Computer-Aided Reasoning for Software

CSE507

Solver-Aided Programming II

Emina Torlak

emina@cs.washington.edu

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.

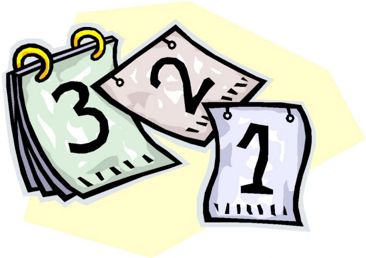
ROSETTE

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.

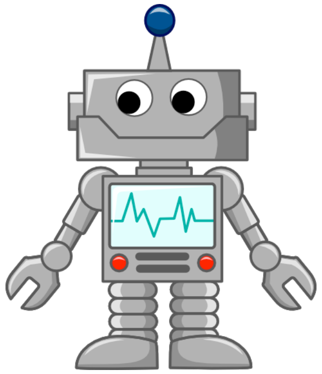
How to build your own solver-aided tool or language



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.

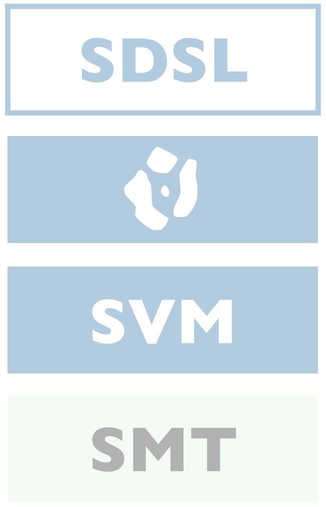
Behind the scenes: symbolic virtual machine
How Rosette works so you don’t have to.



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

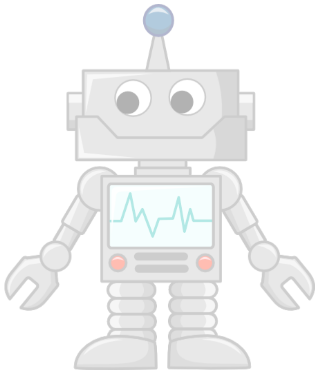
How to build your own solver-aided tool or language

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



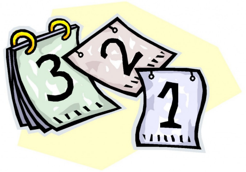
An easier way: tools as languages
How to build tools by stacking layers of languages.

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

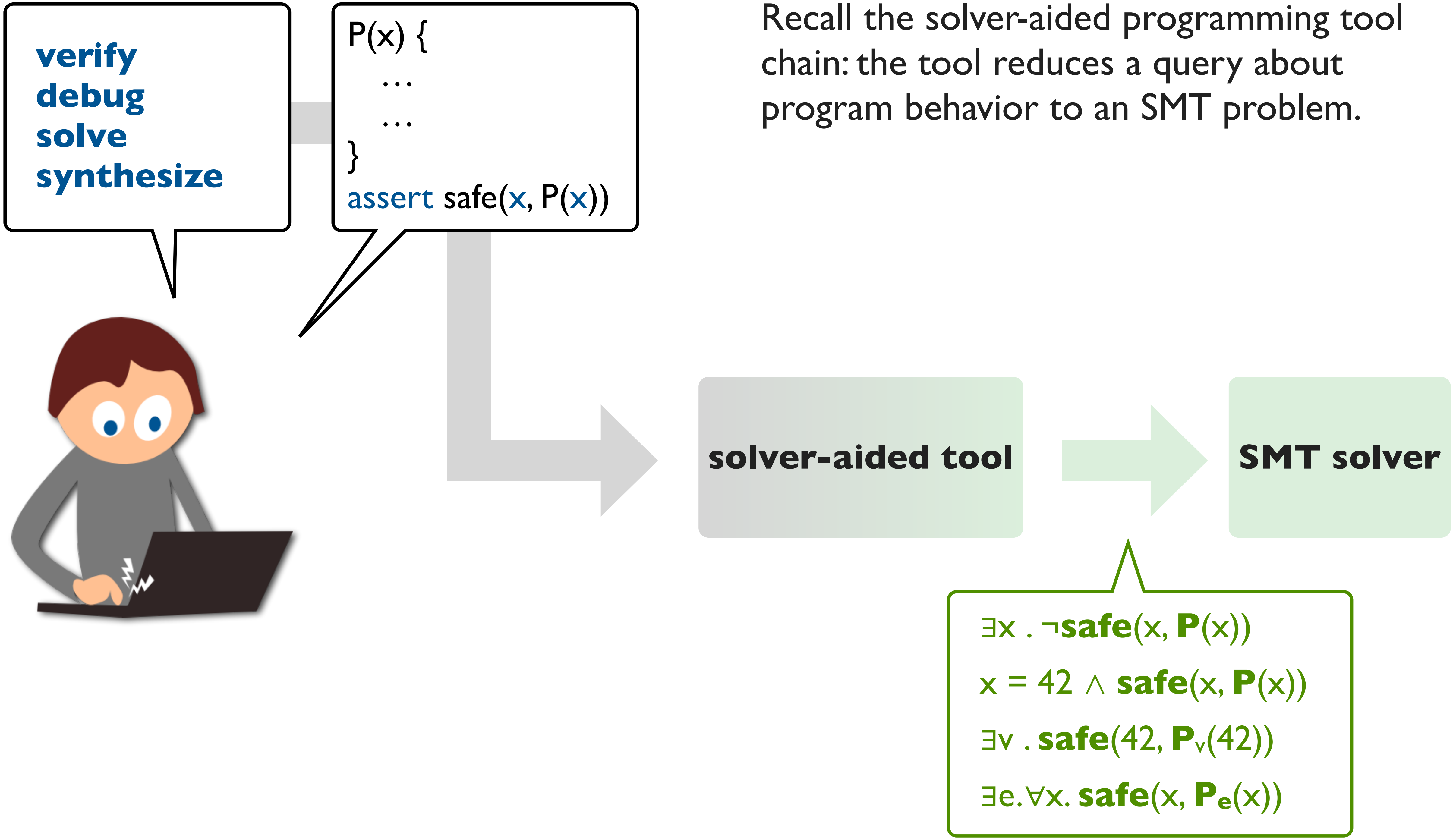


A last look: a few recent applications
Cool tools built with Rosette!

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))
```

solver-aided tool

SMT solver

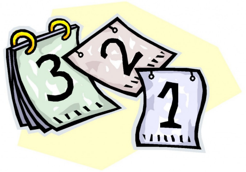
$\exists x \,.\, \neg \mathbf{safe}(x, \mathbf{P}(x))$

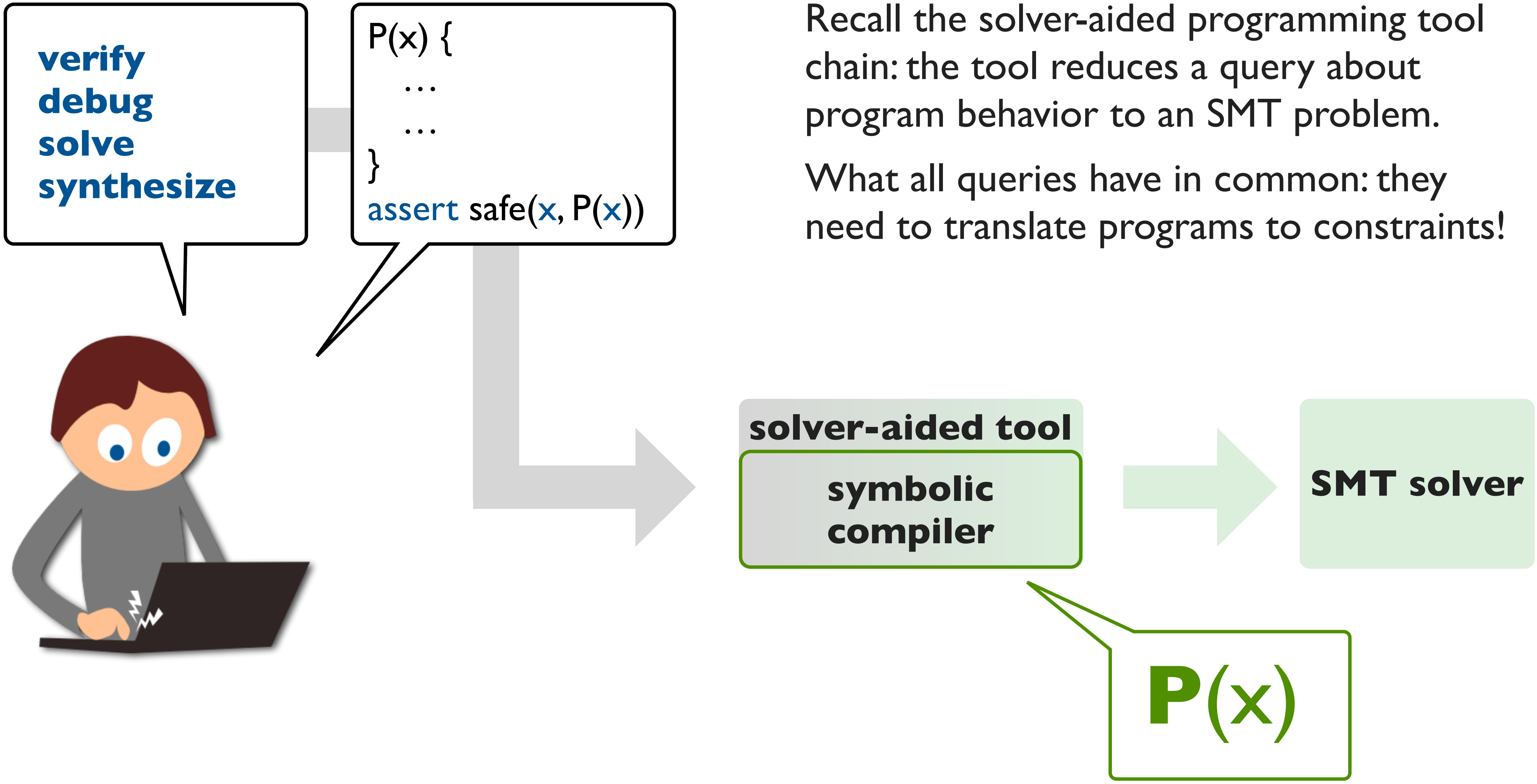
$x = 42 \wedge \mathbf{safe}(x, \mathbf{P}(x))$

$\exists v \,.\, \mathbf{safe}(42, \mathbf{P}_v(42))$

$\exists e . \forall x . \, \mathbf{safe}(x, \mathbf{P}_e(x))$

The classic (hard) way to build a tool



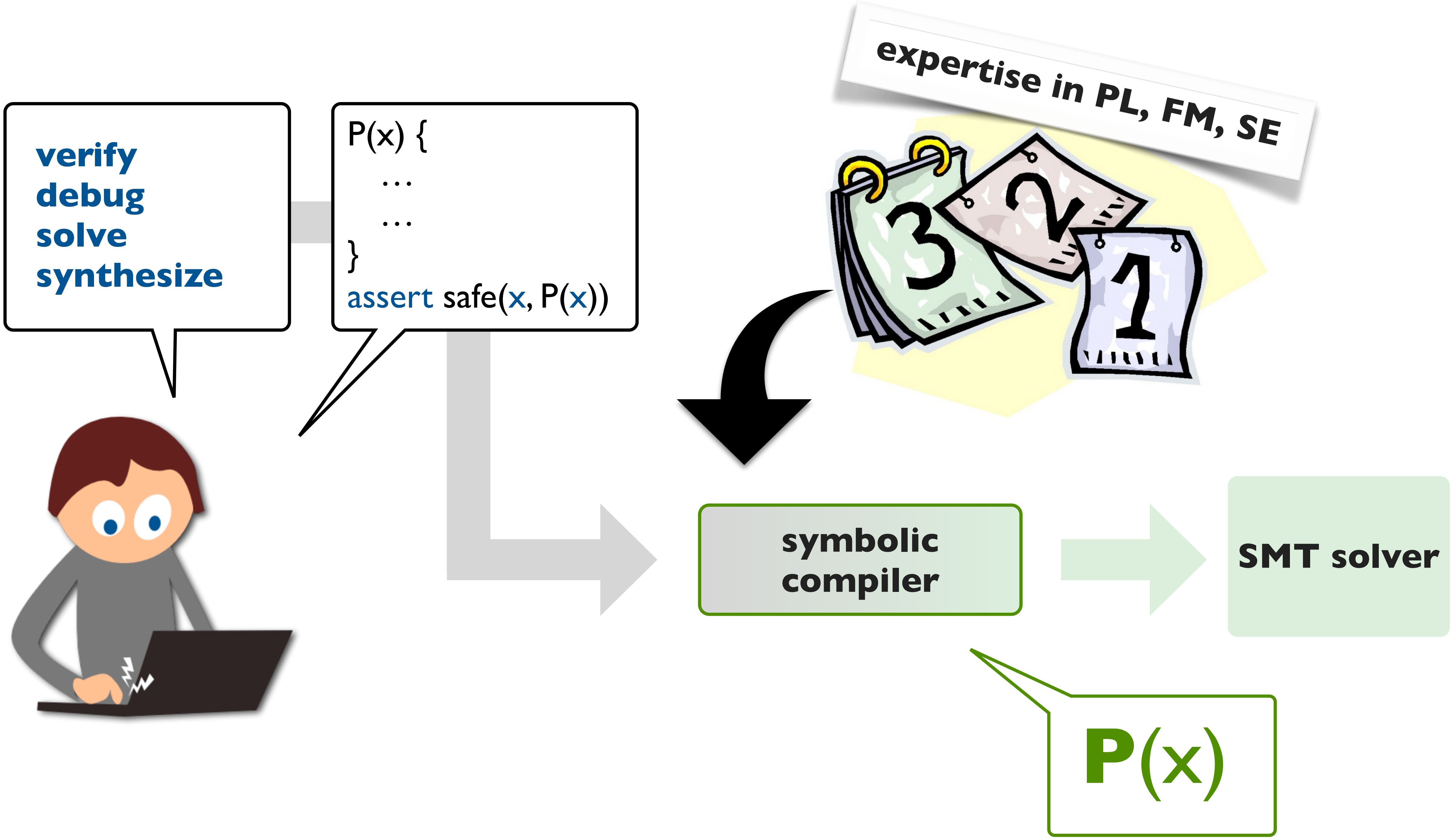


```
P(x) {
   …
   …
}
assert safe(x, P(x))
```

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



verify
debug
solve
synthesize

```
P(x) {
    …
    …
}
assert safe(x, P(x))
```

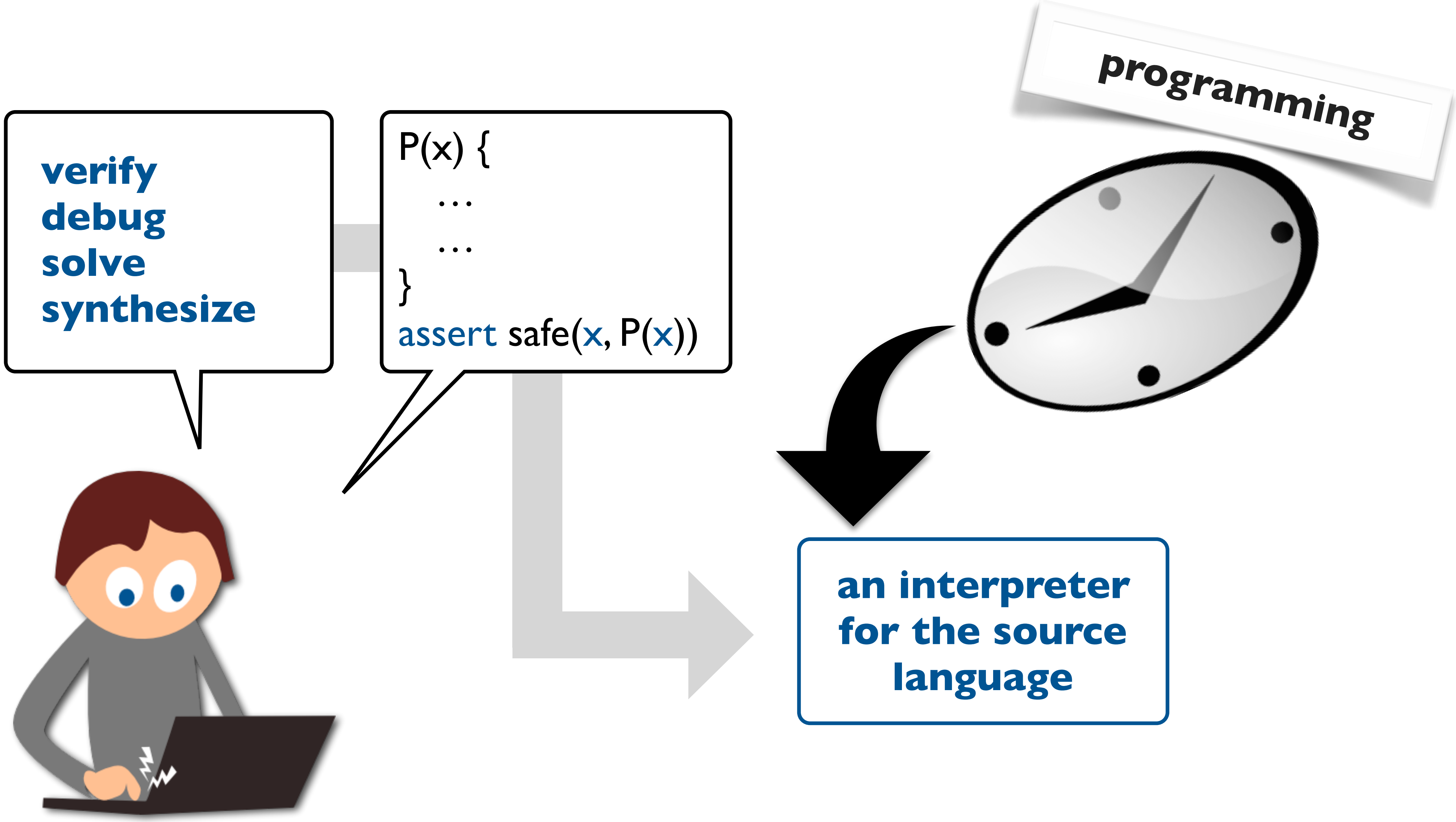
expertise in PL, FM, SE

symbolic compiler

SMT solver

P(x)

Wanted: an easier way to build tools

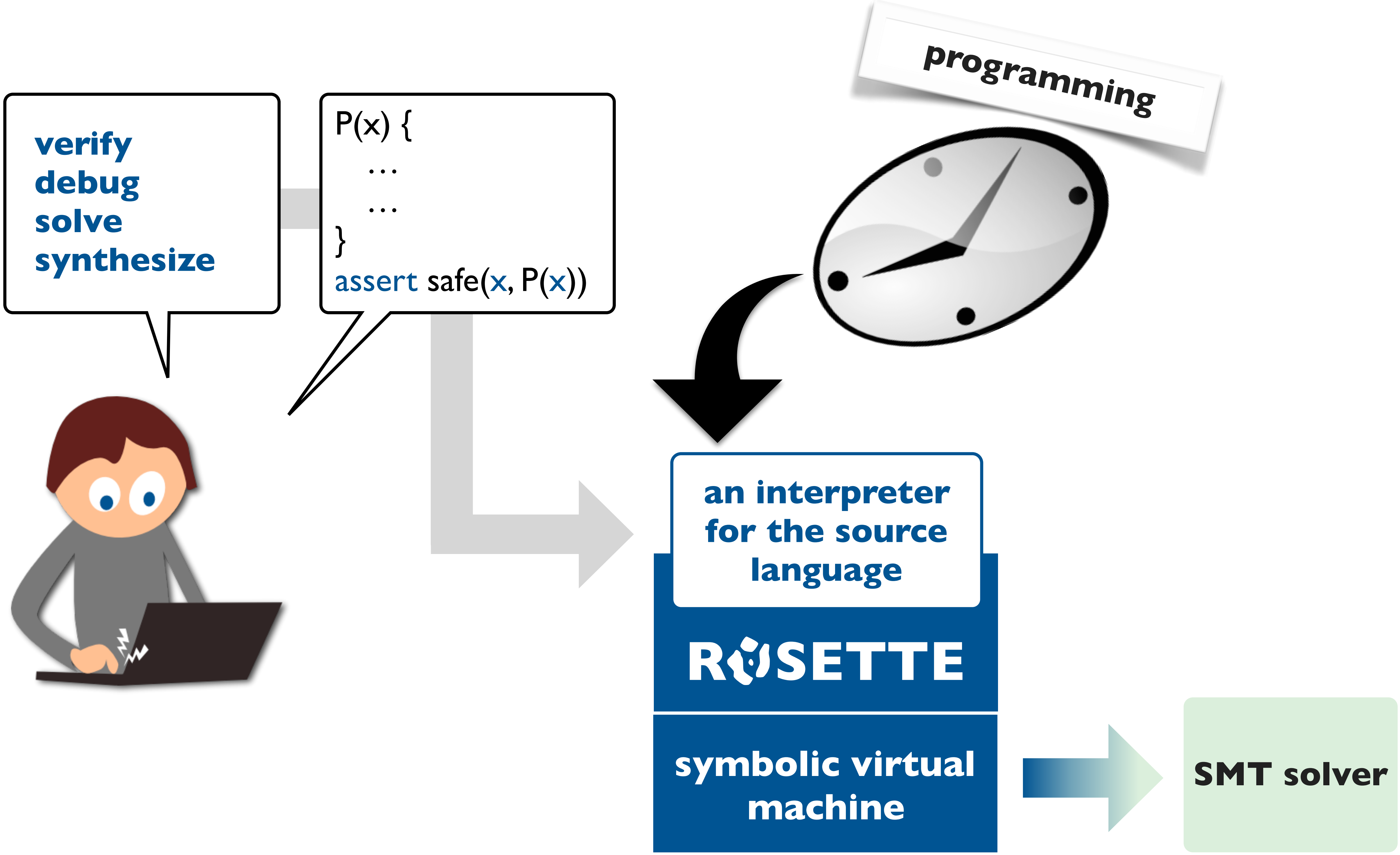


verify
debug
solve
synthesize

```
P(x) {
   …
   …
}
assert safe(x, P(x))
```

an interpreter for the source language

Wanted: an easier way to build tools



verify
debug
solve
synthesize

```
P(x) {
    …
    …
}
assert safe(x, P(x))
```

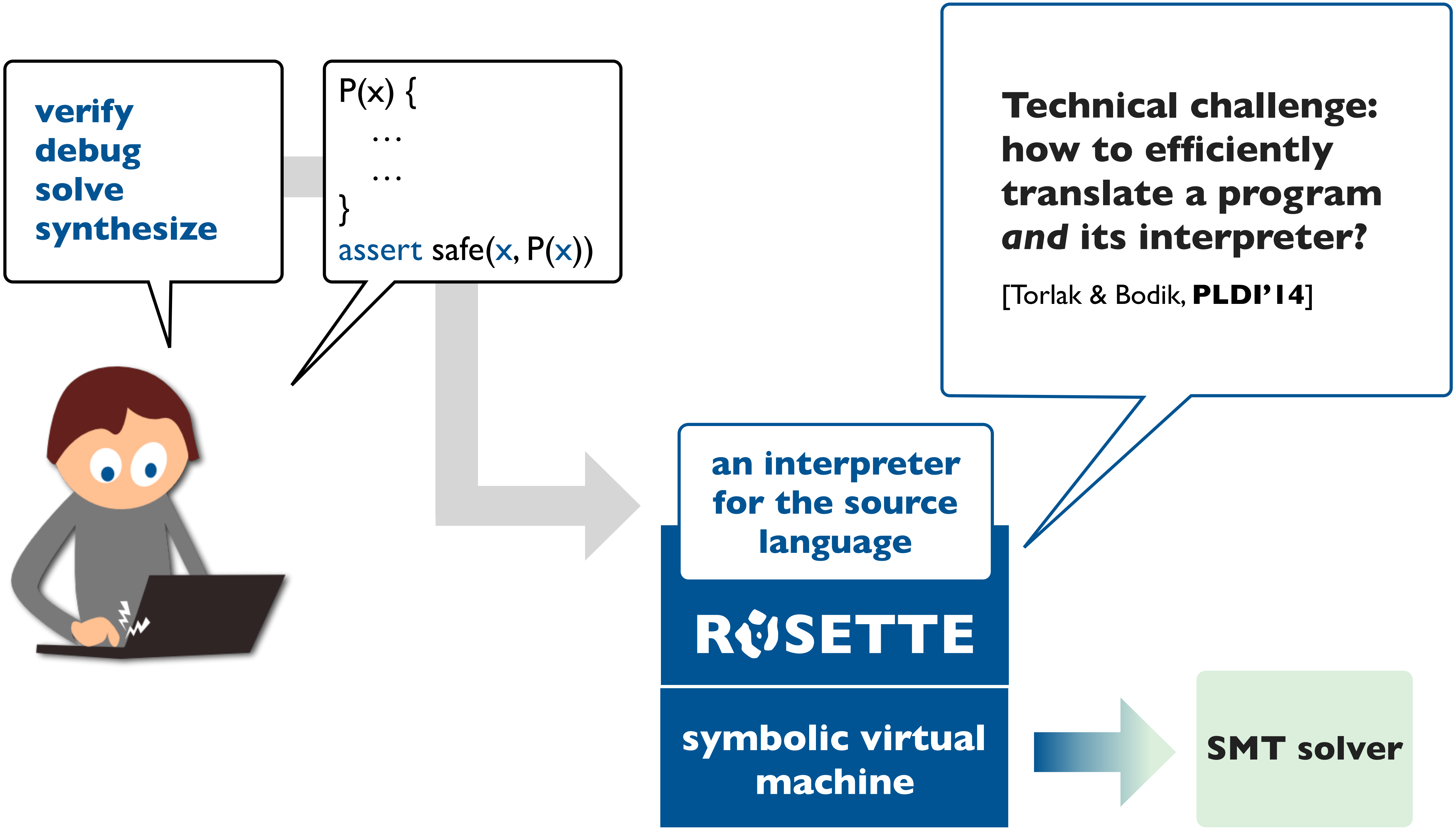
an interpreter for the source language

ROSETTE

symbolic virtual machine

SMT solver

Wanted: an easier way to build tools



verify
debug
solve
synthesize

