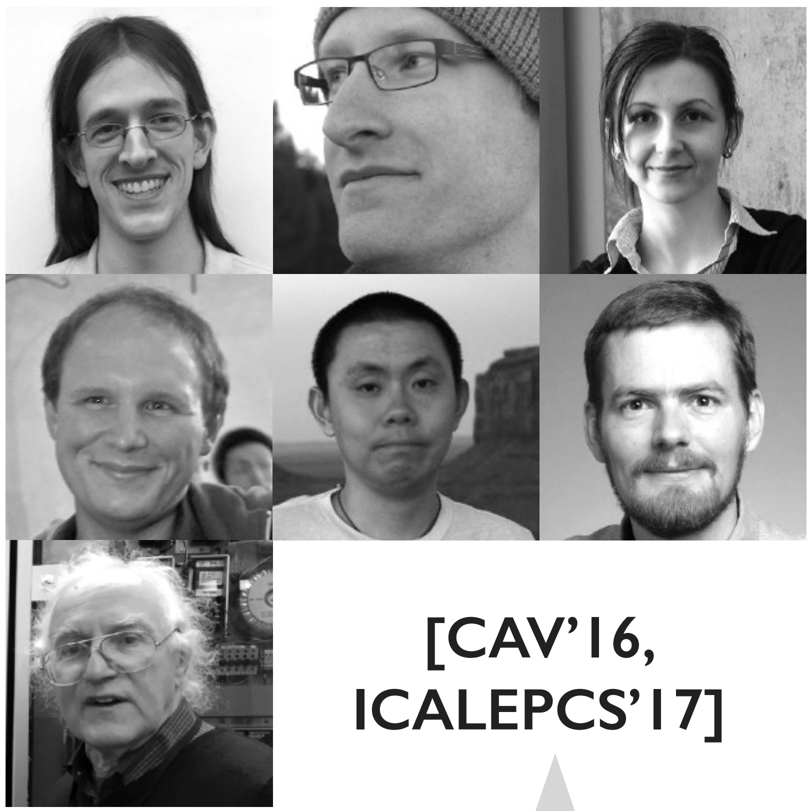
Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW





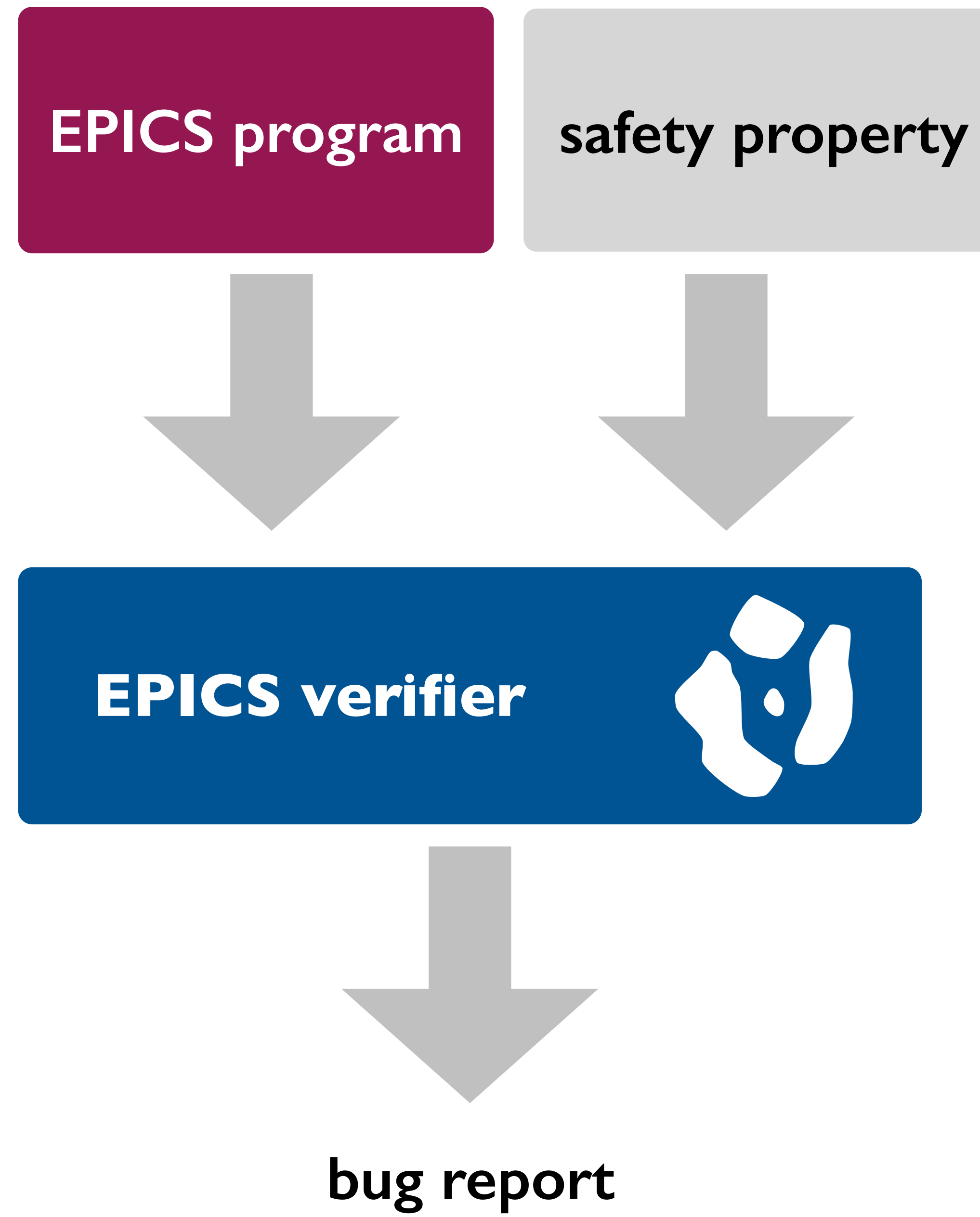
Verifying a radiation therapy system



[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.



Summary

Today

- Going pro with solver-aided programming.

Next lecture

- Getting started with SAT solving!