```
P(x) {
   …
   …
}
assert safe(x, P(x))
```

an interpreter for the source language

ROSETTE

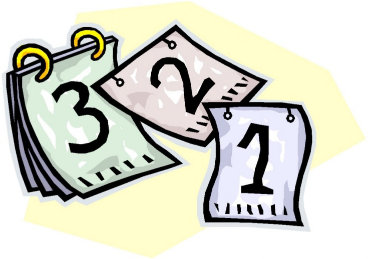
symbolic virtual machine

SMT solver

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

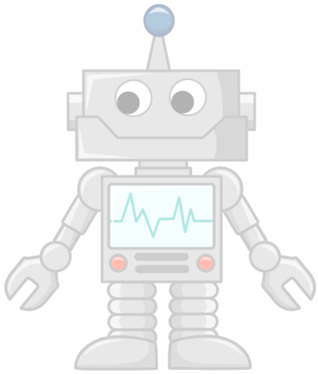


The classic (hard) way to build a tool
What is hard about building a solver-aided tool?


SDSL
SVM
SMT

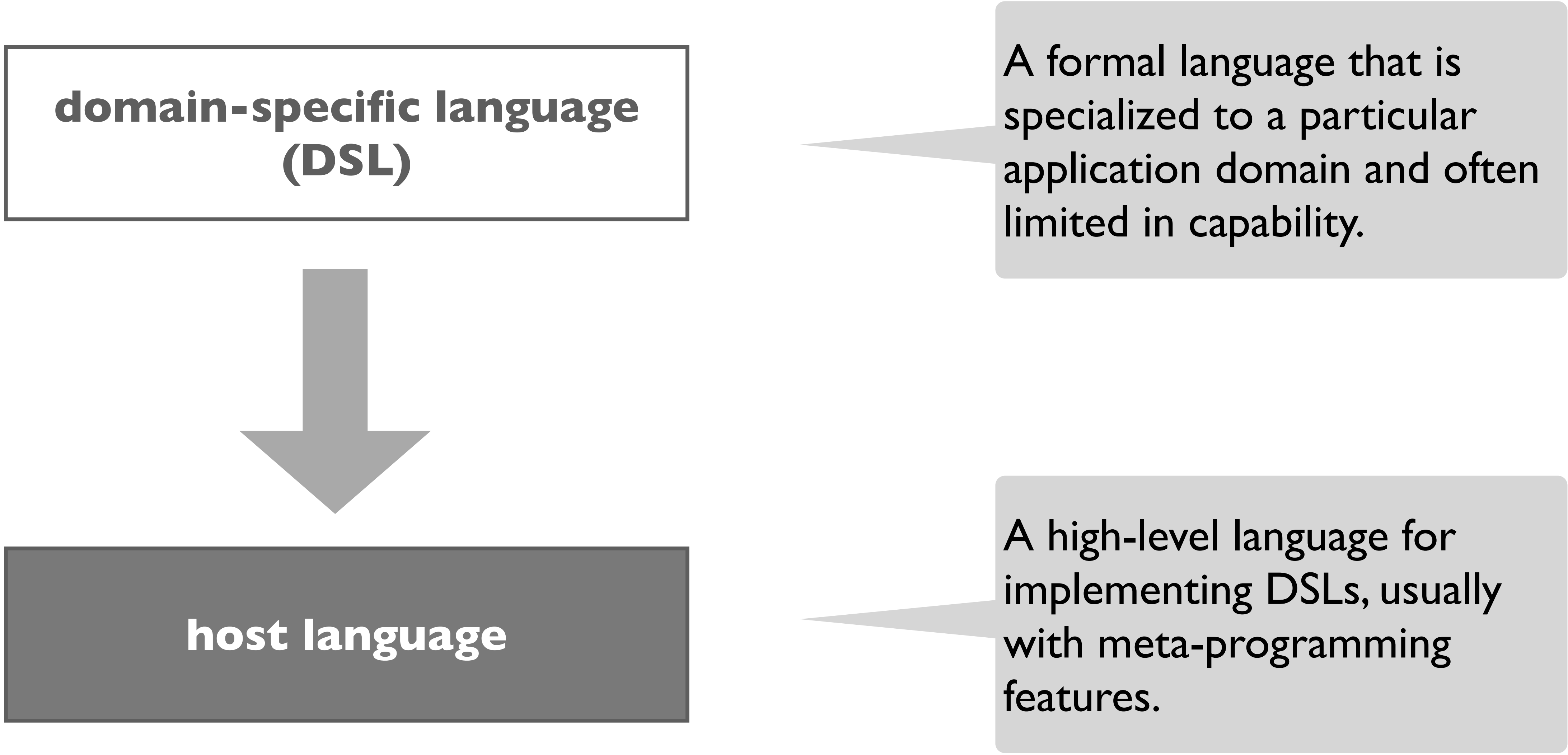
An easier way: tools as languages
How to build tools by stacking layers of languages.

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



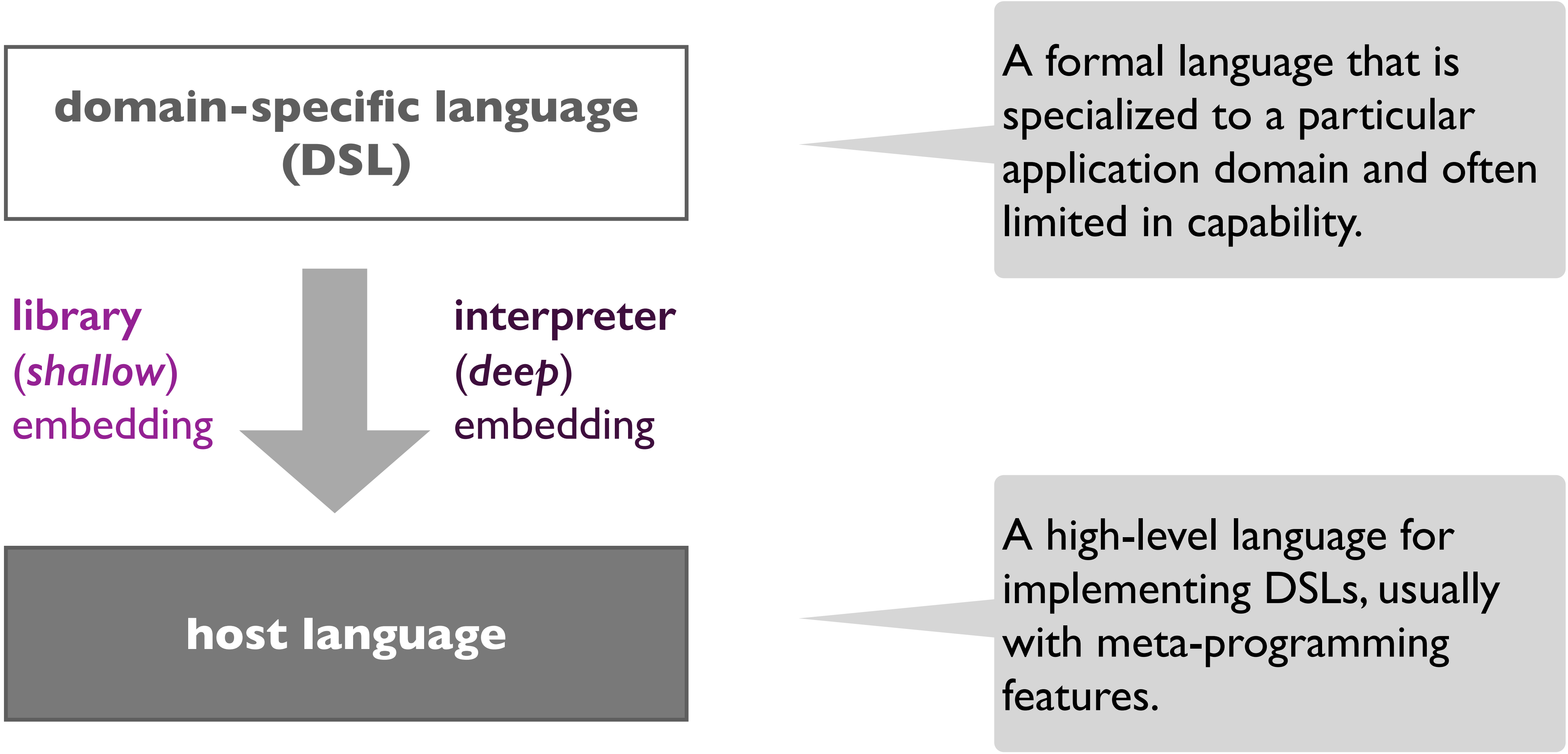
domain-specific language (DSL)

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

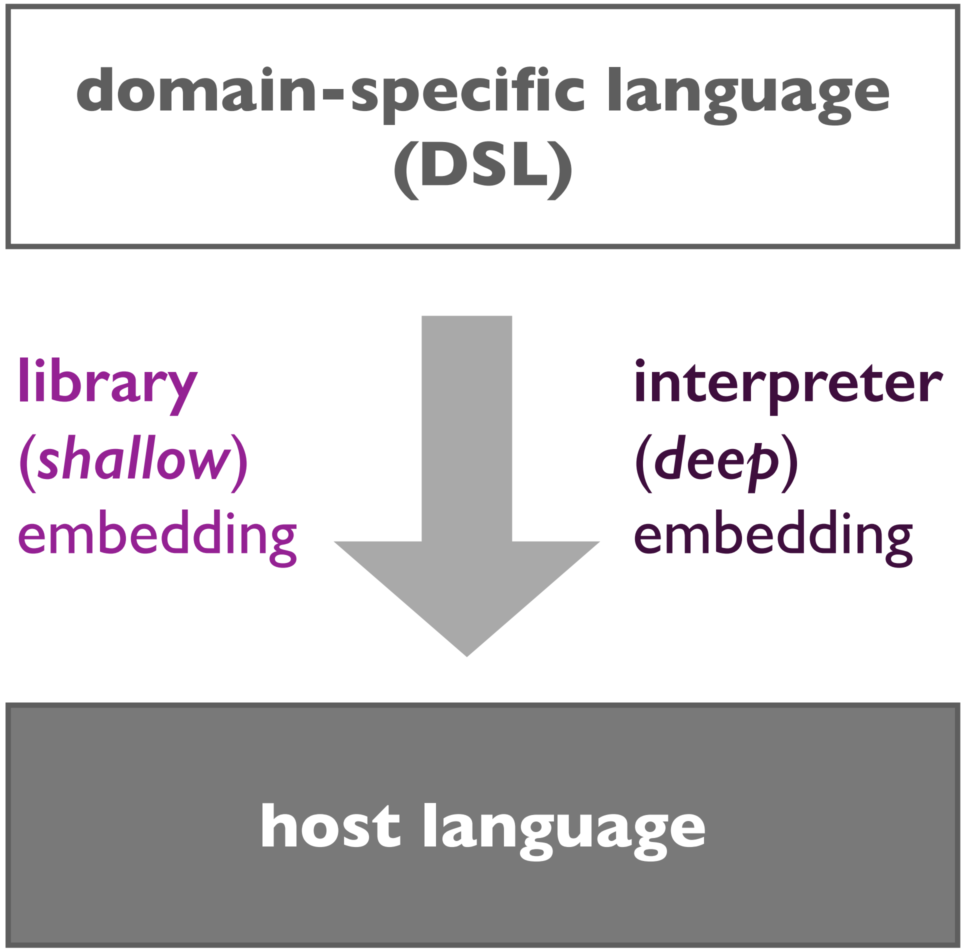
interpreter (*deep*) embedding

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

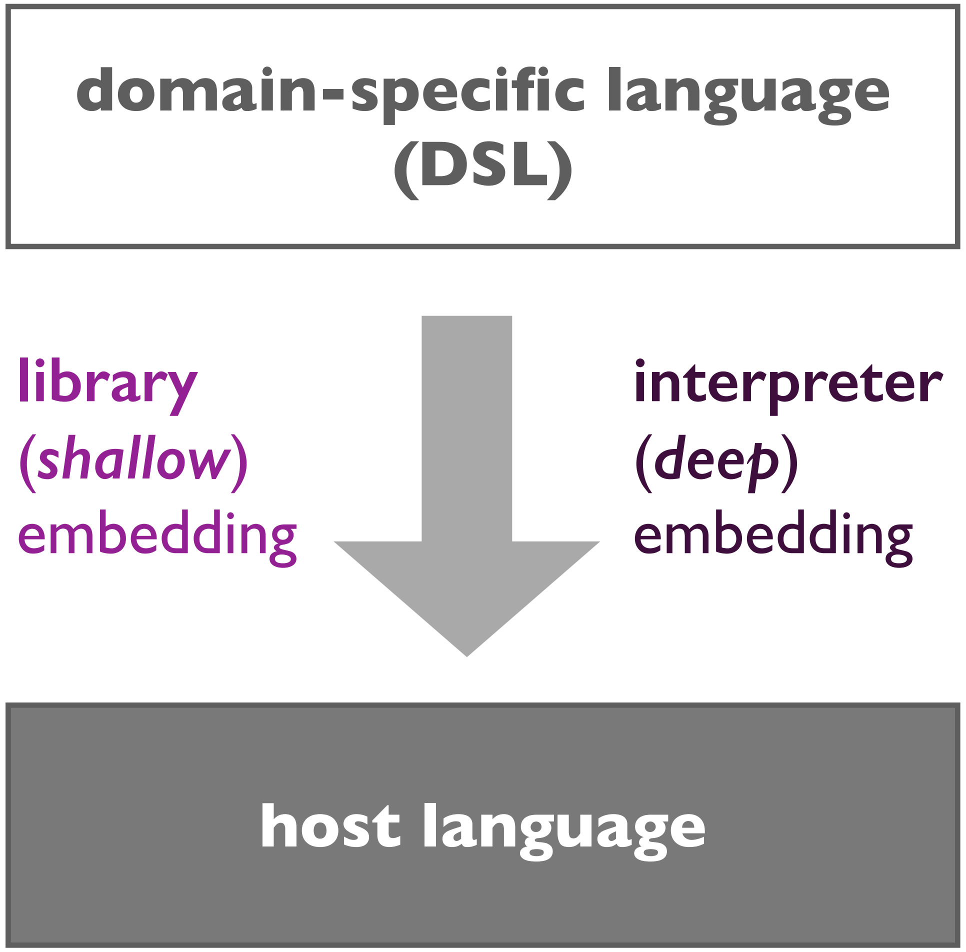
hardware design
Bluespec, Chisel, Verilog, VHDL

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

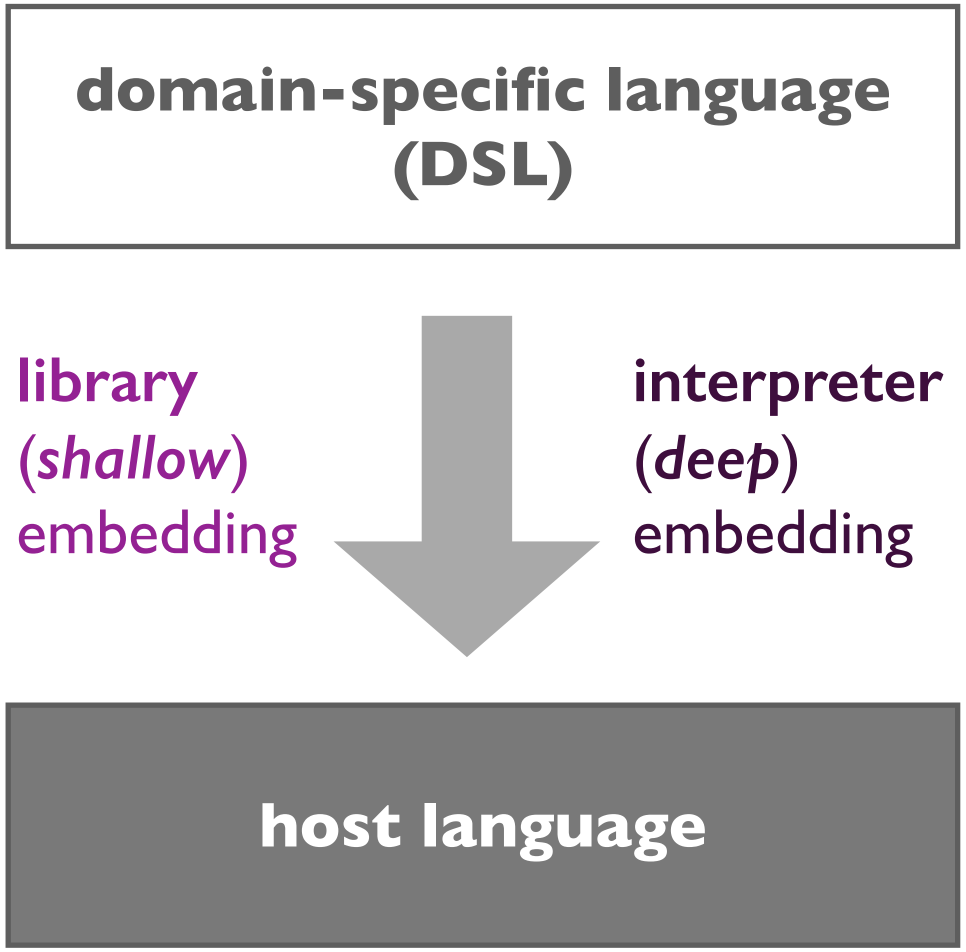
Eigen / Matlab

```
C = A * B
```

C / Java

```
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.

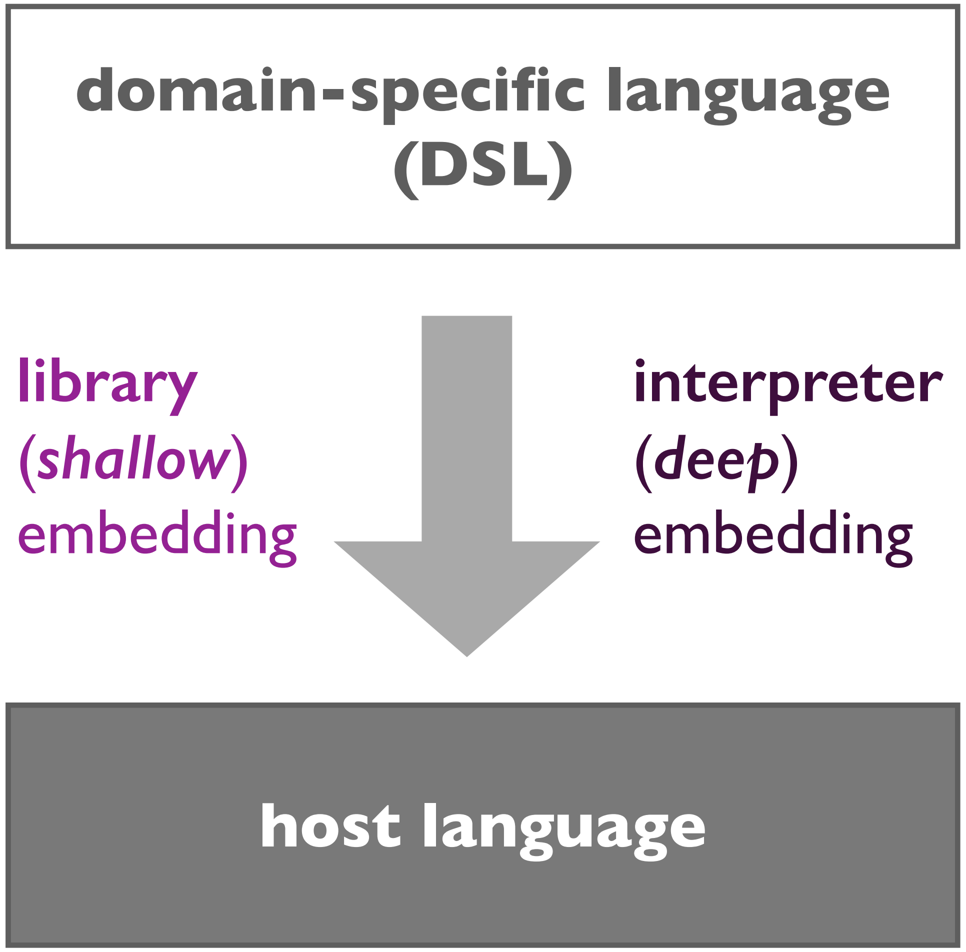
Eigen / Matlab

```
C = A * B
```

C / Java

```
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 * B
```

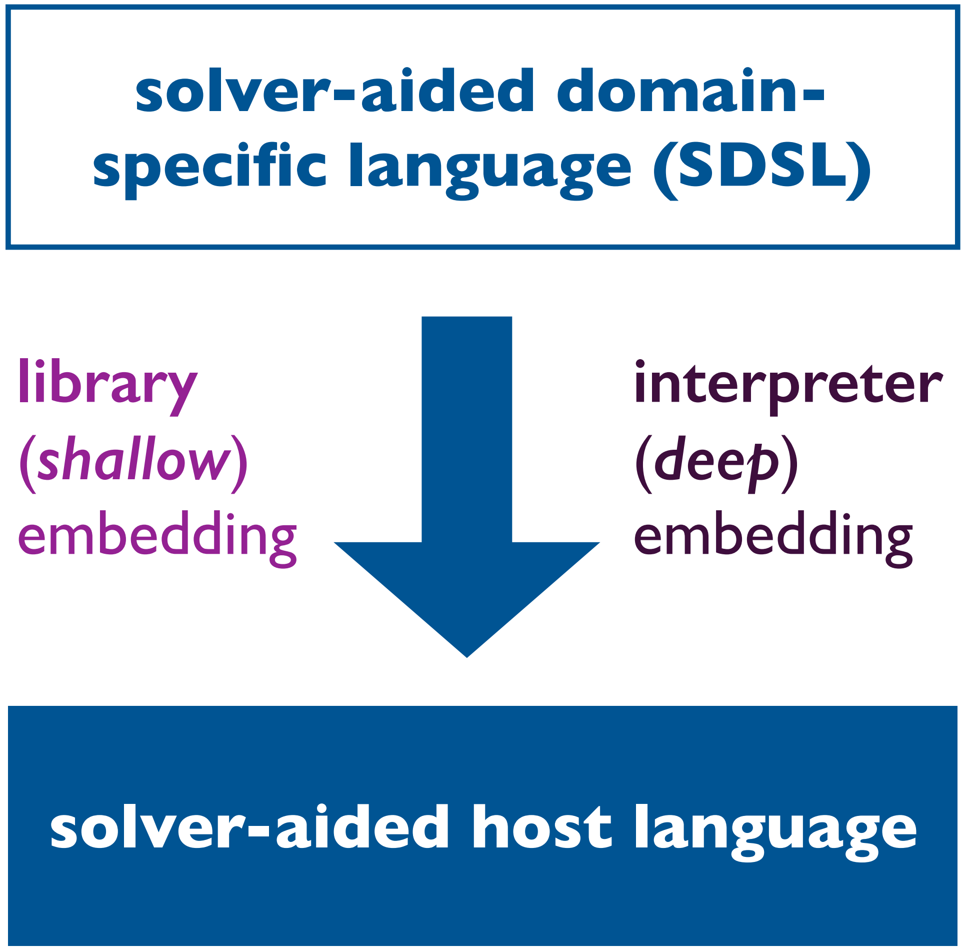
[associativity]

Easier for tools to analyze.

C / Java

```
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 solver-aided languages



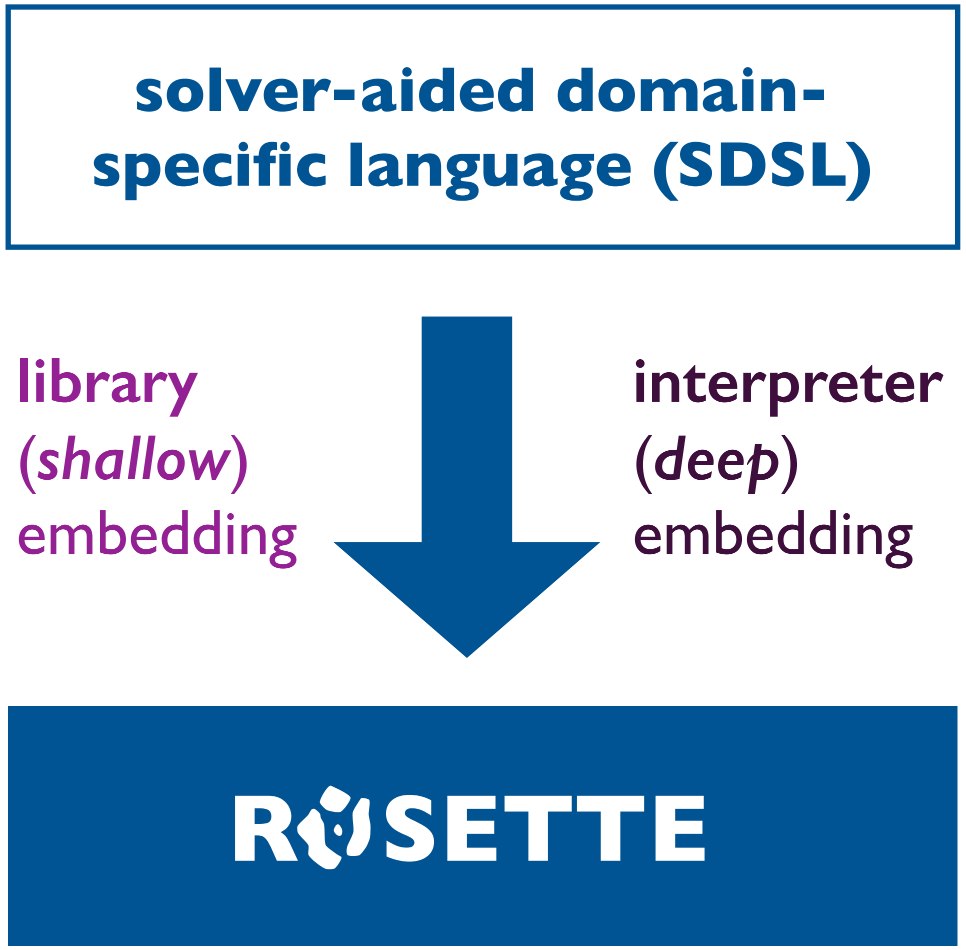
solver-aided domain-specific language (SDSL)

library (*shallow*) embedding

interpreter (*deep*) embedding

solver-aided host language

Layers of solver-aided languages: tools as SDSLs



solver-aided domain-specific language (SDSL)

library (*shallow*) embedding

interpreter (*deep*) embedding

ROSETTE

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

synthesis-aided compilation
Chlorophyll (PLDI'14), GreenThumb (ASPLOS'16)

type system soundness
Bonsai (POPL'18)

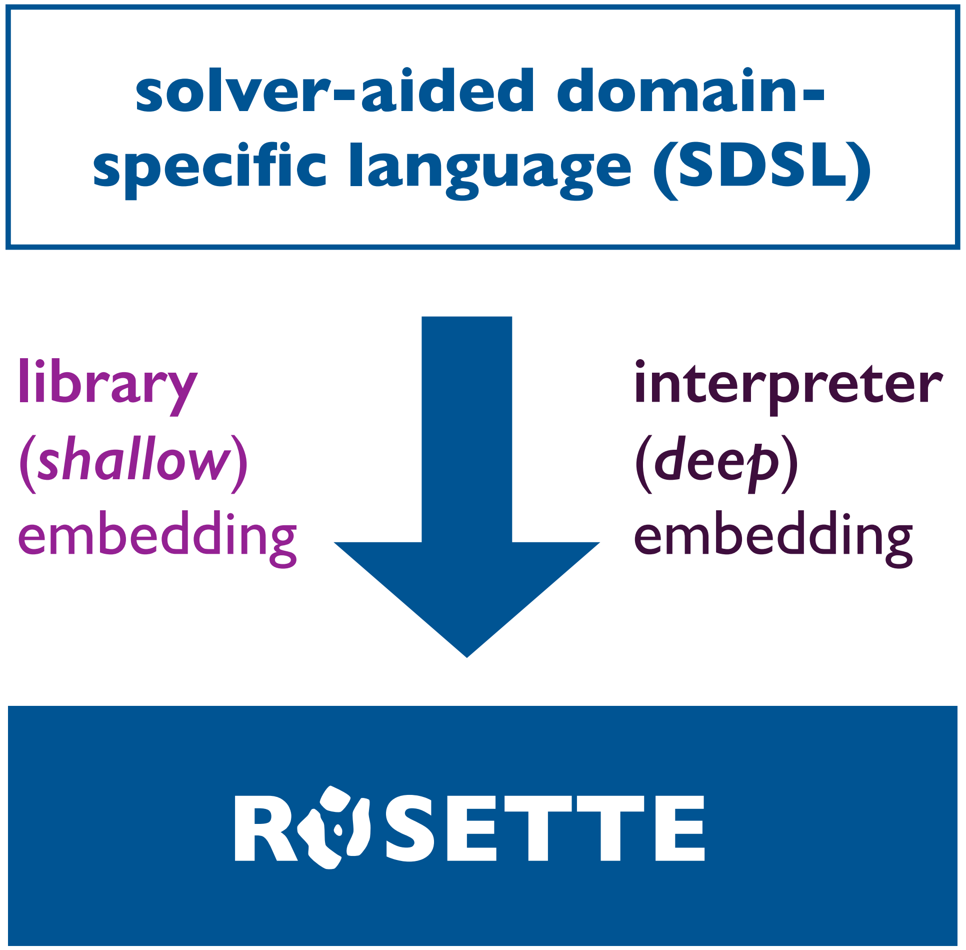
systems software
Serval (SOSP'19)

databases
Cosette (CIDR'17)

radiation therapy control
Neutrons (CAV'16)

… and more

Layers of solver-aided languages: tools as SDSLs



solver-aided domain-specific language (SDSL)

library (*shallow*) embedding

interpreter (*deep*) embedding

ROSETTE

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

synthesis-aided compilation
Chlorophyll (PLDI'14), GreenThumb (ASPLOS'16)

type system soundness
Bonsai (POPL'18)

systems software
Serval (SOSP'19)

databases
Cosette (CIDR'17)

radiation therapy control
Neutrons (CAV'16)

… and more

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
```

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

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
```

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

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

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
```

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

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]

A tiny example SDSL

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)
```

A tiny example SDSL

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)))
```

A tiny example SDSL

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 SDSL

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 (eval opcode))
       (define args (map load in))
       (store out (apply op args))]))
  (load (last)))
```

A tiny example SDSL

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)
```

```
(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	

interpret

```
(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

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)))
```

0	-2
1	-1
2	
3	
4	
5	
6	

```
(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

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)
```

```
(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	

interpret

```
(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

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)))
```

0	-2
1	-1
2	
3	
4	
5	
6	

```
(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

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)
```

```
(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	0
3	
4	
5	
6	

interpret

```
(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

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)
```

```
(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	0
3	0
4	-2
5	0
6	-1

interpret

```
(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)
-1
```

interpret

ROSETTE

```
(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	0
3	0
4	-2
5	0
6	-1

```
(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)
-1
```

ROSETTE

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

- pattern matching
- dynamic evaluation
- 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 (eval opcode))
       (define args (map load in))
       (store out (apply op args))]))
  (load (last)))
```

A tiny example SDSL

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

A tiny example SDSL

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

A tiny example SDSL

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

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

A tiny example SDSL

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)
                  (interpret 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 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]
```

query

ROSETTE

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

A tiny example SDSL

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]

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

query

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

A tiny example SDSL

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

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

query

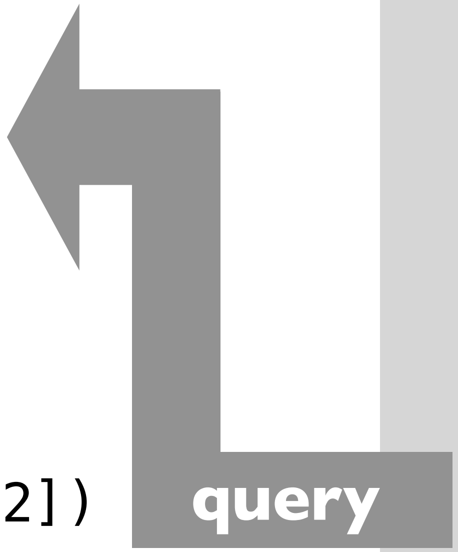
```
(define in (list (int32 0) (int32 -2)))
(debug [register?]
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

A tiny example SDSL

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
```

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



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

A tiny example SDSL

ROSETTE

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

> synthesize(bvmax, max)
```

query

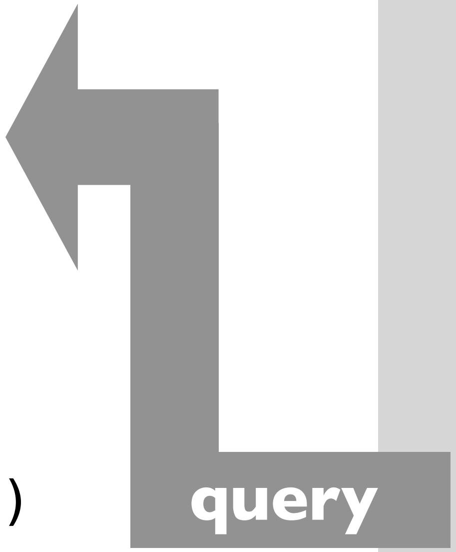
```
(define-symbolic x y int32?)
(define in (list x y))
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in)
                  (interpret max in)))))
```

A tiny example SDSL

ROSETTE

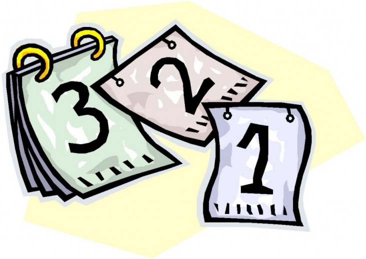
```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r1)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> synthesize(bvmax, max)
```



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

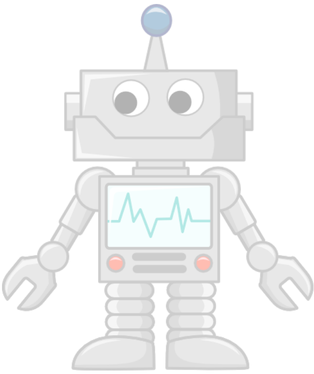
How to build your own solver-aided tool or language



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

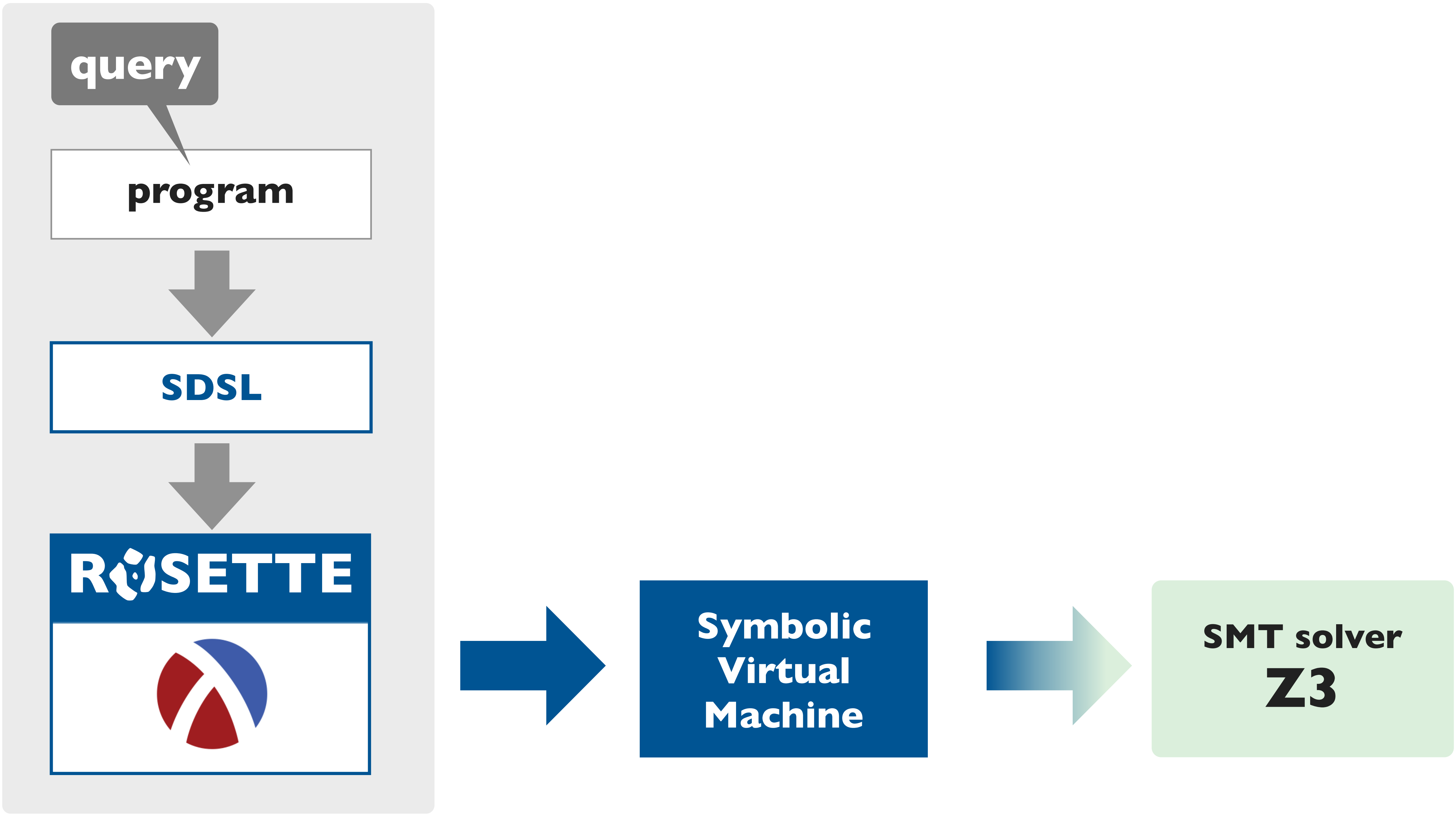
An easier way: tools as languages
How to build tools by stacking layers of languages.

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

program

SDSL

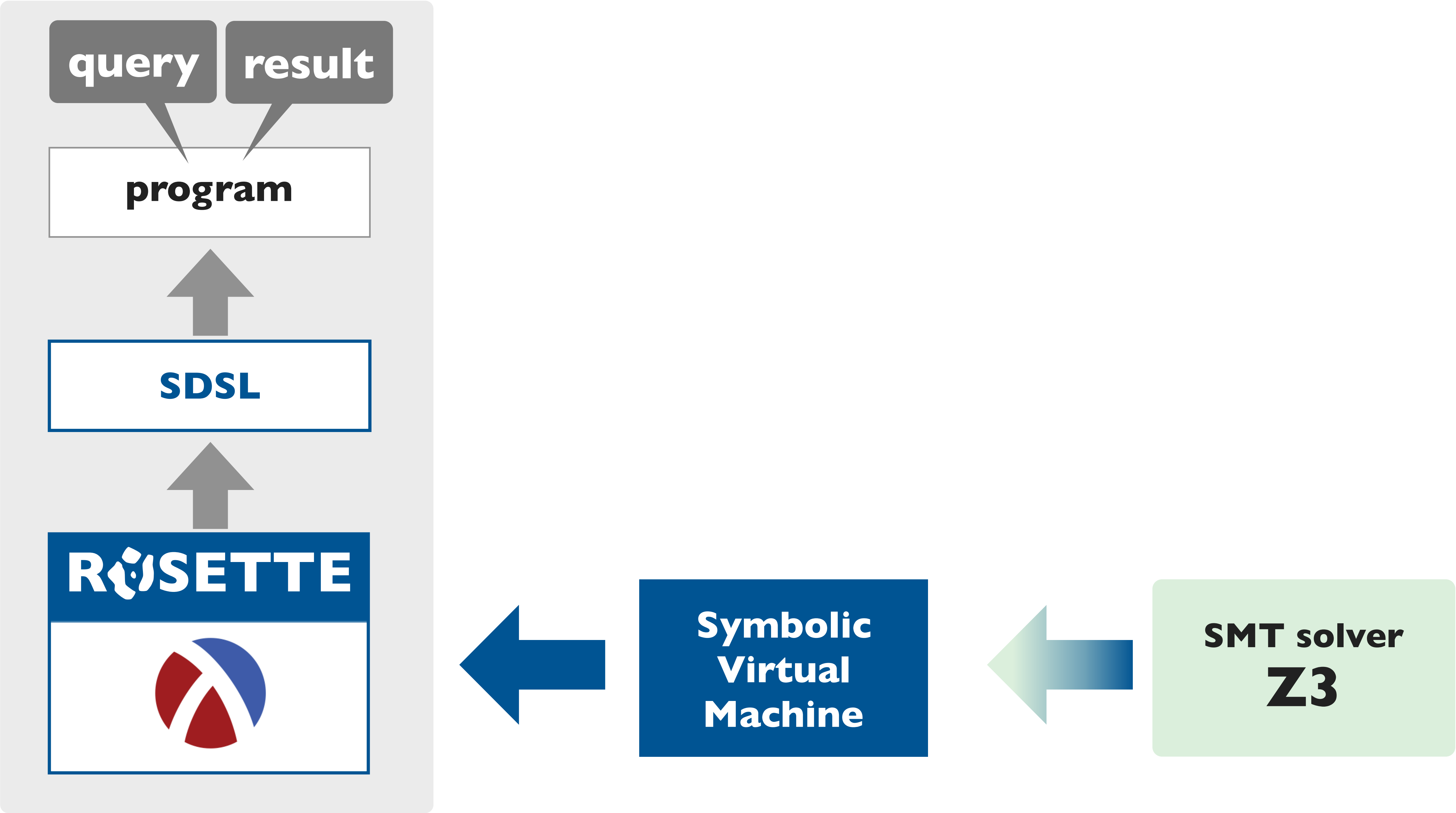
ROSETTE

Symbolic Virtual Machine

SMT solver

Z3

How it all works: a big picture view



query

result

program

SDSL

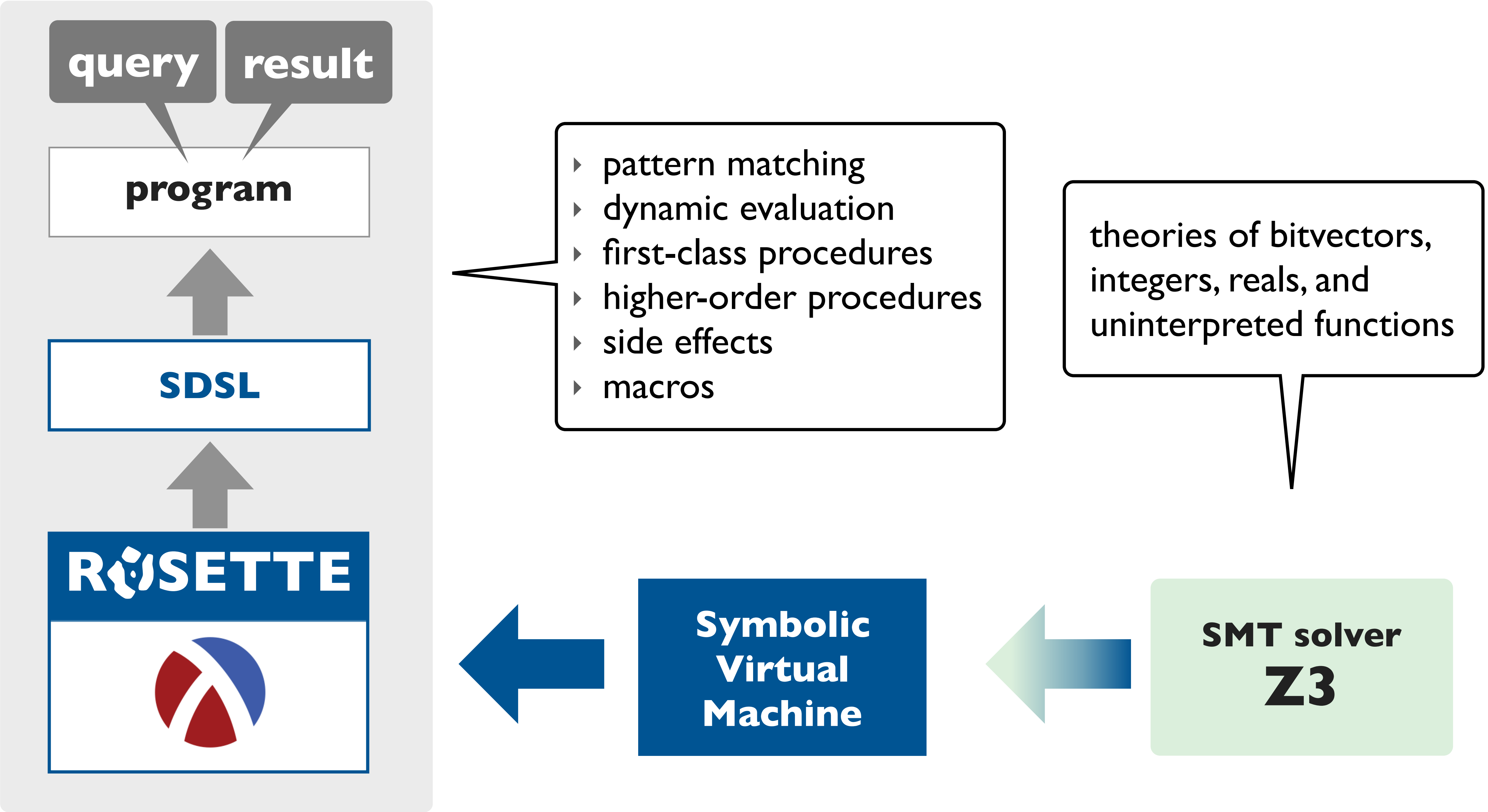
ROSETTE

Symbolic Virtual Machine

SMT solver

Z3

How it all works: a big picture view



query

result

program

SDSL

ROSETTE

- pattern matching
- dynamic evaluation
- first-class procedures
- higher-order procedures
- side effects
- macros

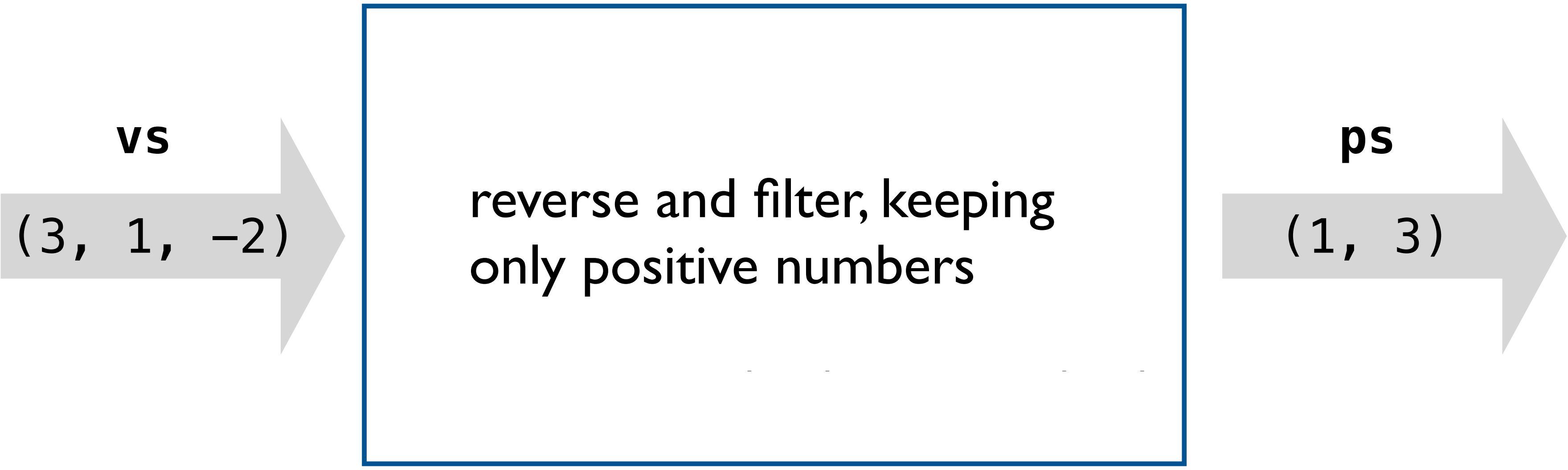
Symbolic Virtual Machine

theories of bitvectors, integers, reals, and uninterpreted functions

SMT solver

Z3

Translation to constraints by example



`vs`

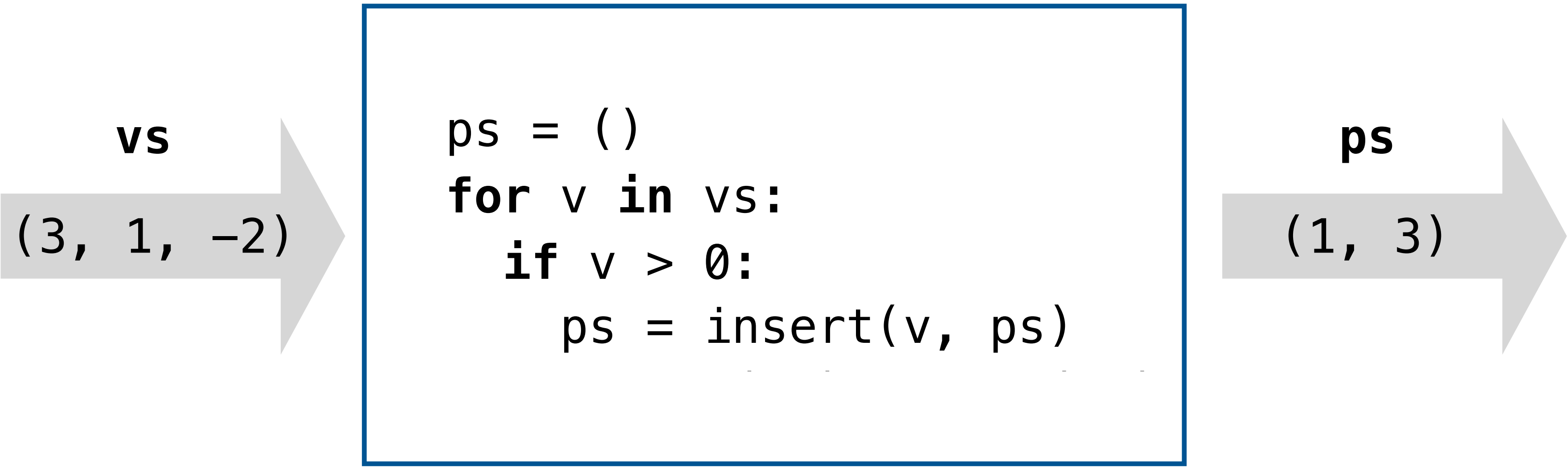
`(3, 1, −2)`

reverse and filter, keeping only positive numbers

`ps`

`(1, 3)`

Translation to constraints by example



vs

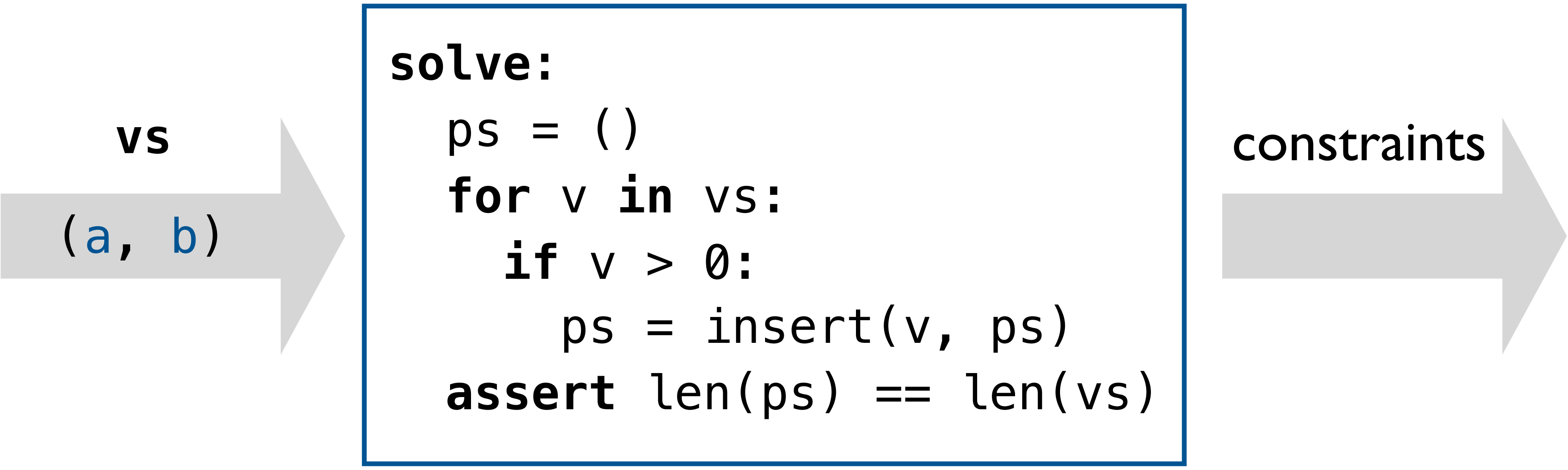
`(3, 1, -2)`

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

ps

`(1, 3)`

Translation to constraints by example



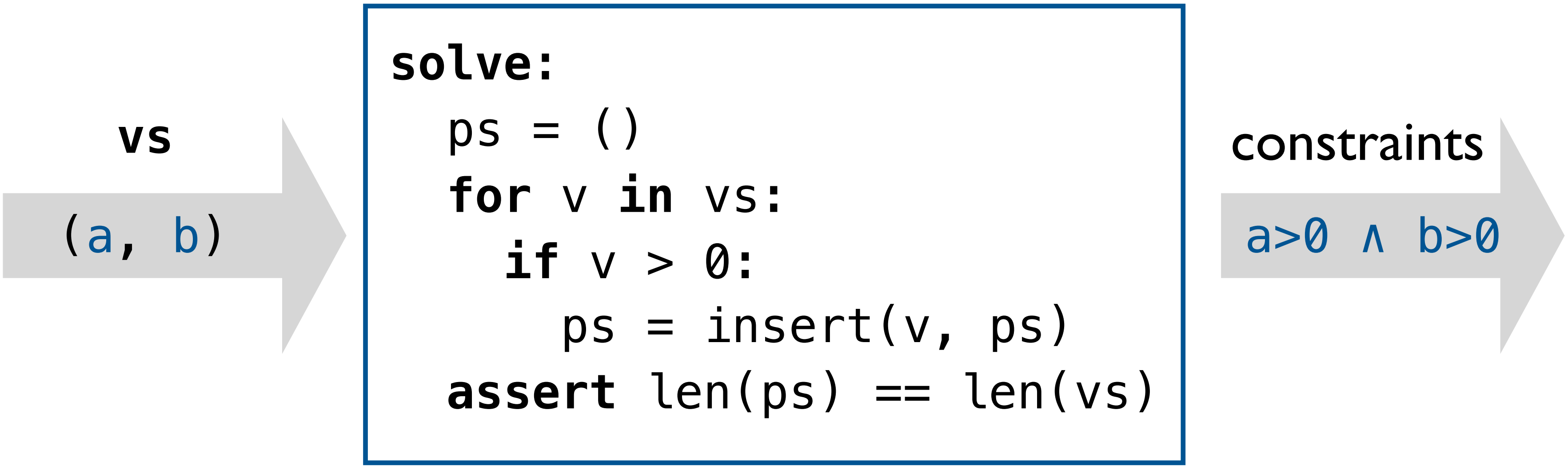
`vs`

`(a, b)`

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

constraints

Translation to constraints by example



vs

`(a, b)`

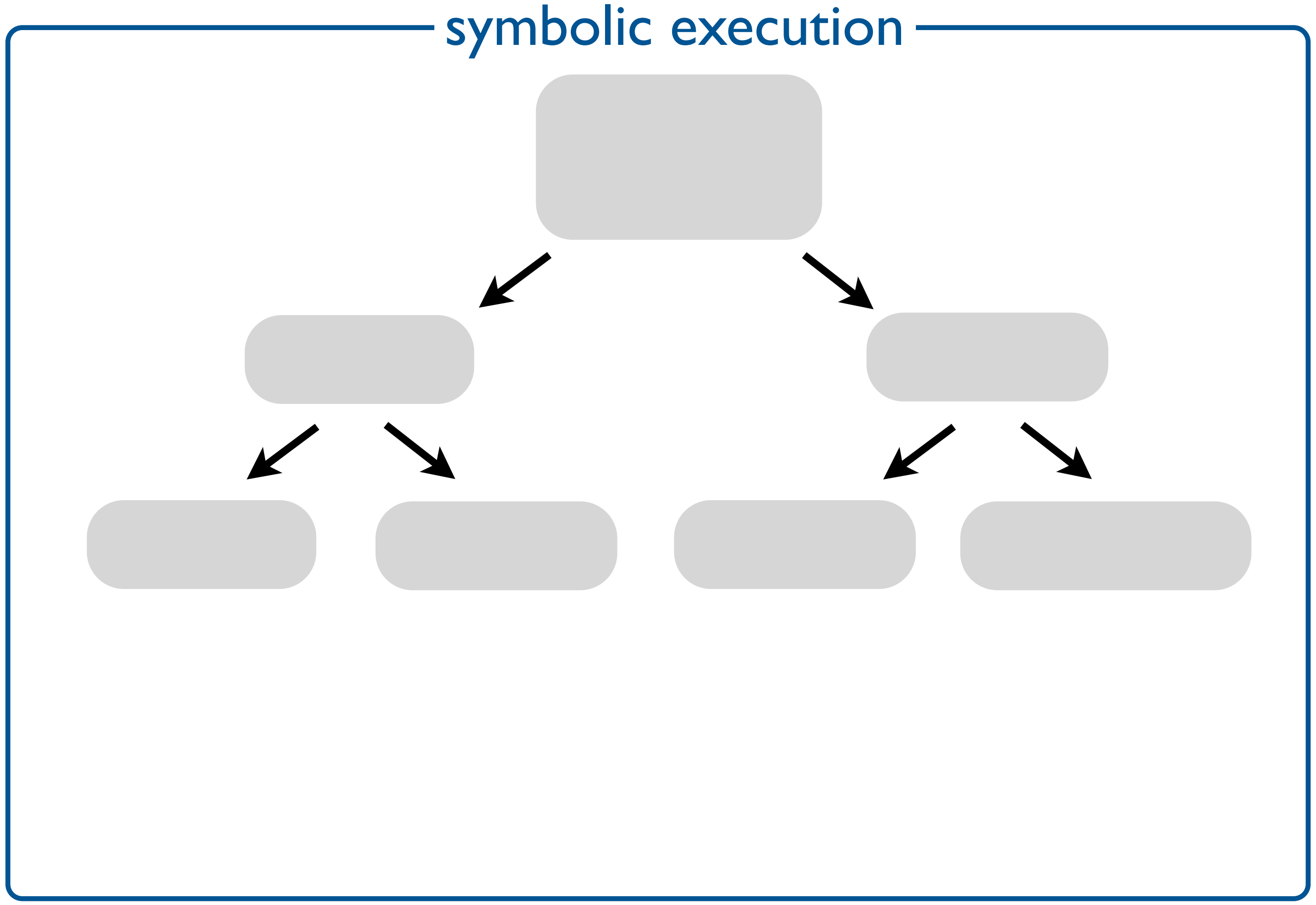
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

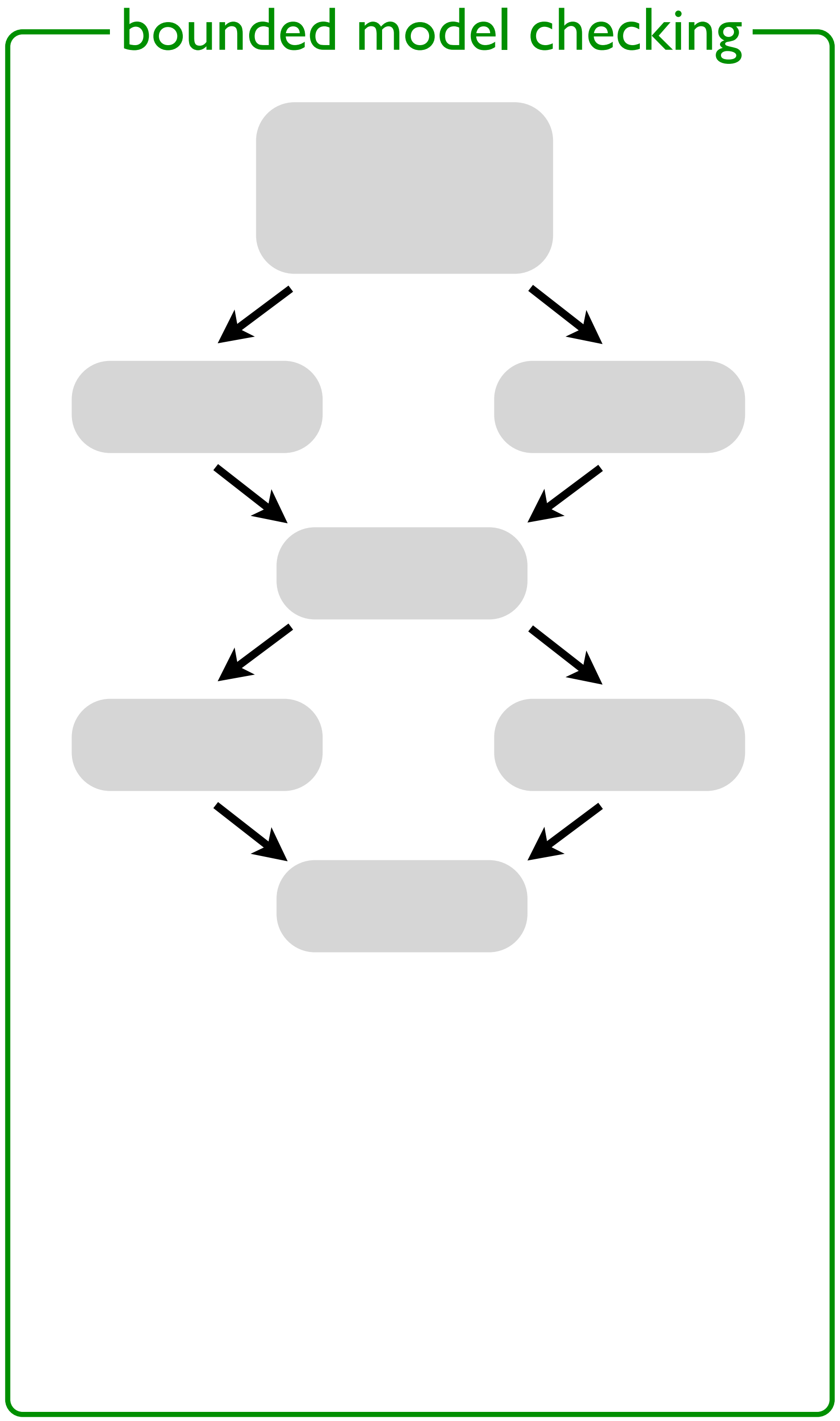
constraints

`a>0 ∧ b>0`

Design space of symbolic encodings: SE and BMC

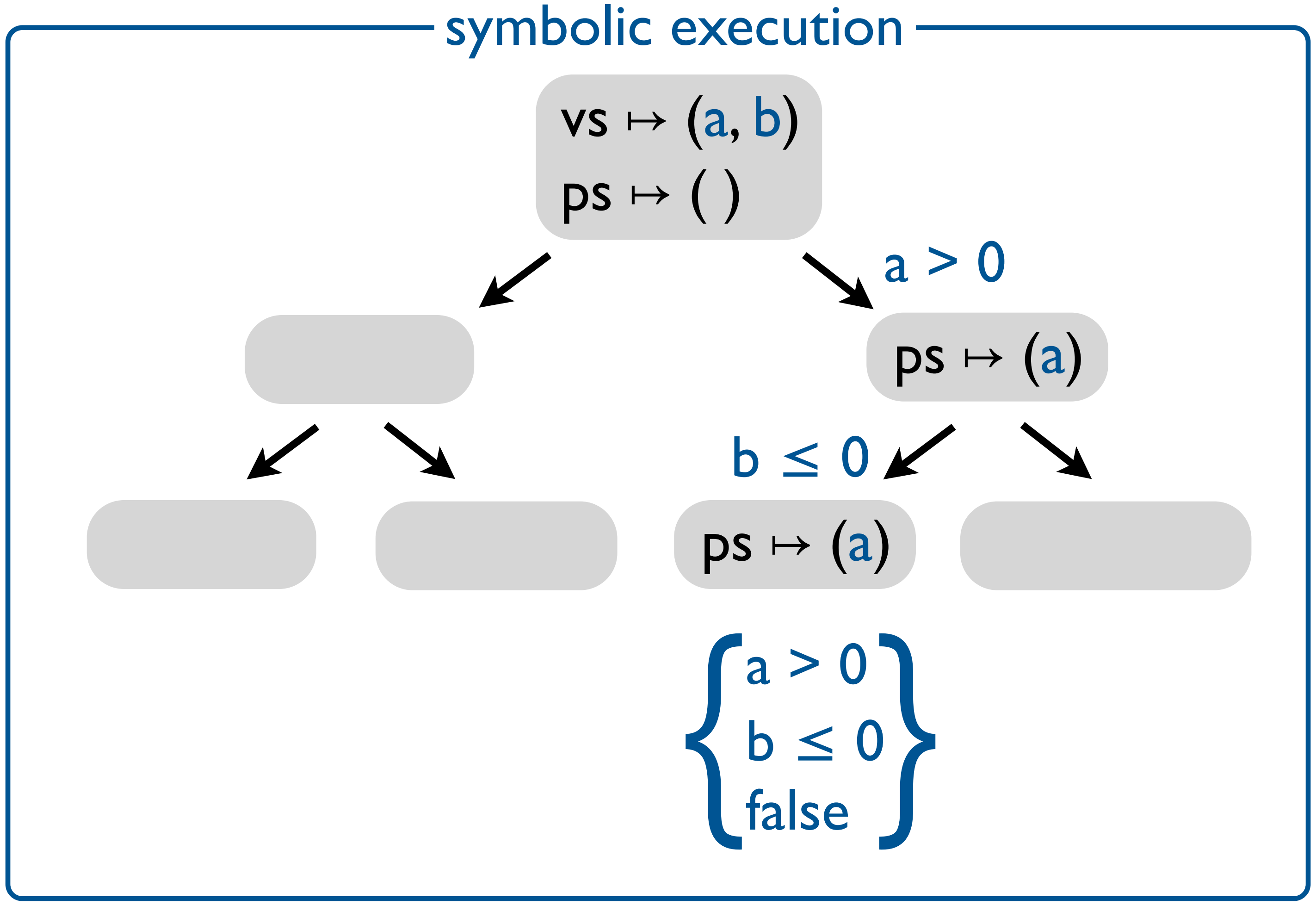
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```


symbolic execution


bounded model checking

Design space of symbolic encodings: SE and BMC

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



symbolic execution

vs ↦ (a, b)
ps ↦ ( )

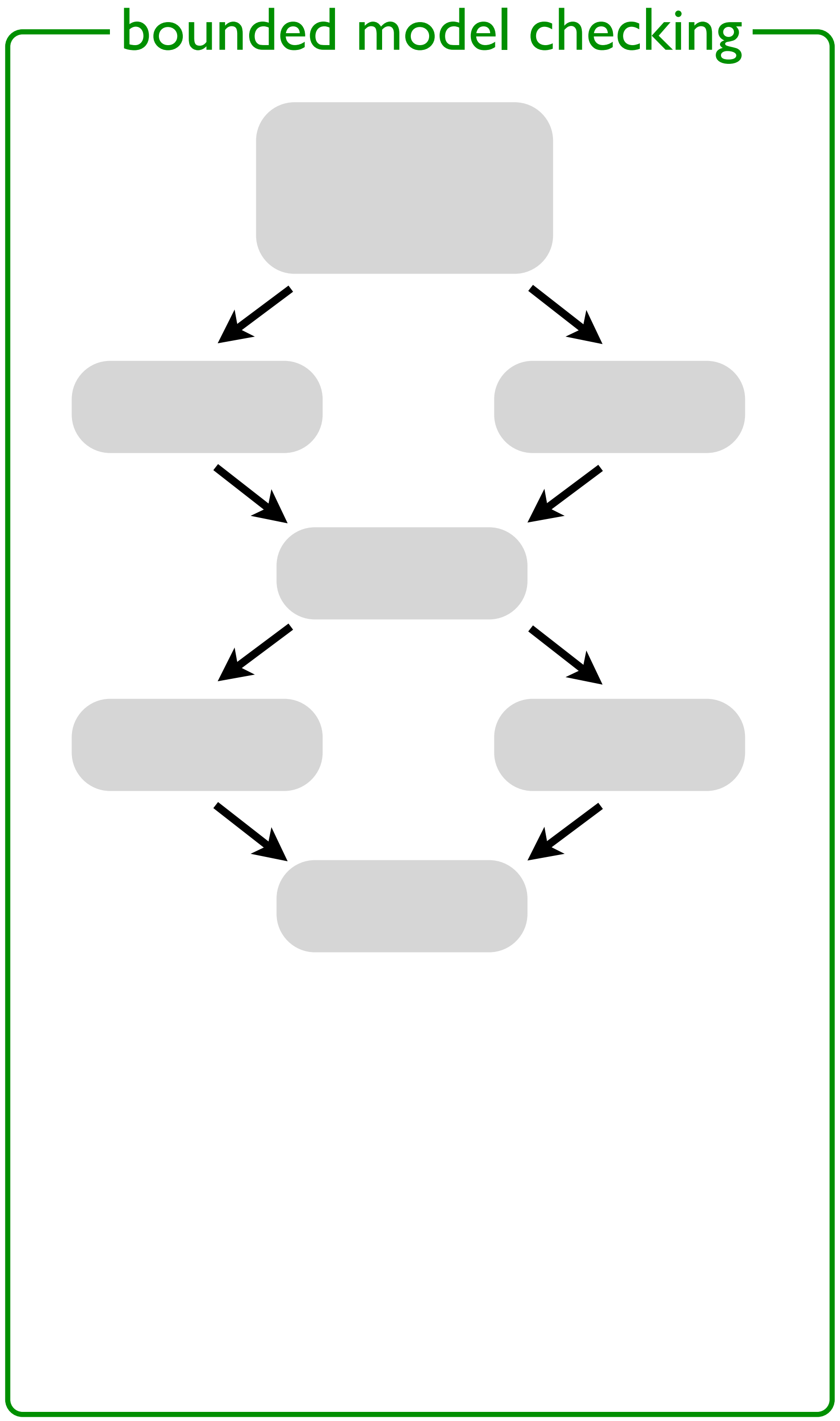
$a > 0$

ps ↦ (a)

$b \le 0$

ps ↦ (a)

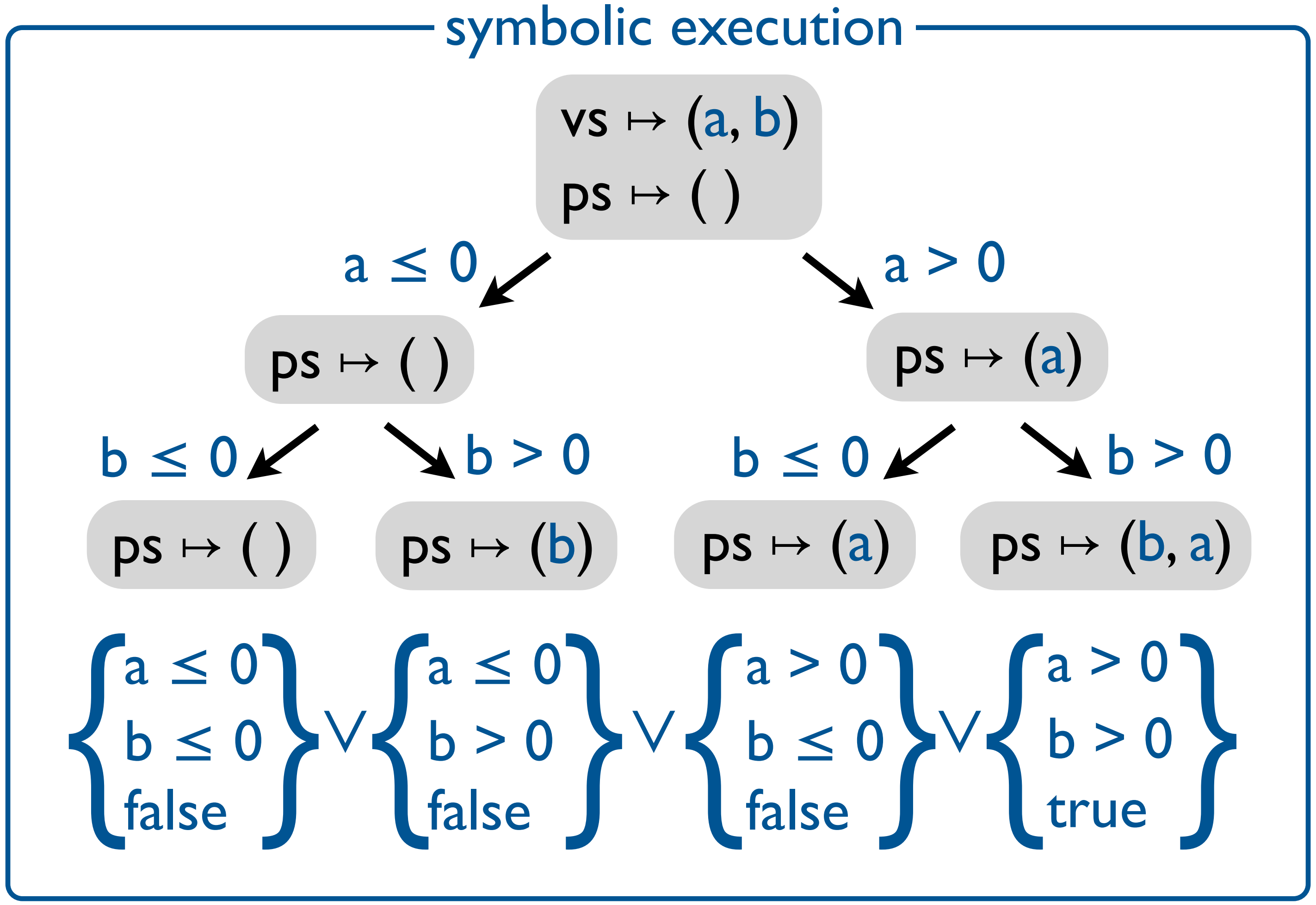
$$\left\{\begin{array}{l} a > 0 \\ b \le 0 \\ \text{false} \end{array}\right\}$$



bounded model checking

Design space of symbolic encodings: SE and BMC

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



symbolic execution

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

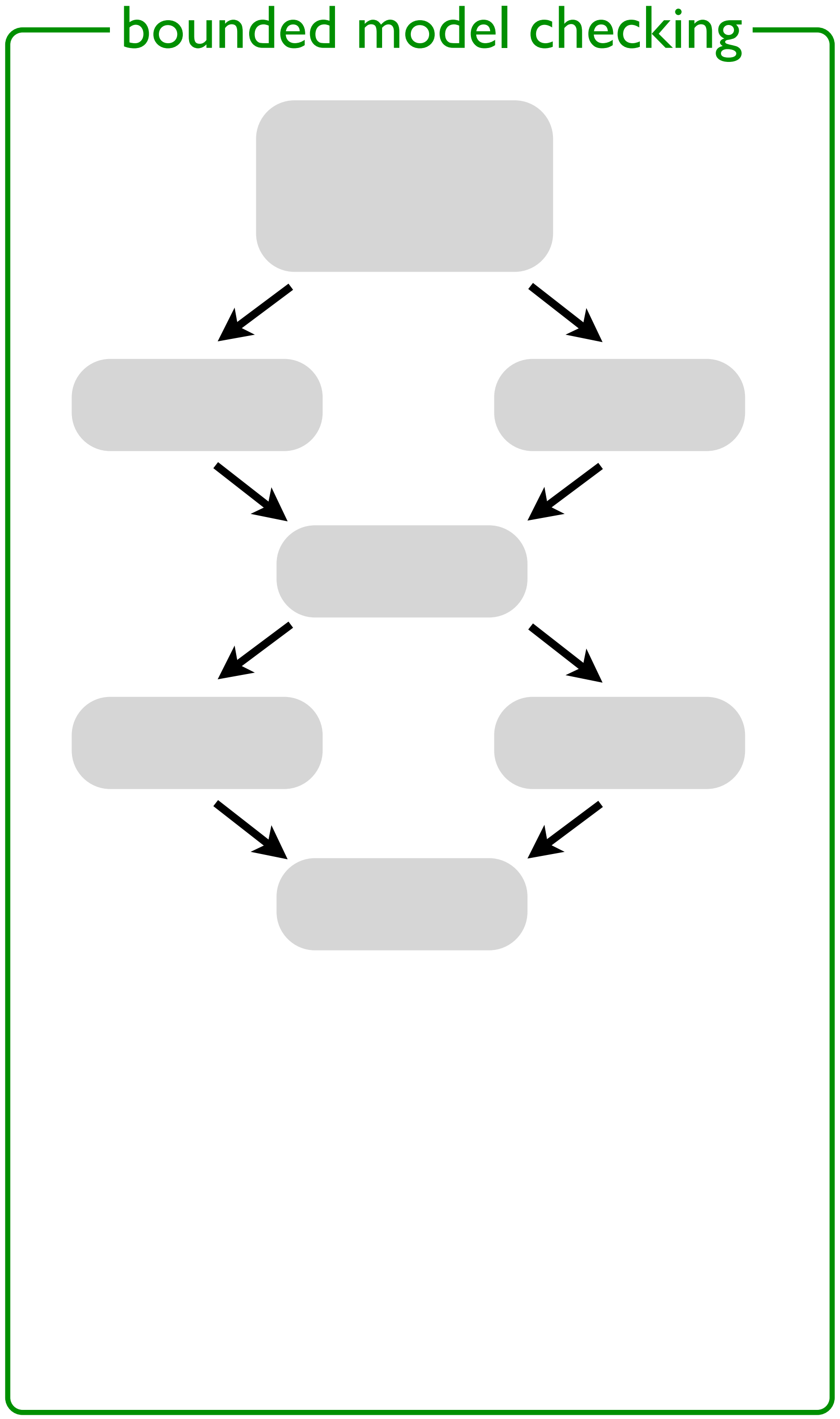
$a \leq 0$ | $a > 0$

$ps \mapsto (\,)$ | $ps \mapsto (a)$

$b \leq 0$ | $b > 0$ | $b \leq 0$ | $b > 0$

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

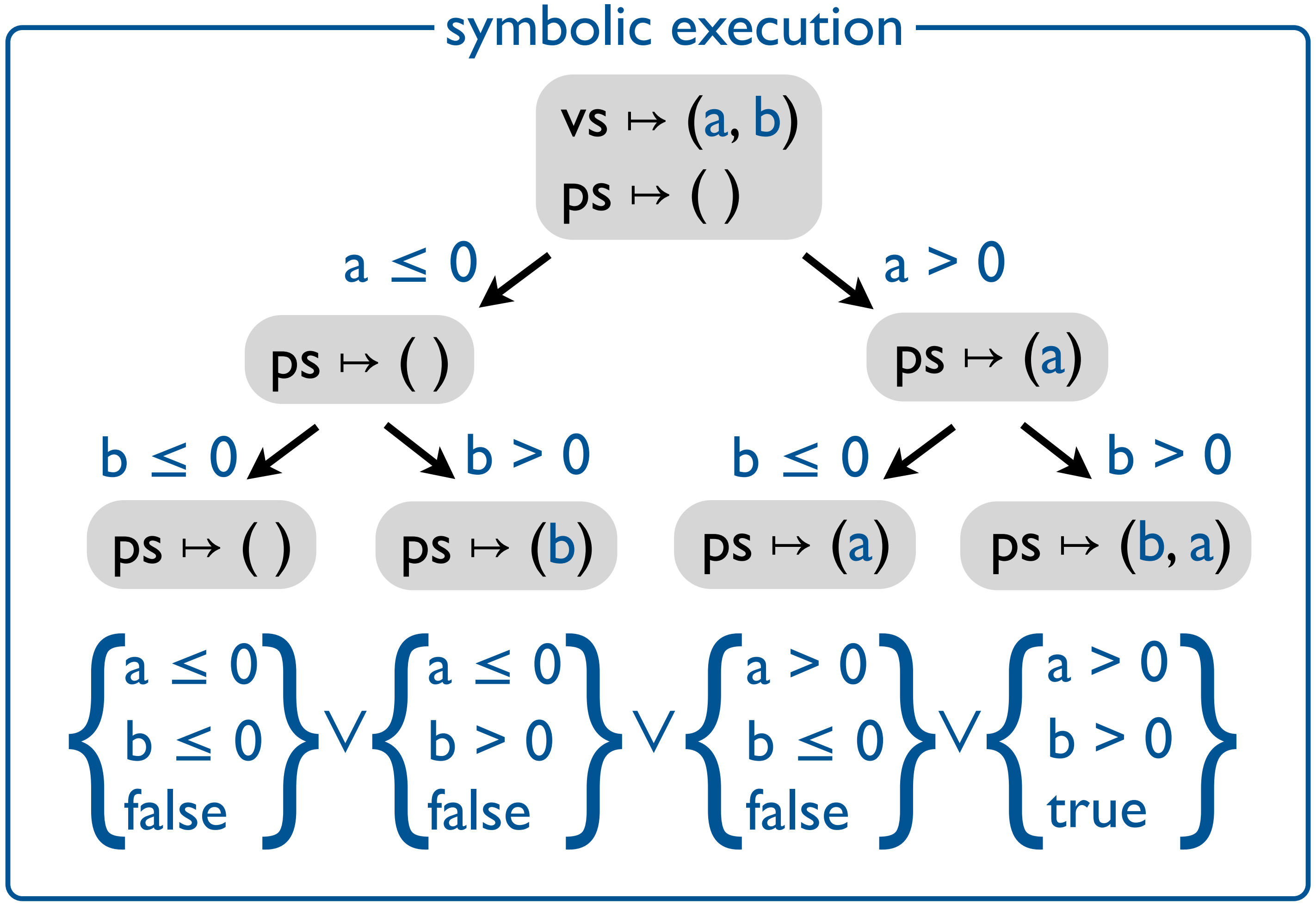
$$\left\{\begin{matrix} a \leq 0 \\ b \leq 0 \\ \text{false} \end{matrix}\right\} \vee \left\{\begin{matrix} a \leq 0 \\ b > 0 \\ \text{false} \end{matrix}\right\} \vee \left\{\begin{matrix} a > 0 \\ b \leq 0 \\ \text{false} \end{matrix}\right\} \vee \left\{\begin{matrix} a > 0 \\ b > 0 \\ \text{true} \end{matrix}\right\}$$



bounded model checking

Design space of symbolic encodings: SE and BMC

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



symbolic execution

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

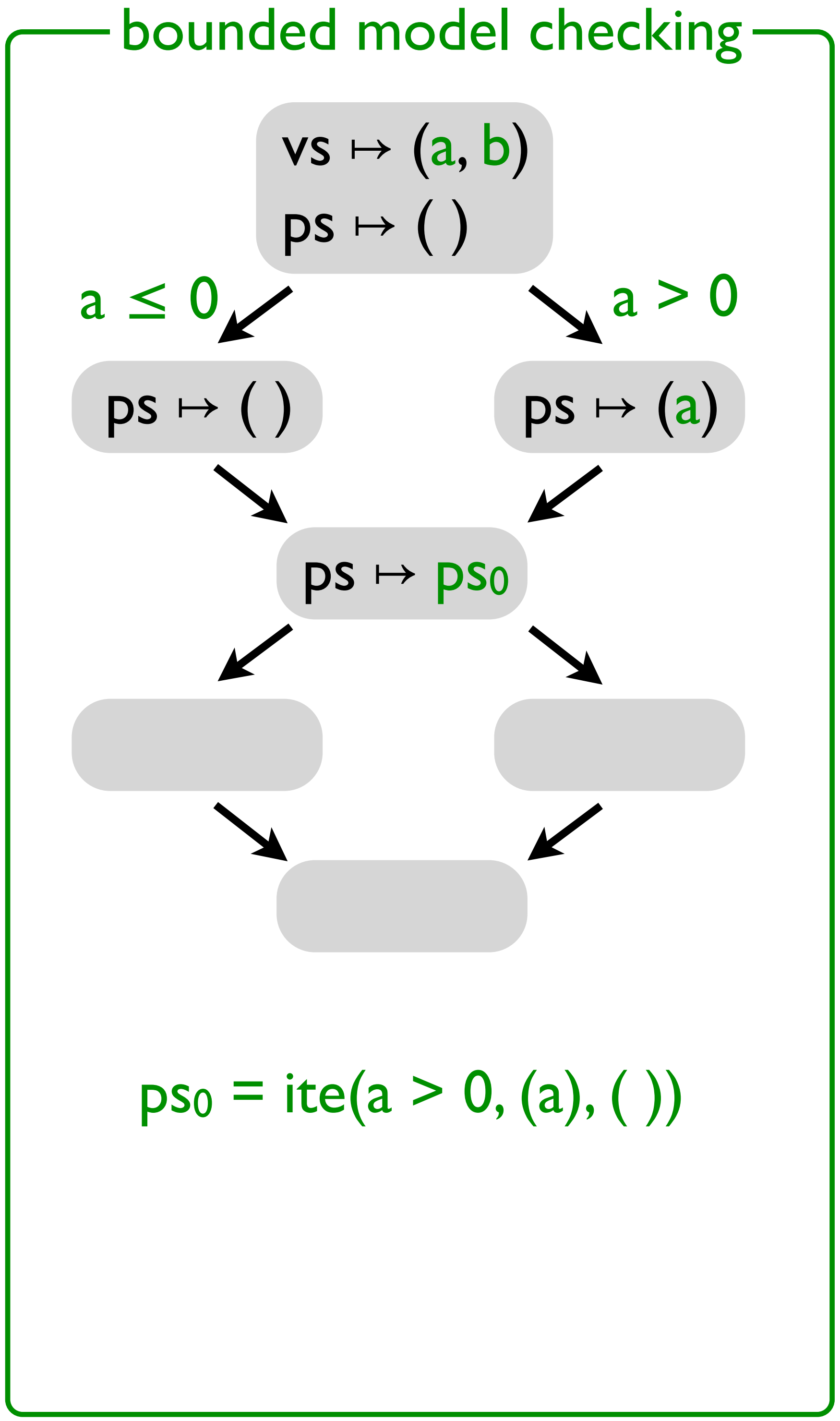
$a \le 0$ $a > 0$

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

$b \le 0$ $b > 0$ $b \le 0$ $b > 0$

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

$$\begin{Bmatrix} a \le 0 \\ b \le 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a \le 0 \\ b > 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b \le 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b > 0 \\ \text{true} \end{Bmatrix}$$



bounded model checking

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

$a \le 0$ $a > 0$

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

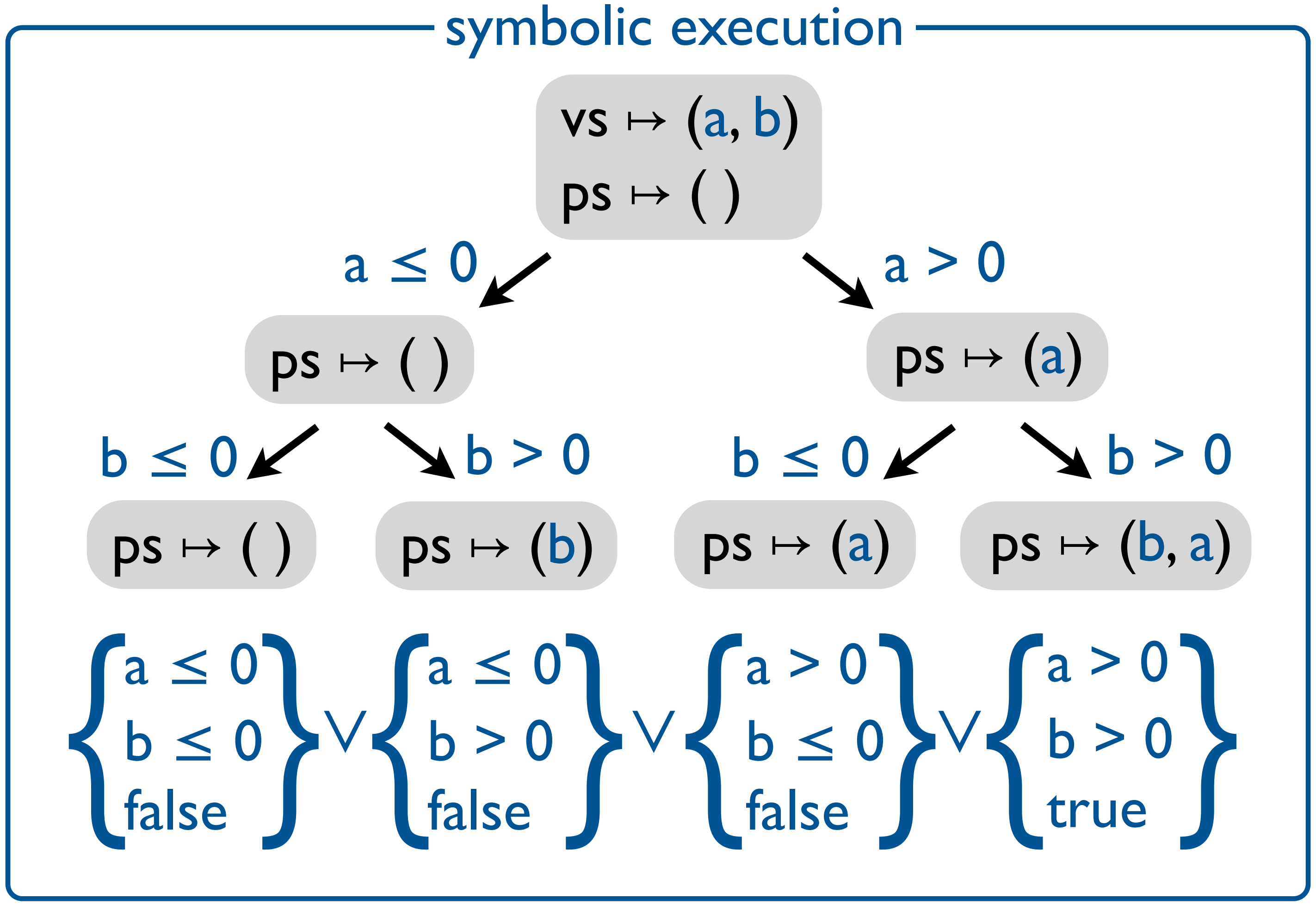
$ps \mapsto ps_0$

$ps_0 = \text{ite}(a > 0, (a), (\,))$

Design space of symbolic encodings: SE and BMC

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

symbolic execution



vs ↦ (a, b)
ps ↦ ( )

$a \le 0$ → ps ↦ ( )

$a > 0$ → ps ↦ (a)

$b \le 0$ → ps ↦ ( )

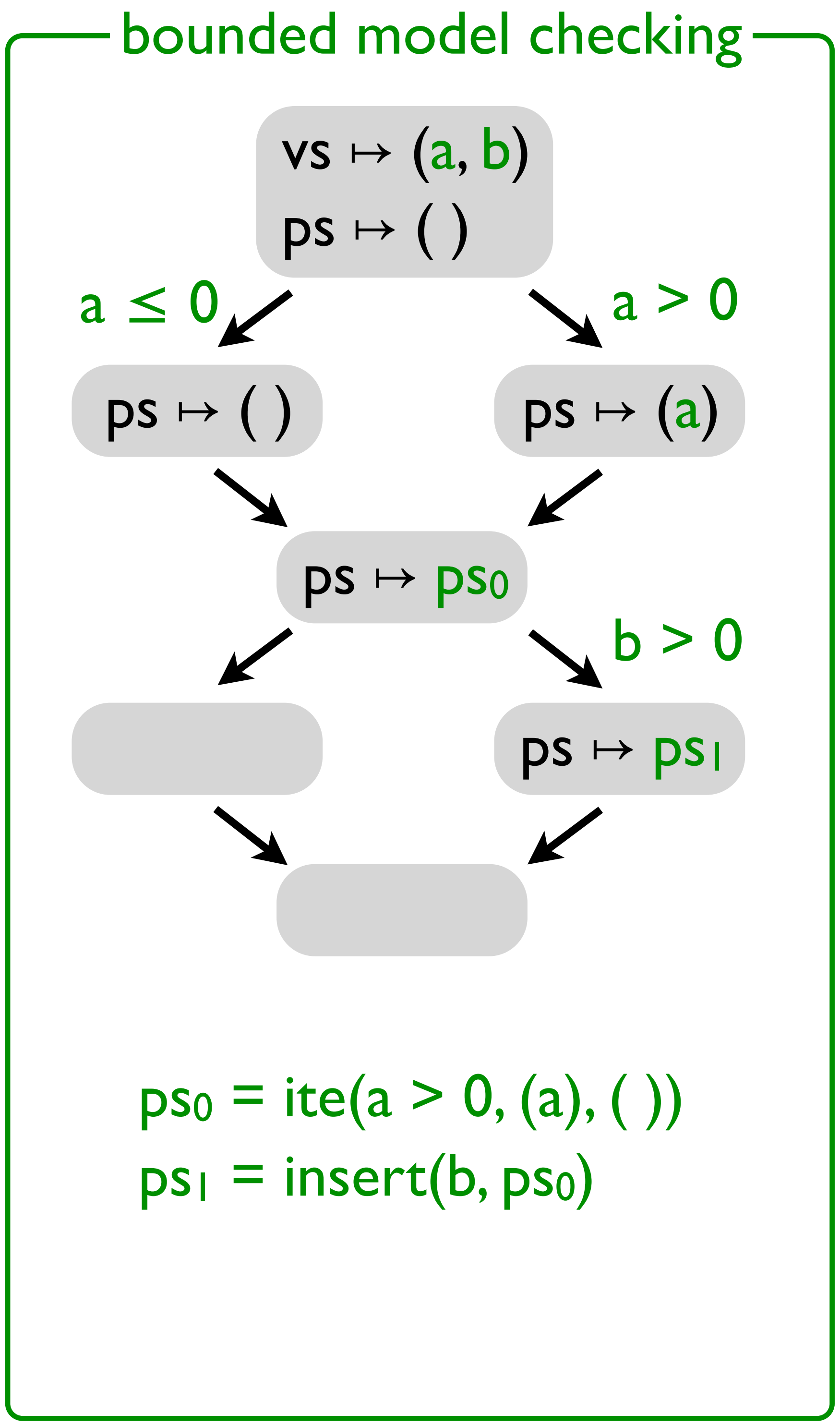
$b > 0$ → ps ↦ (b)

$b \le 0$ → ps ↦ (a)

$b > 0$ → ps ↦ (b, a)

$$\left\{\begin{array}{l} a \le 0 \\ b \le 0 \\ \text{false} \end{array}\right\} \vee \left\{\begin{array}{l} a \le 0 \\ b > 0 \\ \text{false} \end{array}\right\} \vee \left\{\begin{array}{l} a > 0 \\ b \le 0 \\ \text{false} \end{array}\right\} \vee \left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$

bounded model checking



vs ↦ (a, b)
ps ↦ ( )

$a \le 0$ → ps ↦ ( )

$a > 0$ → ps ↦ (a)

ps ↦ ps_0

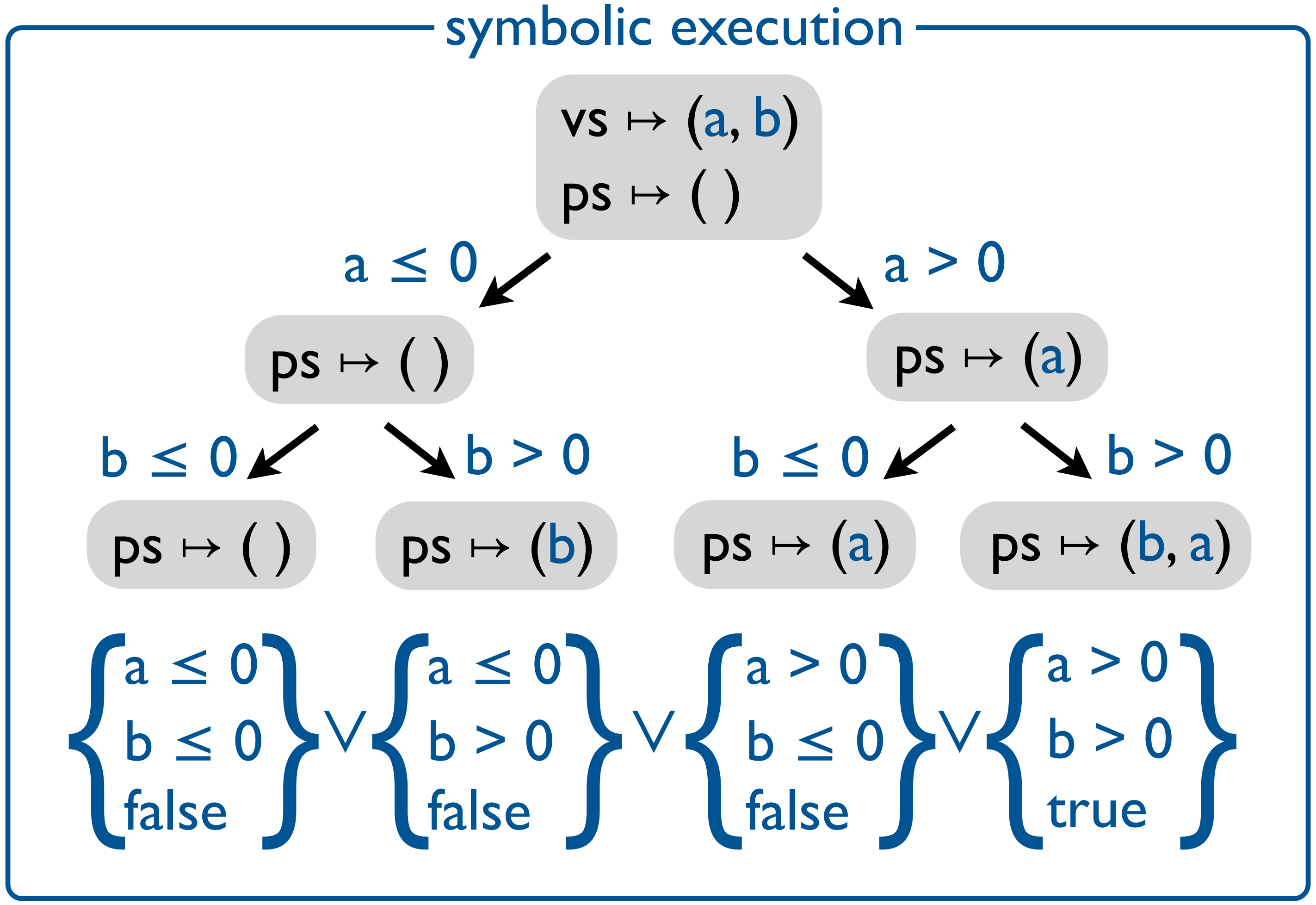
$b > 0$ → ps ↦ ps_1

ps_0 = ite(a > 0, (a), ( ))

ps_1 = insert(b, ps_0)

Design space of symbolic encodings: SE and BMC

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



symbolic execution

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

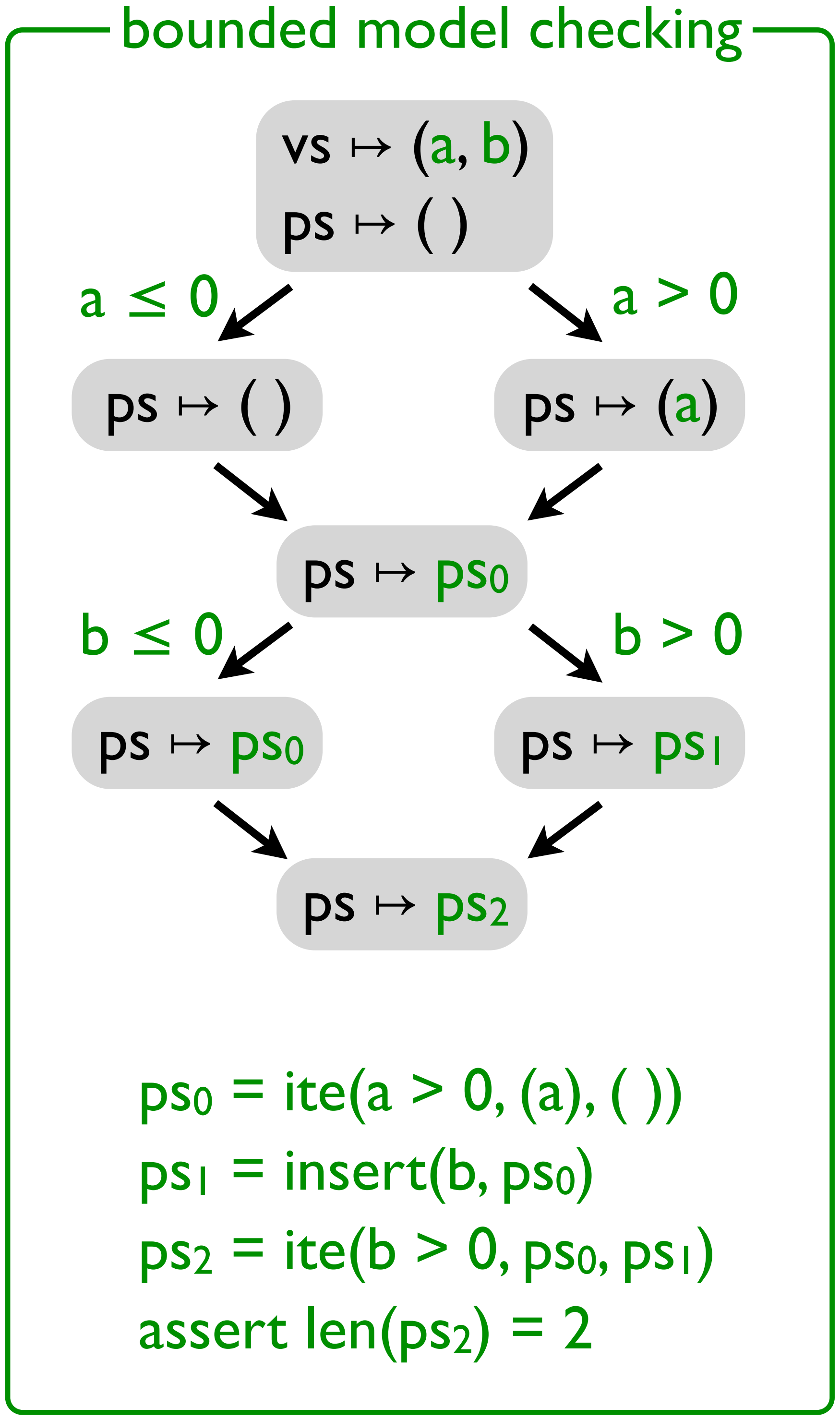
$a \leq 0$ $a > 0$

$ps \mapsto ()$ $ps \mapsto (a)$

$b \leq 0$ $b > 0$ $b \leq 0$ $b > 0$

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

$$\begin{Bmatrix} a \leq 0 \\ b \leq 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a \leq 0 \\ b > 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b \leq 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b > 0 \\ \text{true} \end{Bmatrix}$$



bounded model checking

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

$a \leq 0$ $a > 0$

$ps \mapsto ()$ $ps \mapsto (a)$

$ps \mapsto ps_0$

$b \leq 0$ $b > 0$

$ps \mapsto ps_0$ $ps \mapsto ps_1$

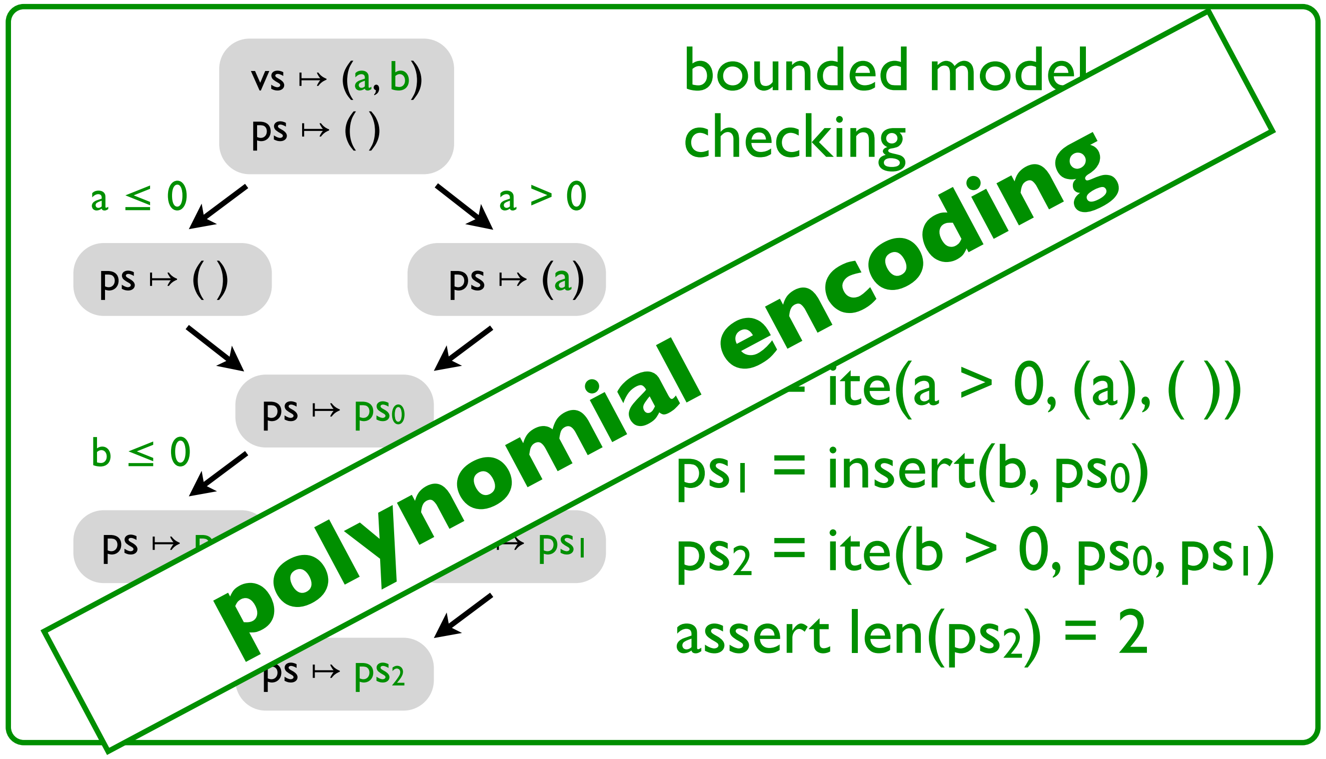
$ps \mapsto ps_2$

$ps_0 = \text{ite}(a > 0, (a), ())$
$ps_1 = \text{insert}(b, ps_0)$
$ps_2 = \text{ite}(b > 0, ps_0, ps_1)$
$\text{assert len}(ps_2) = 2$

Design space of symbolic encodings: best of all worlds?

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

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

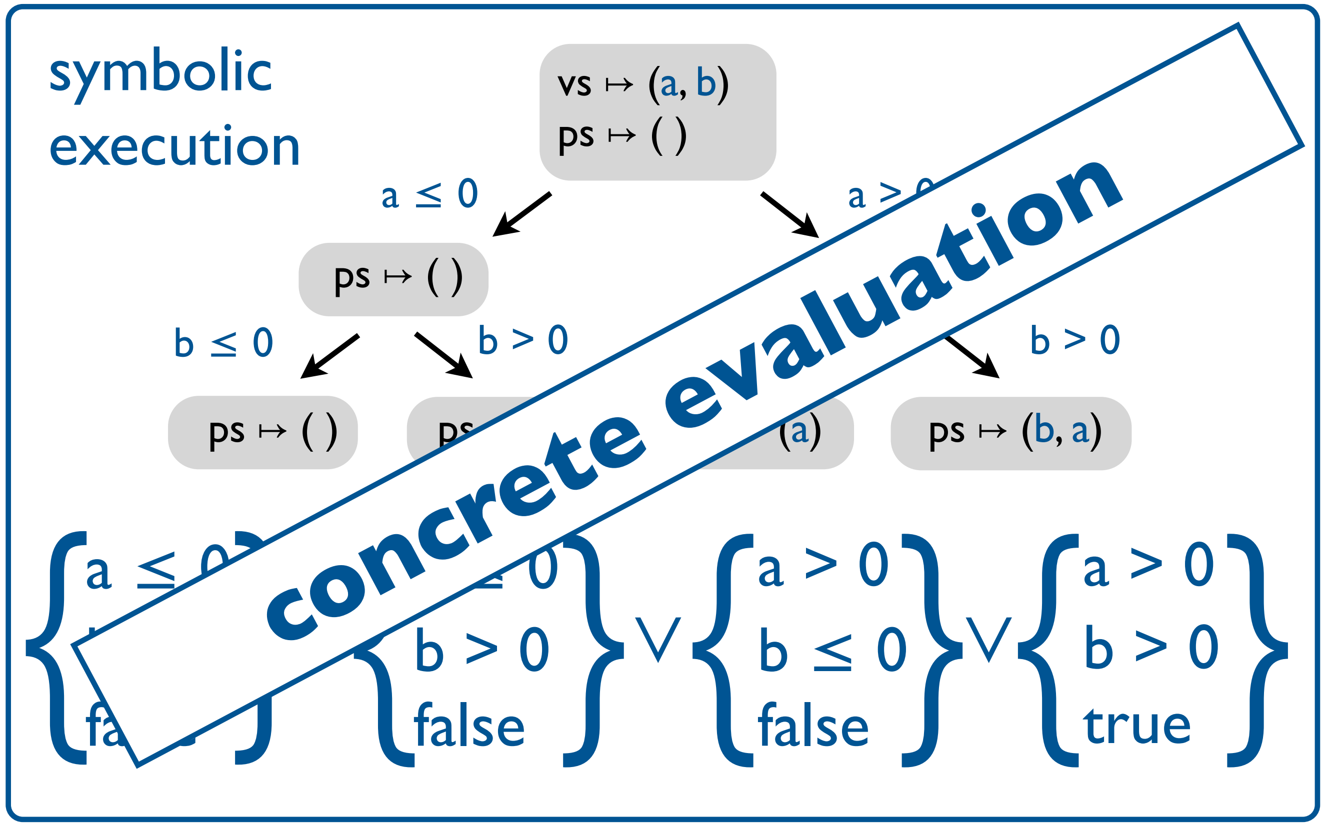


$ite(a > 0, (a), ( ))$

$ps_1 = insert(b, ps_0)$

$ps_2 = ite(b > 0, ps_0, ps_1)$

assert $len(ps_2) = 2$



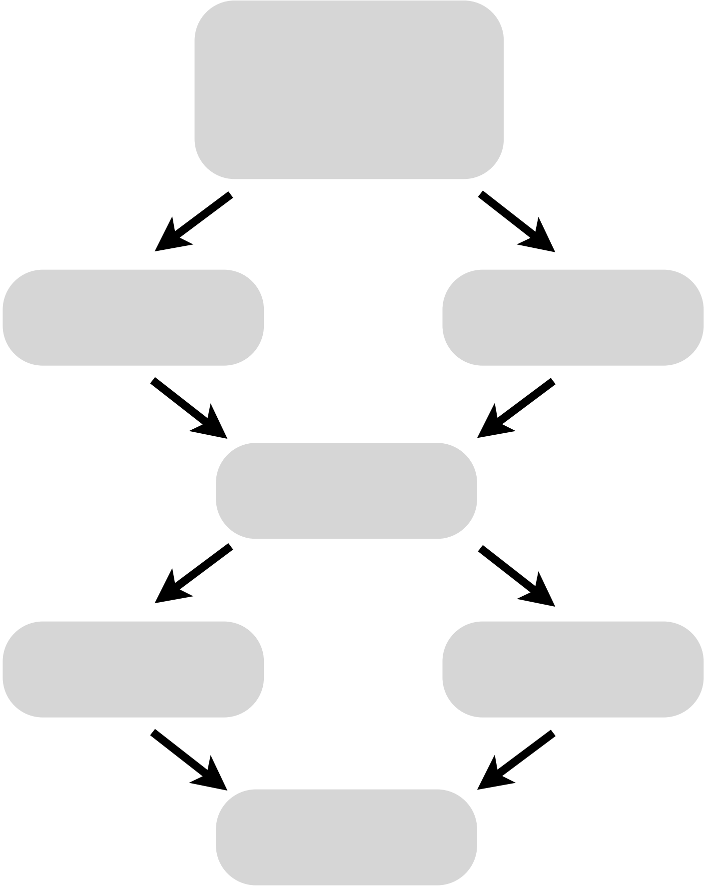
$\{a \leq 0 \ldots fa\ldots\} \vee \{\ldots \leq 0,\ b > 0,\ false\} \vee \{a > 0,\ b \leq 0,\ false\} \vee \{a > 0,\ b > 0,\ true\}$

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

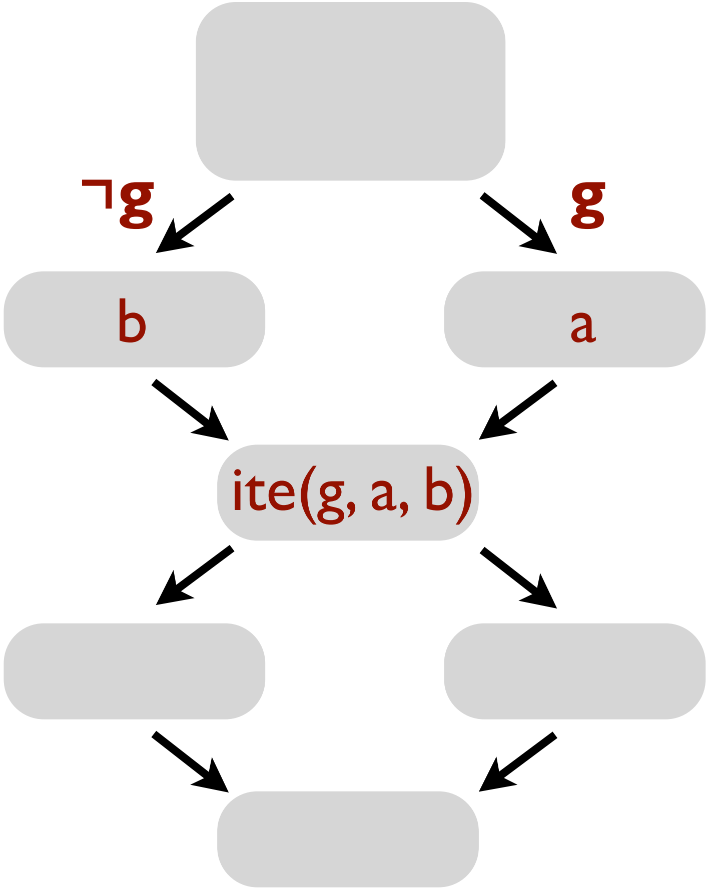


A new design: type-driven state merging

```
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

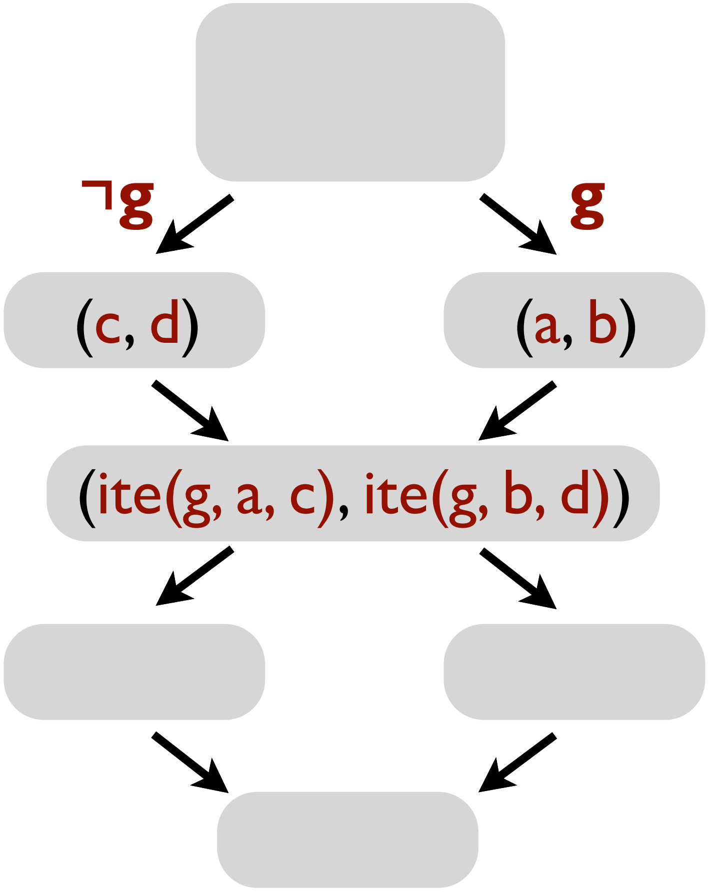


A new design: type-driven state merging

```
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

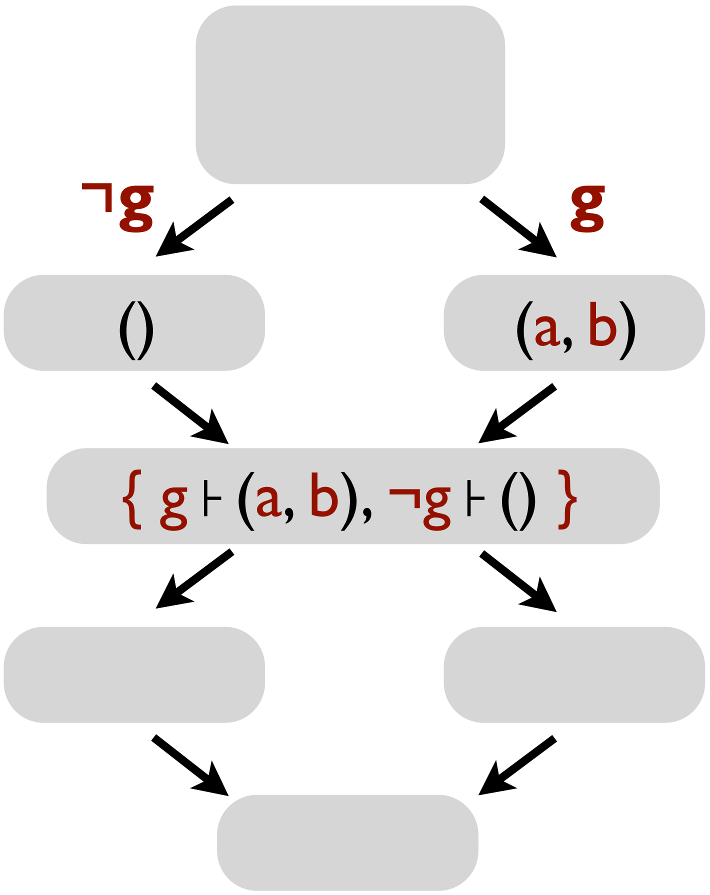


A new design: type-driven state merging

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

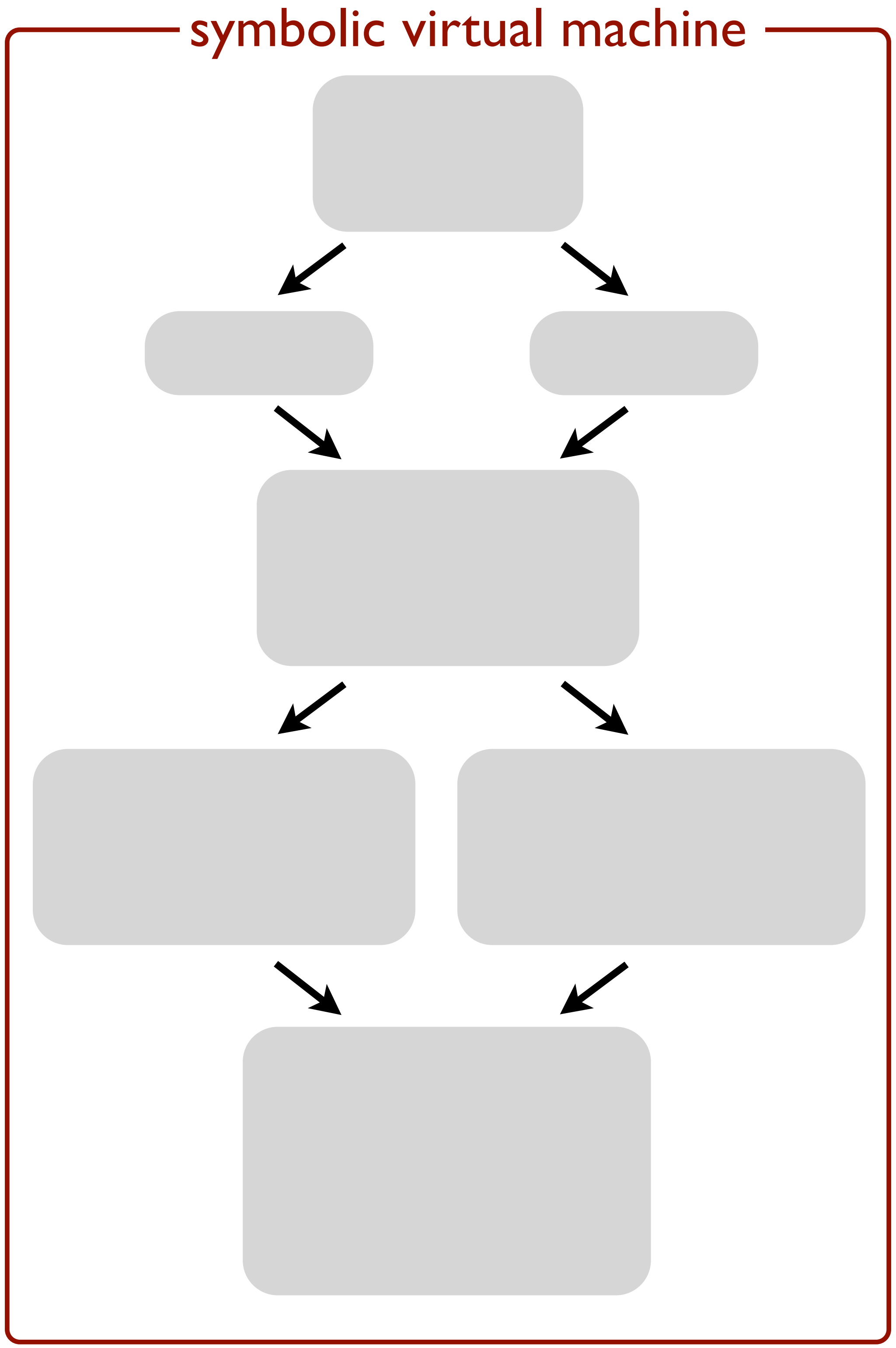
Merge instances of

- primitive types: symbolically
- value types: structurally
- all other types: via unions



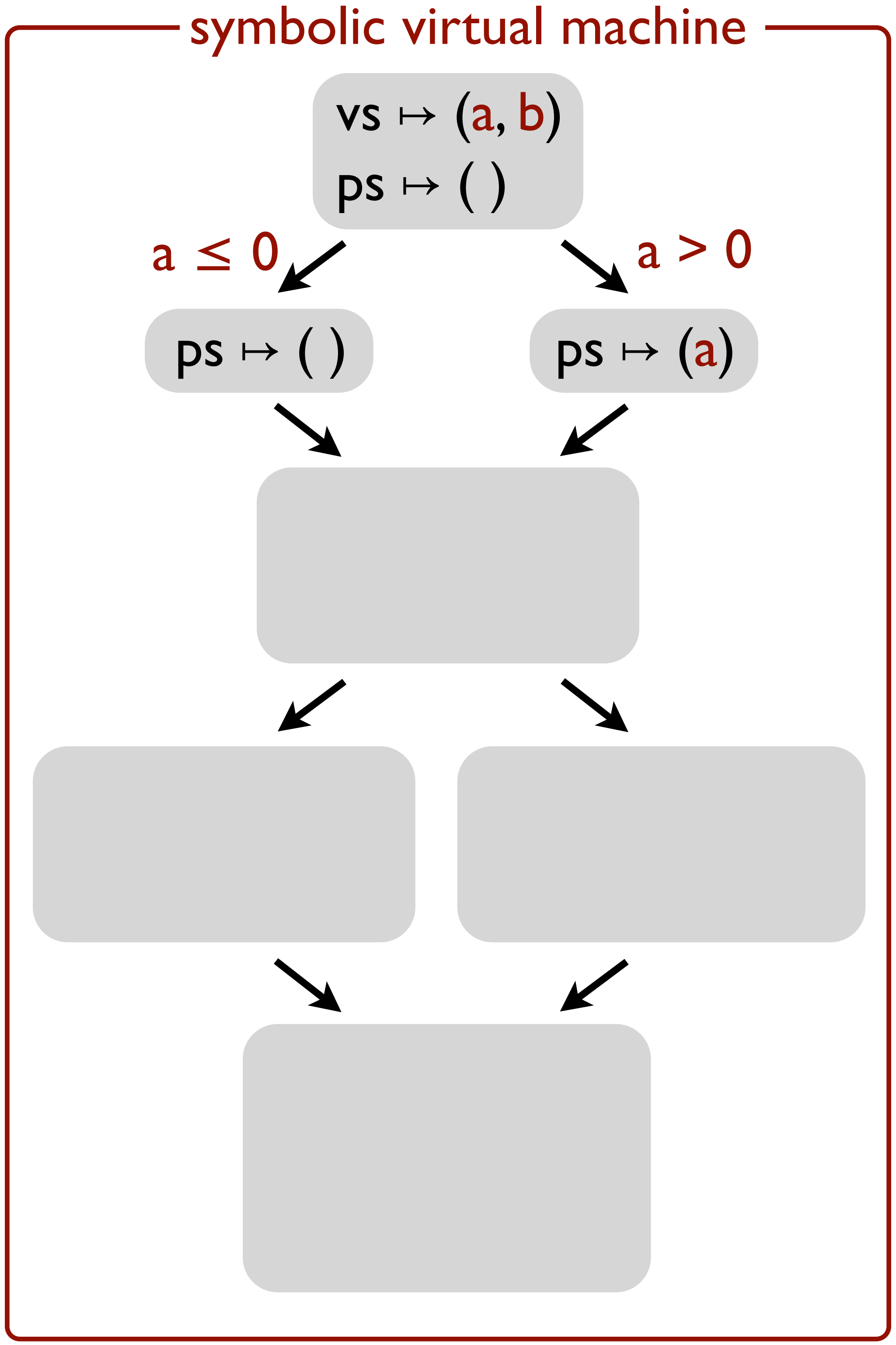
A new design: type-driven state merging

```
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)
```



symbolic virtual machine

vs ↦ (a, b)
ps ↦ ( )

a ≤ 0

a > 0

ps ↦ ( )

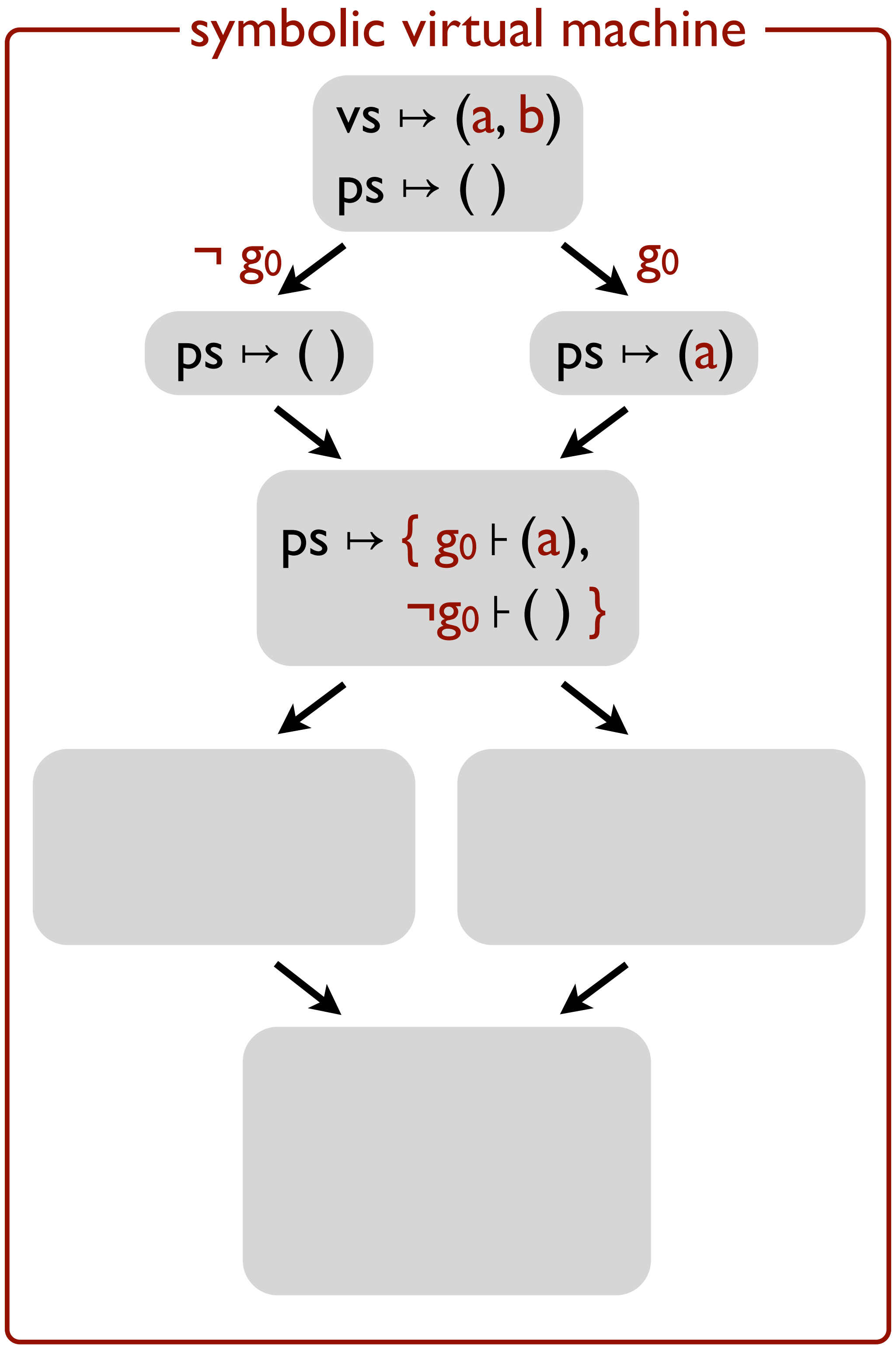
ps ↦ (a)

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)
```

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

$g_0 = a > 0$



symbolic virtual machine

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

$\neg g_0$ g_0

$ps \mapsto ()$ $ps \mapsto (a)$

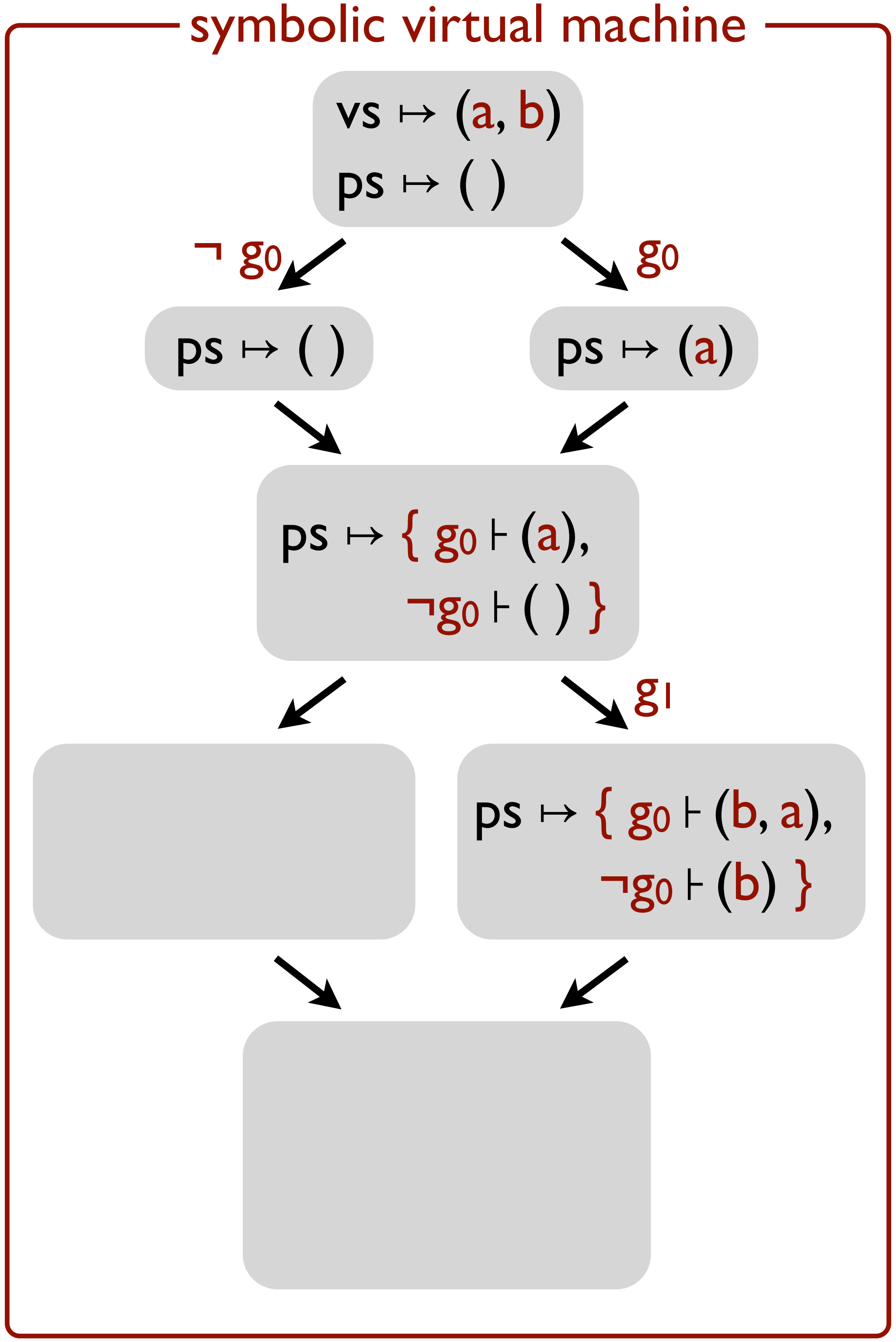
$ps \mapsto \{ g_0 \vdash (a),$
$\neg g_0 \vdash () \}$

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)
```

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

$g_0 = a > 0$
$g_1 = b > 0$



symbolic virtual machine

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

$\neg\, g_0$ $\qquad$ g_0

$ps \mapsto (\,)$ $\qquad$ $ps \mapsto (a)$

$ps \mapsto \{\, g_0 \vdash (a), \; \neg g_0 \vdash (\,) \,\}$

g_1

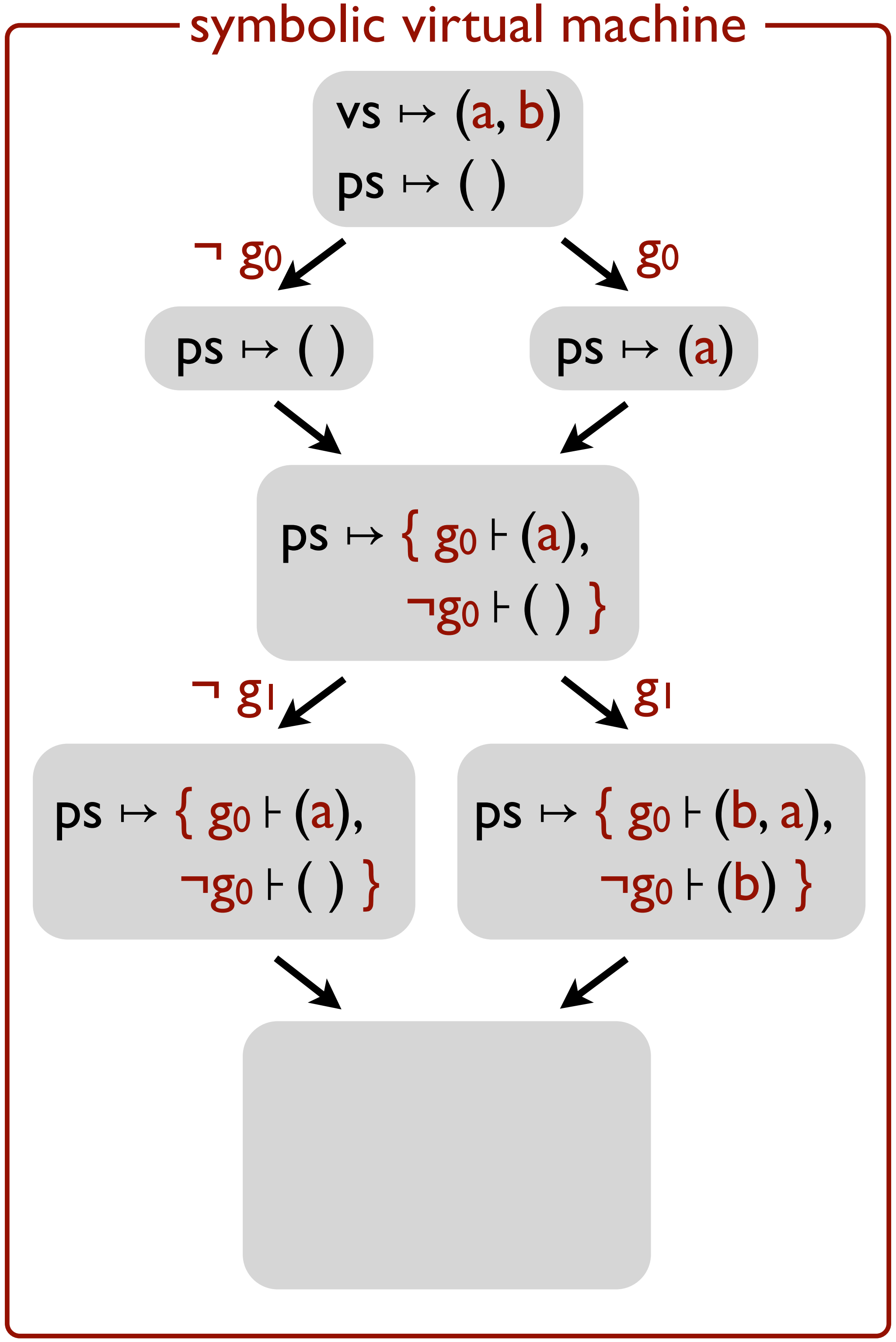
$ps \mapsto \{\, g_0 \vdash (b, a), \; \neg g_0 \vdash (b) \,\}$

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)
```

$g_0 = a > 0$

$g_1 = b > 0$



symbolic virtual machine

$vs \mapsto (a, b)$

$ps \mapsto (\,)$

$\neg\, g_0$

g_0

$ps \mapsto (\,)$

$ps \mapsto (a)$

$ps \mapsto \{\, g_0 \vdash (a),\ \neg g_0 \vdash (\,) \,\}$

$\neg\, g_1$

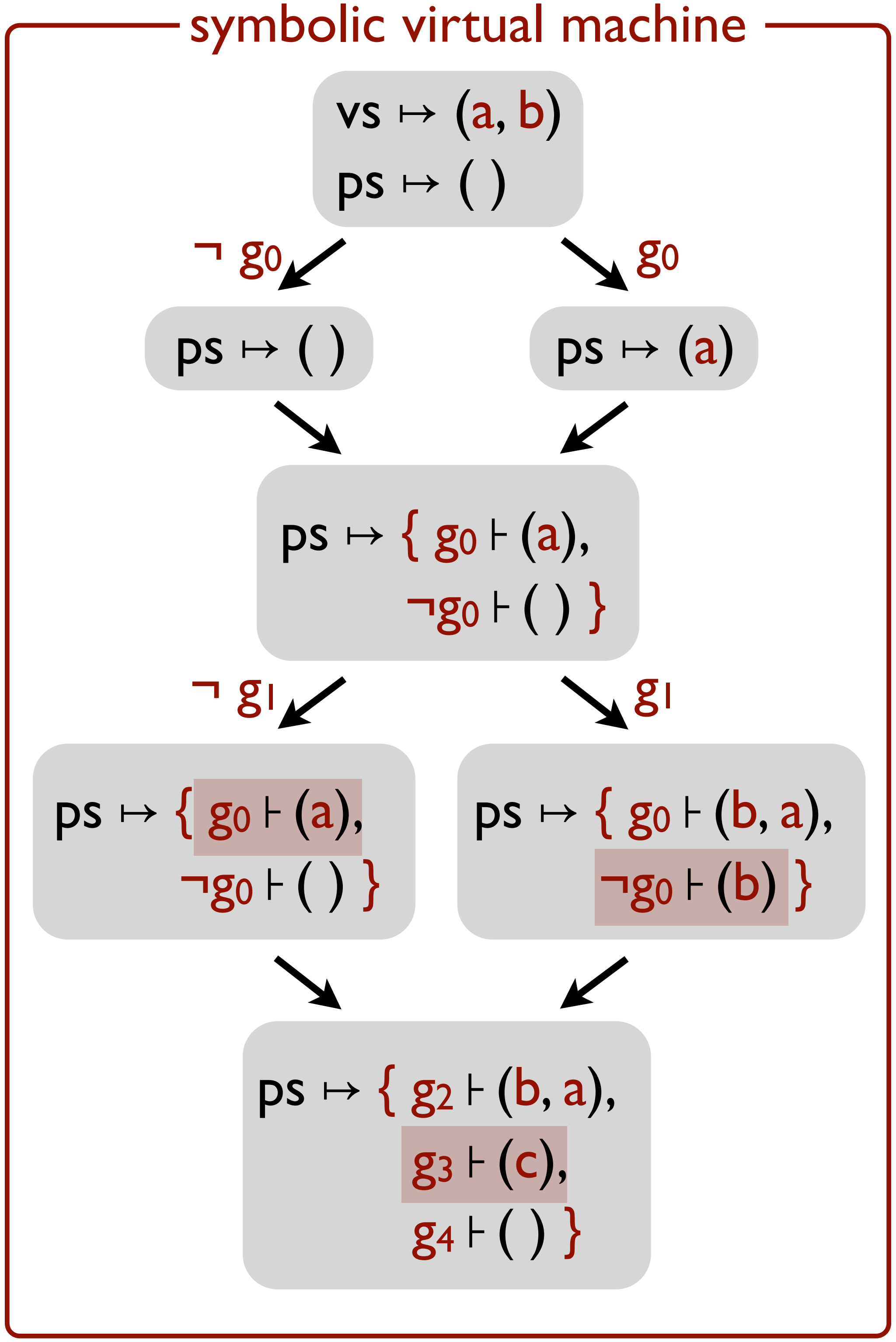
g_1

$ps \mapsto \{\, g_0 \vdash (a),\ \neg g_0 \vdash (\,) \,\}$

$ps \mapsto \{\, g_0 \vdash (b, a),\ \neg g_0 \vdash (b) \,\}$

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)
```



symbolic virtual machine

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

$\neg g_0$ g_0

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

$ps \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash (\,) \}$

$\neg g_1$ g_1

$ps \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash (\,) \}$ $ps \mapsto \{ g_0 \vdash (b, a), \neg g_0 \vdash (b) \}$

$ps \mapsto \{ g_2 \vdash (b, a), g_3 \vdash (c), g_4 \vdash (\,) \}$

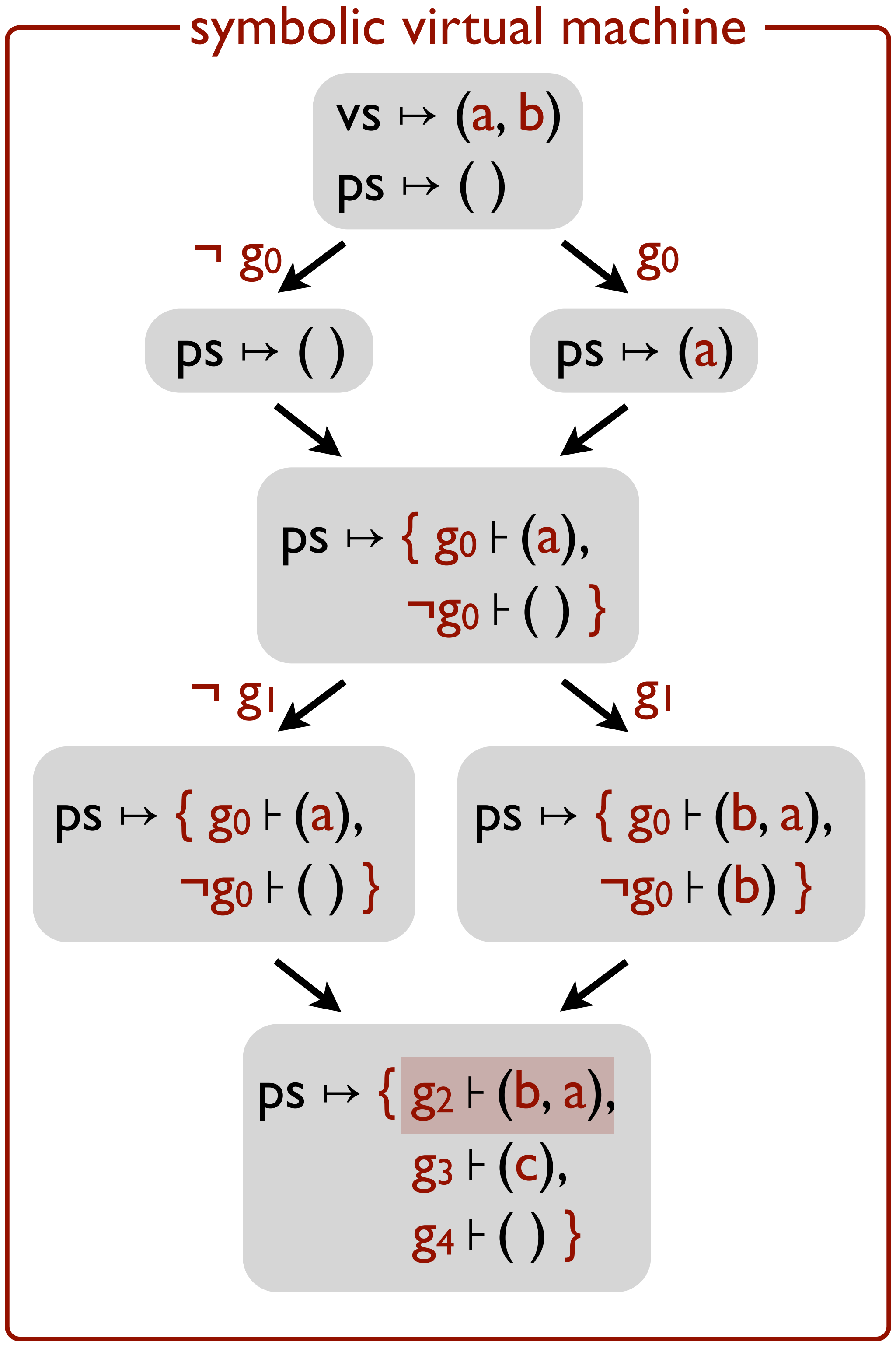
$g_0 = a > 0$
$g_1 = b > 0$
$g_2 = g_0 \wedge g_1$
$g_3 = \neg(g_0 \Leftrightarrow g_1)$
$g_4 = \neg g_0 \wedge \neg g_1$
$c = ite(g_1, b, a)$

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.

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



symbolic virtual machine

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

$\neg\, g_0$
g_0

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

$ps \mapsto \{\, g_0 \vdash (a), \neg g_0 \vdash (\,) \,\}$

$\neg\, g_1$
g_1

$ps \mapsto \{\, g_0 \vdash (a), \neg g_0 \vdash (\,) \,\}$
$ps \mapsto \{\, g_0 \vdash (b, a), \neg g_0 \vdash (b) \,\}$

$ps \mapsto \{\, g_2 \vdash (b, a), g_3 \vdash (c), g_4 \vdash (\,) \,\}$

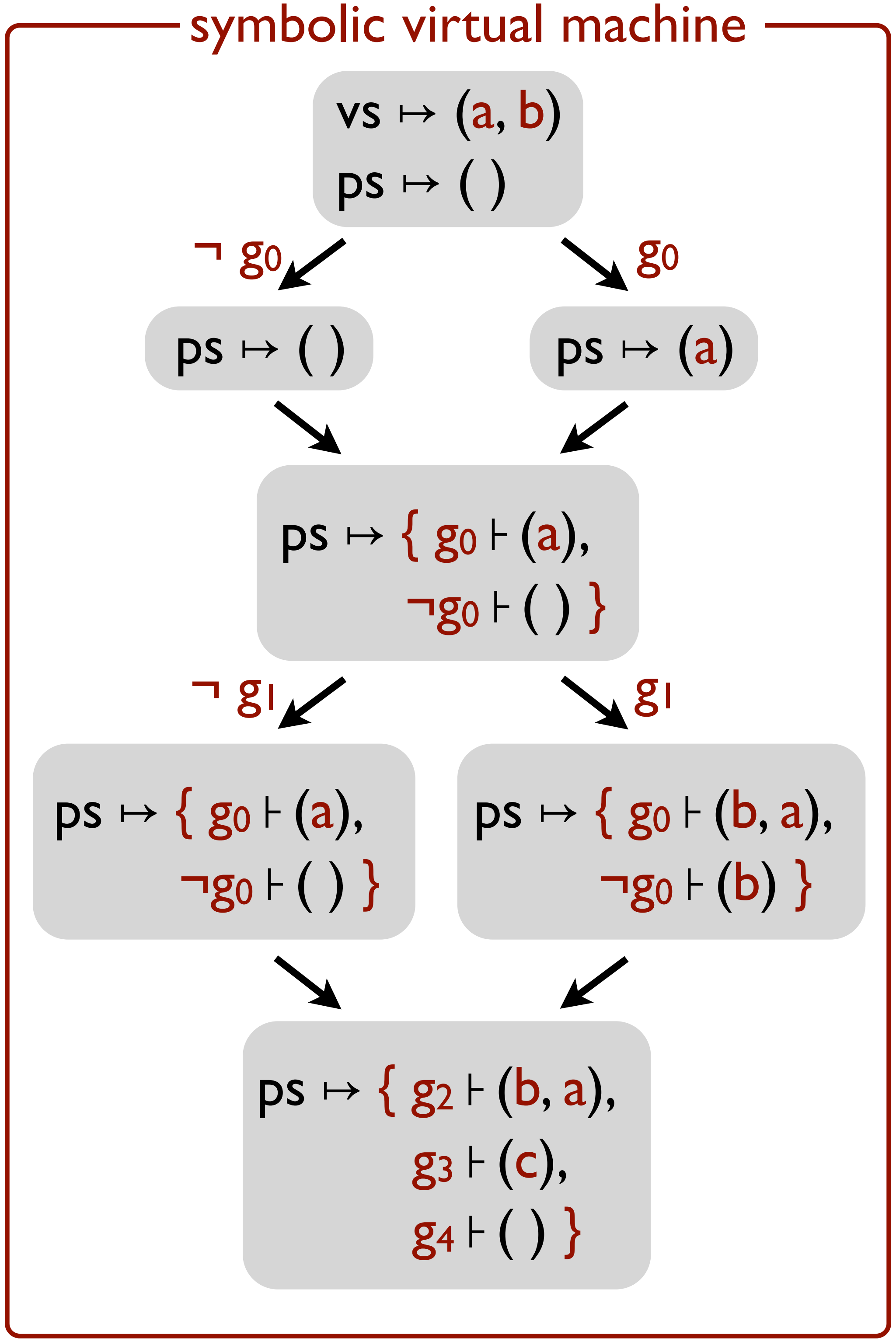
A new design: type-driven state merging

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

polynomial encoding

concrete evaluation

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



symbolic virtual machine

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

$\neg\, g_0$ | g_0

$ps \mapsto (\,)$ | $ps \mapsto (a)$

$ps \mapsto \{\, g_0 \vdash (a),\ \neg g_0 \vdash (\,) \,\}$

$\neg\, g_1$ | g_1

$ps \mapsto \{\, g_0 \vdash (a),\ \neg g_0 \vdash (\,) \,\}$ | $ps \mapsto \{\, g_0 \vdash (b, a),\ \neg g_0 \vdash (b) \,\}$

$ps \mapsto \{\, g_2 \vdash (b, a),\ g_3 \vdash (c),\ g_4 \vdash (\,) \,\}$

A new design: type-driven state merging

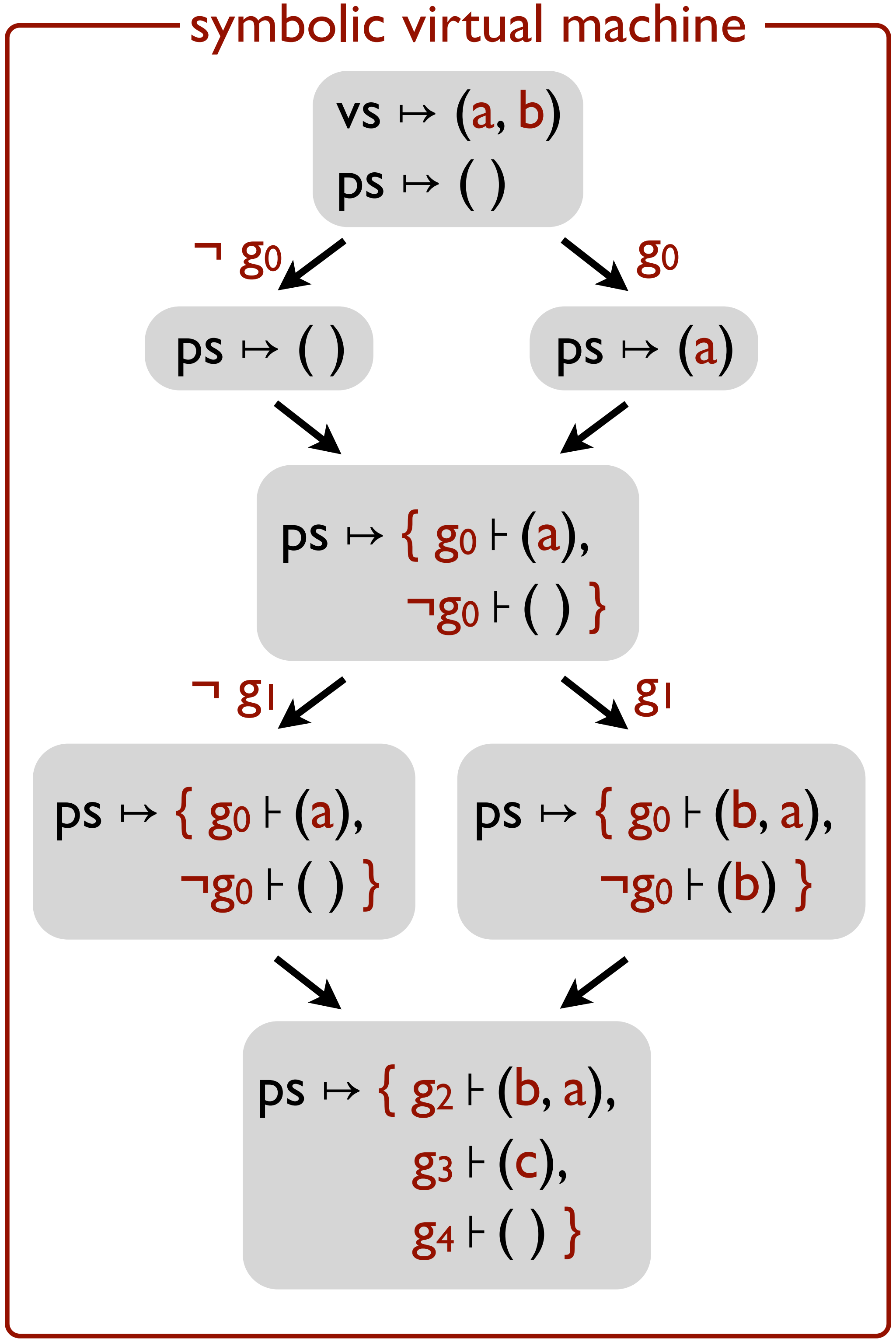
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

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

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



symbolic virtual machine

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

$\neg\, g_0$ | g_0

$ps \mapsto (\,)$ | $ps \mapsto (a)$

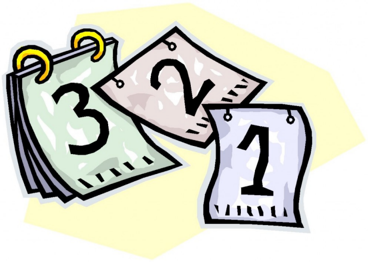
$ps \mapsto \{\, g_0 \vdash (a), \neg g_0 \vdash (\,) \,\}$

$\neg\, g_1$ | g_1

$ps \mapsto \{\, g_0 \vdash (a), \neg g_0 \vdash (\,) \,\}$ | $ps \mapsto \{\, g_0 \vdash (b, a), \neg g_0 \vdash (b) \,\}$

$ps \mapsto \{\, g_2 \vdash (b, a), g_3 \vdash (c), g_4 \vdash (\,) \,\}$

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?

SDSL

SVM

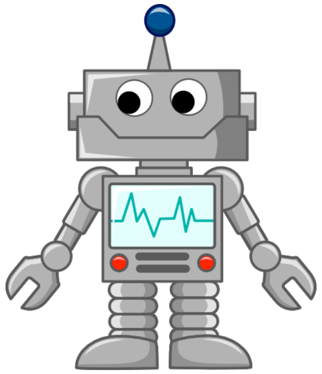
SMT

An easier way: tools as languages

How to build tools by stacking layers of languages.

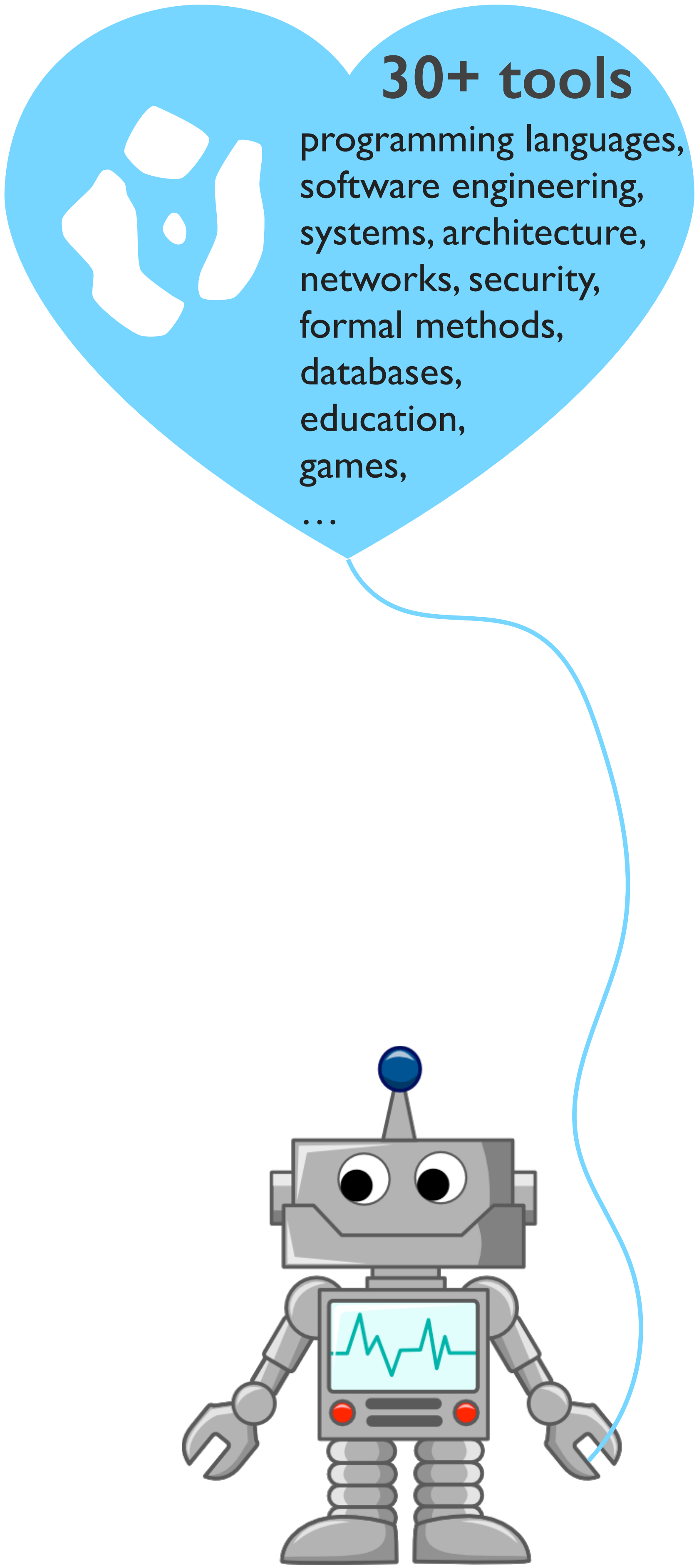
Behind the scenes: symbolic virtual machine

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

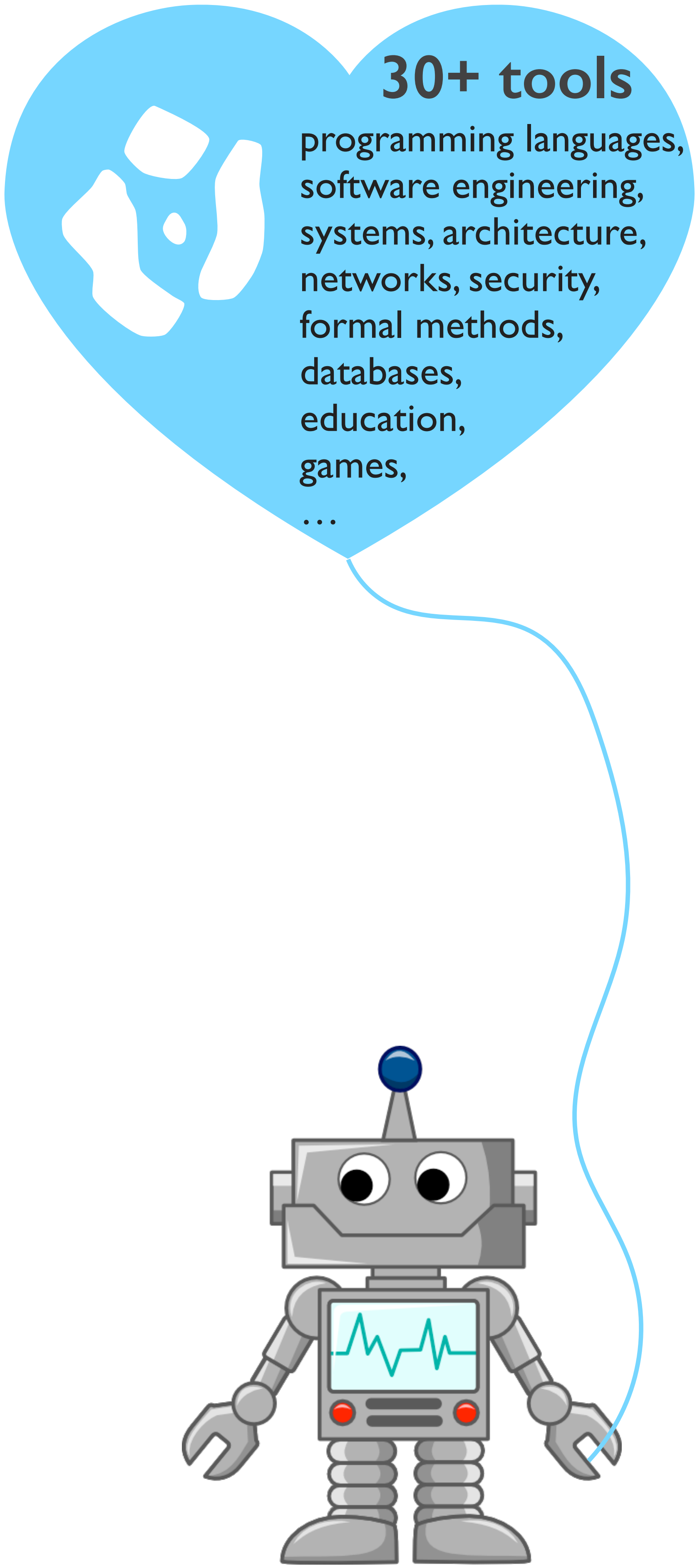
- type systems and programming models
- compilation and parallelization
- safety-critical systems
- test input generation
- software diversification

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



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

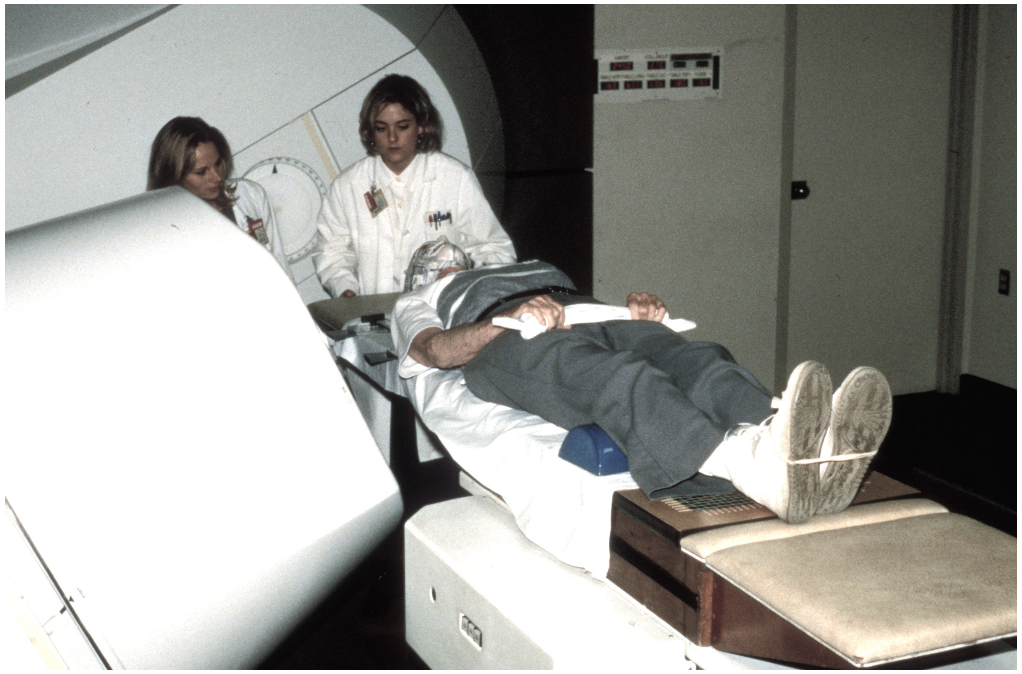
- hints and feedback
- problem generation
- problem-solving strategies [VMCAI'18, FDG'17]

systems, architecture, networks, security, and databases

- memory models
- OS components [SOSP'19]
- data movement for GPUs
- router configuration
- cryptographic protocols

Verifying a radiation therapy system

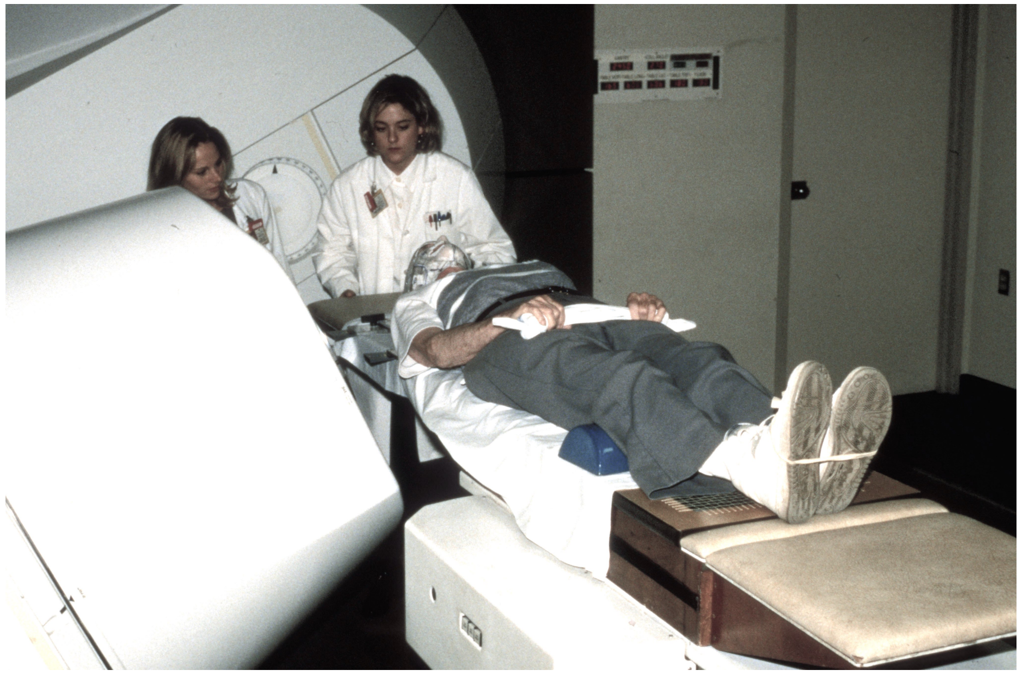
Clinical Neutron Therapy System (CNTS) at UW

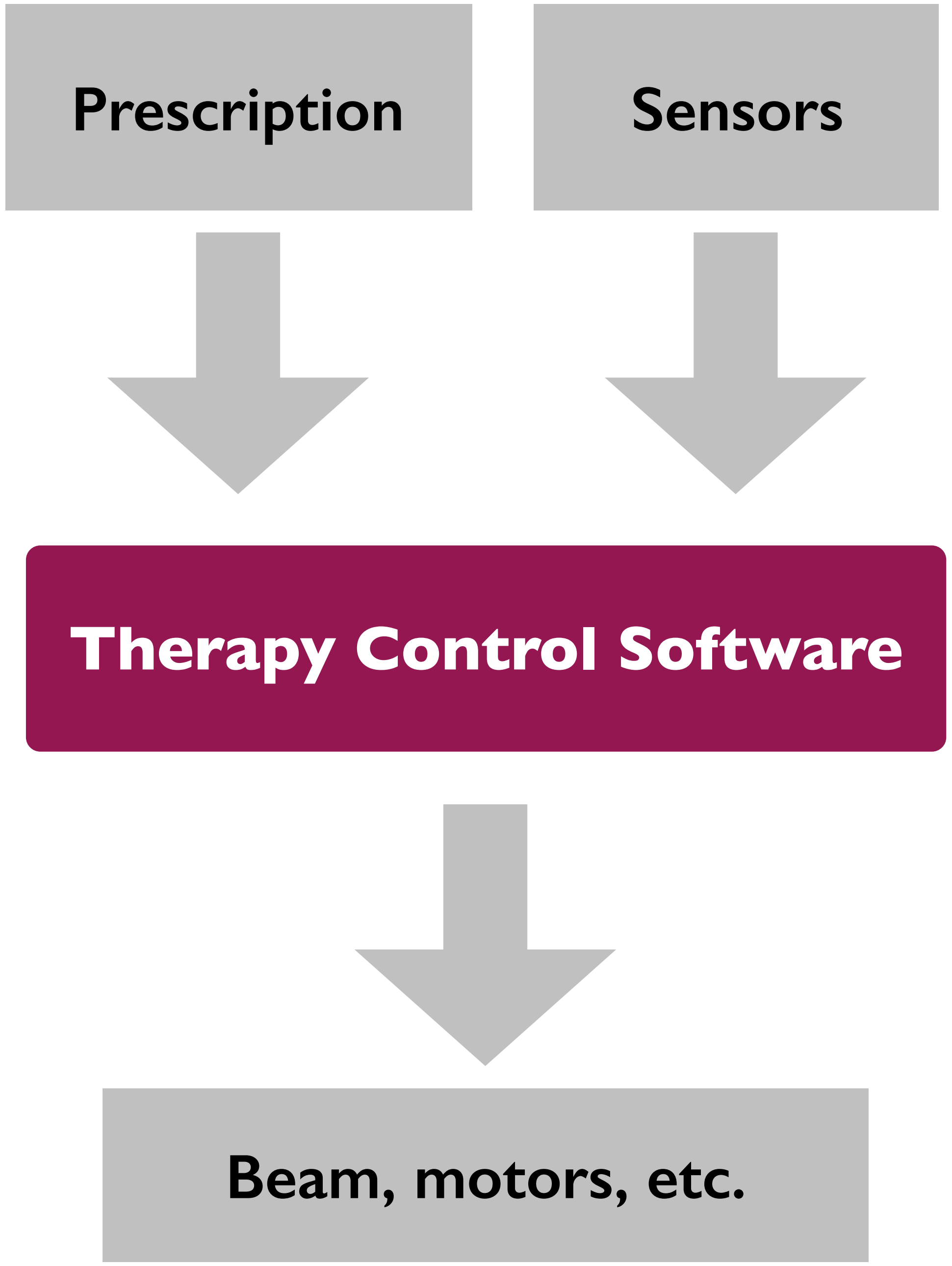


- 30 years of incident-free service.
- Controlled by custom software, built by CNTS engineering staff.
- Third generation of Therapy Control software built recently.

Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW





Prescription

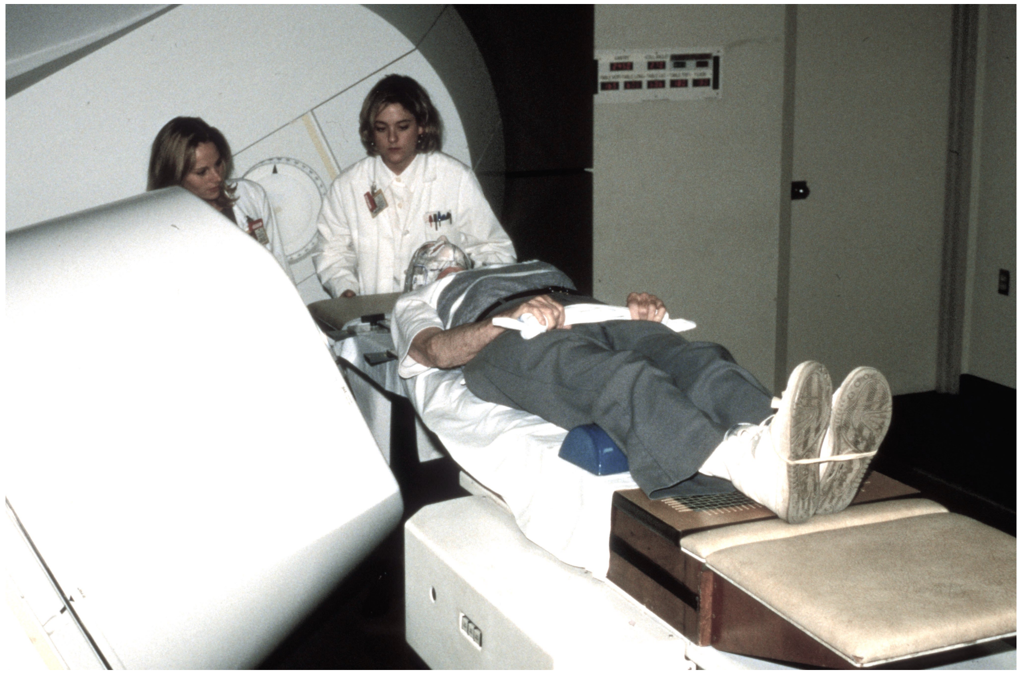
Sensors

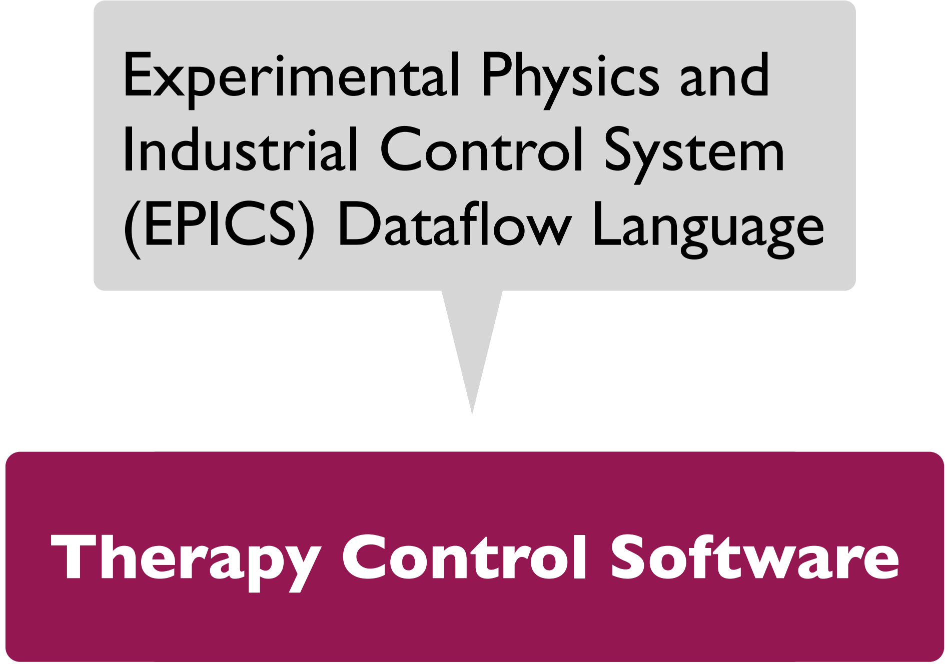
Therapy Control Software

Beam, motors, etc.

Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW



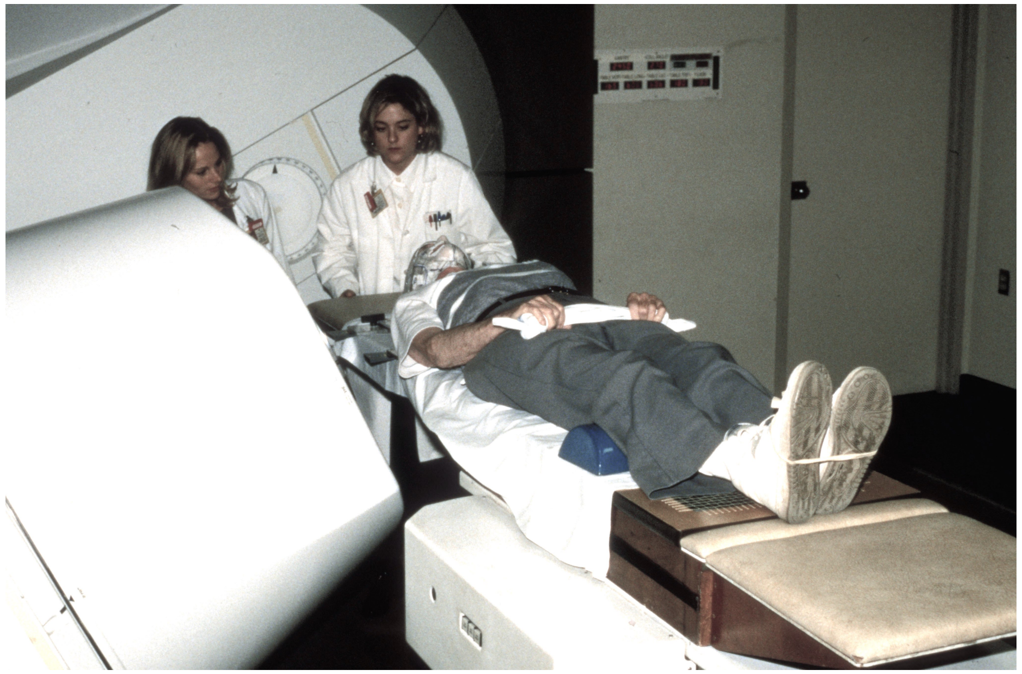


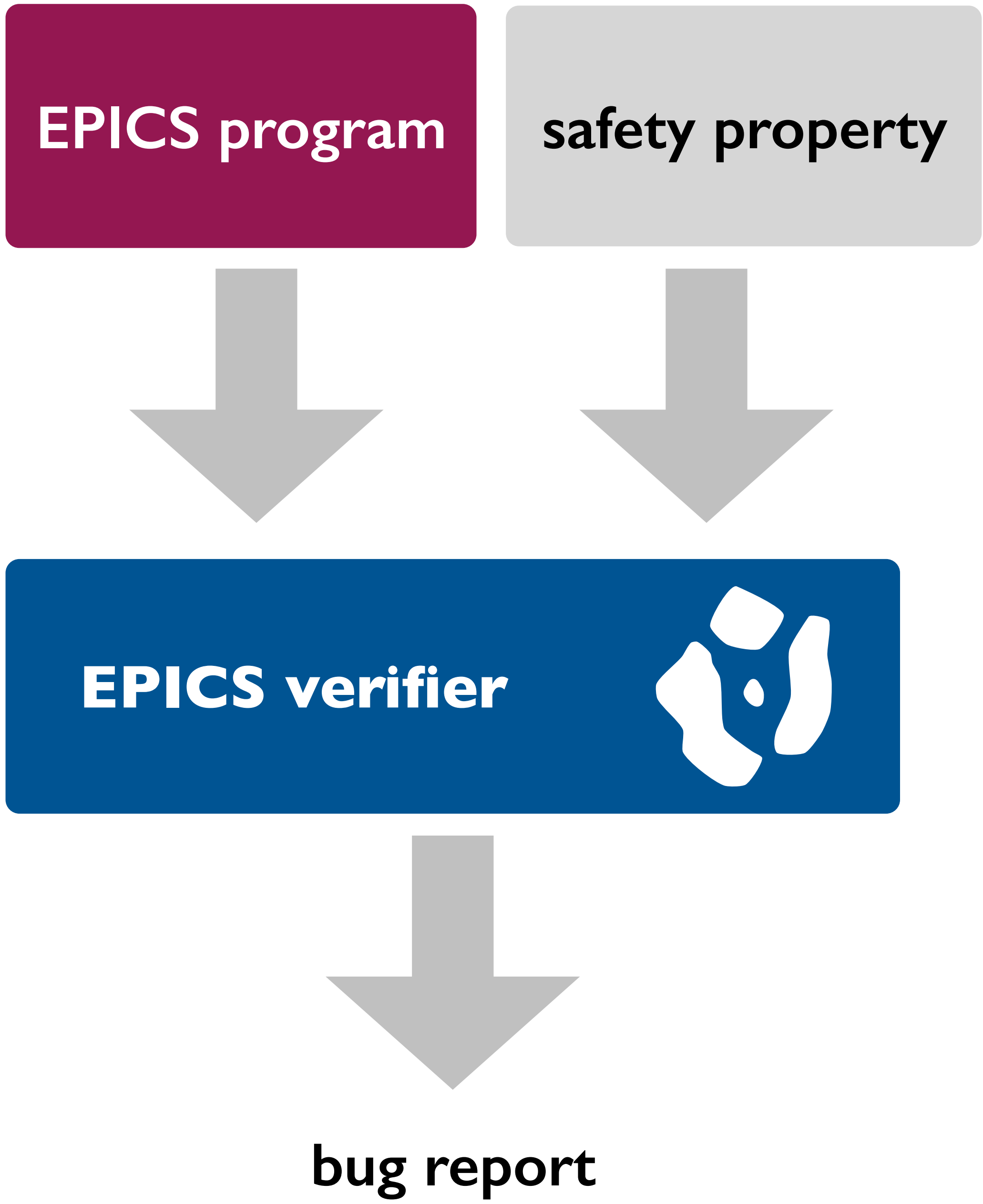
Experimental Physics and Industrial Control System (EPICS) Dataflow Language

Therapy Control Software

Verifying a radiation therapy system

Clinical Neutron Therapy
System (CNTS) at UW





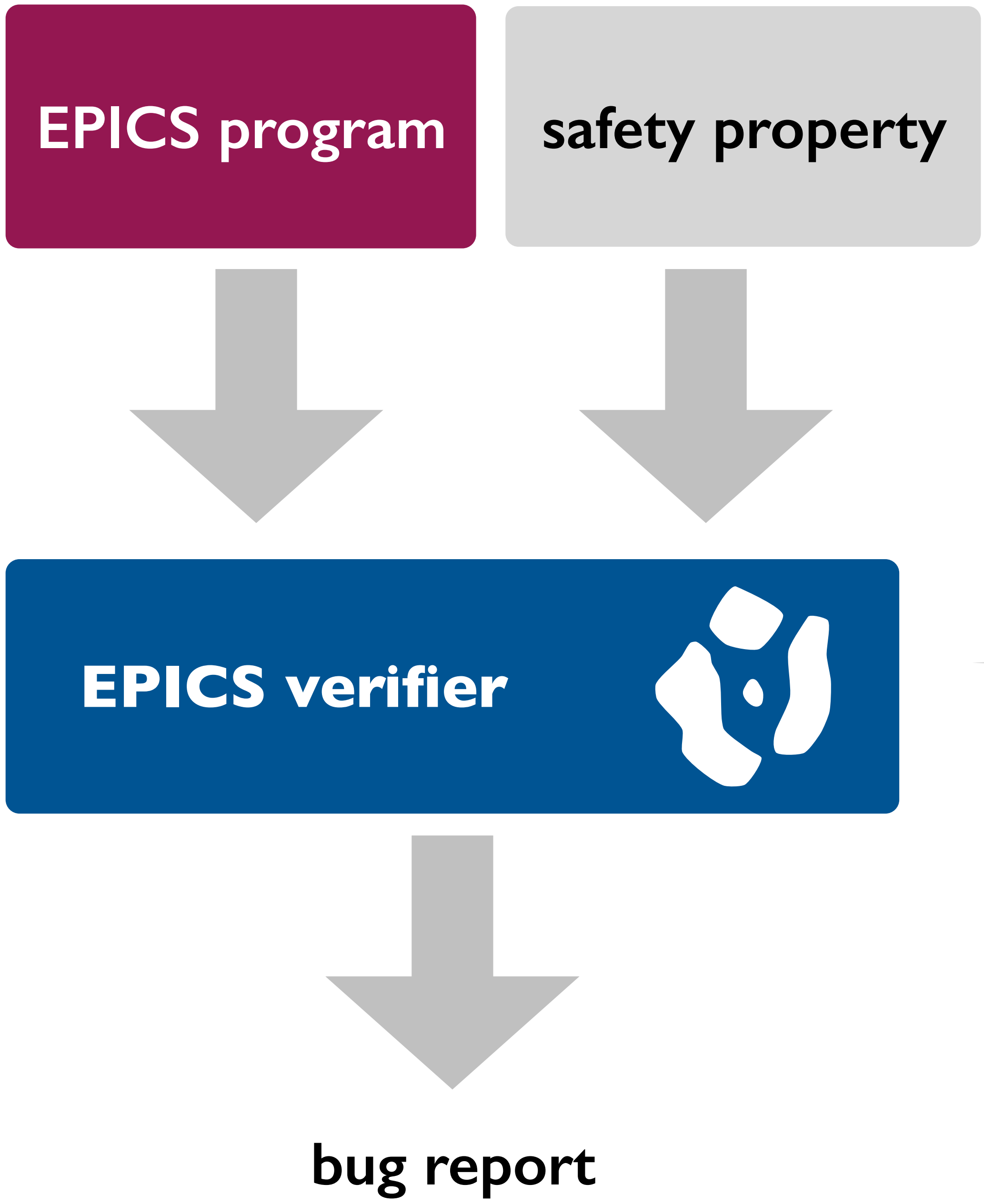
EPICS program

safety property

EPICS verifier

bug report

Verifying a radiation therapy system



EPICS program

safety property

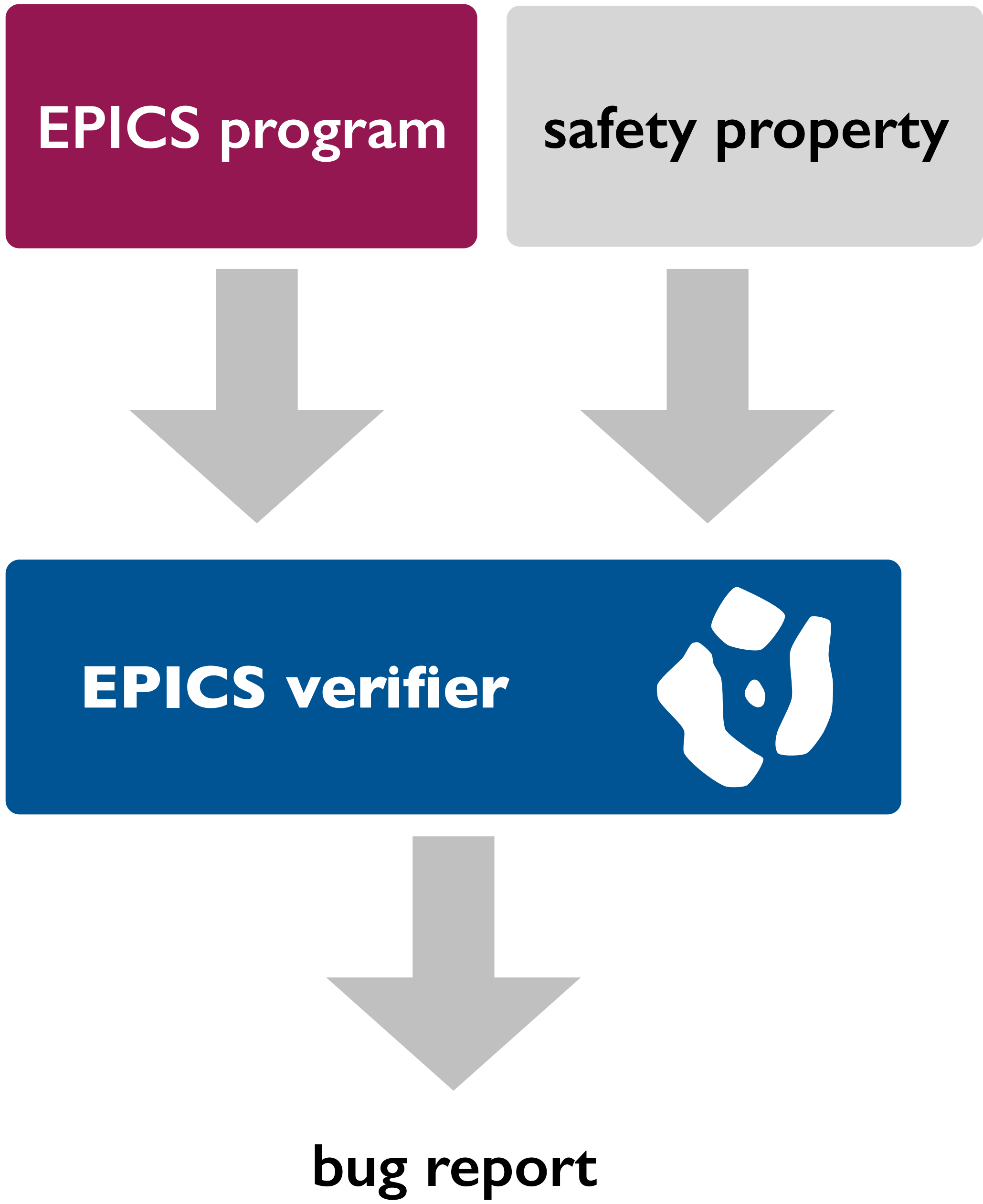
EPICS verifier

bug report

Prototyped in a few days and found bugs.

Calvin Loncaric

Verifying a radiation therapy system



EPICS program

safety property

EPICS verifier

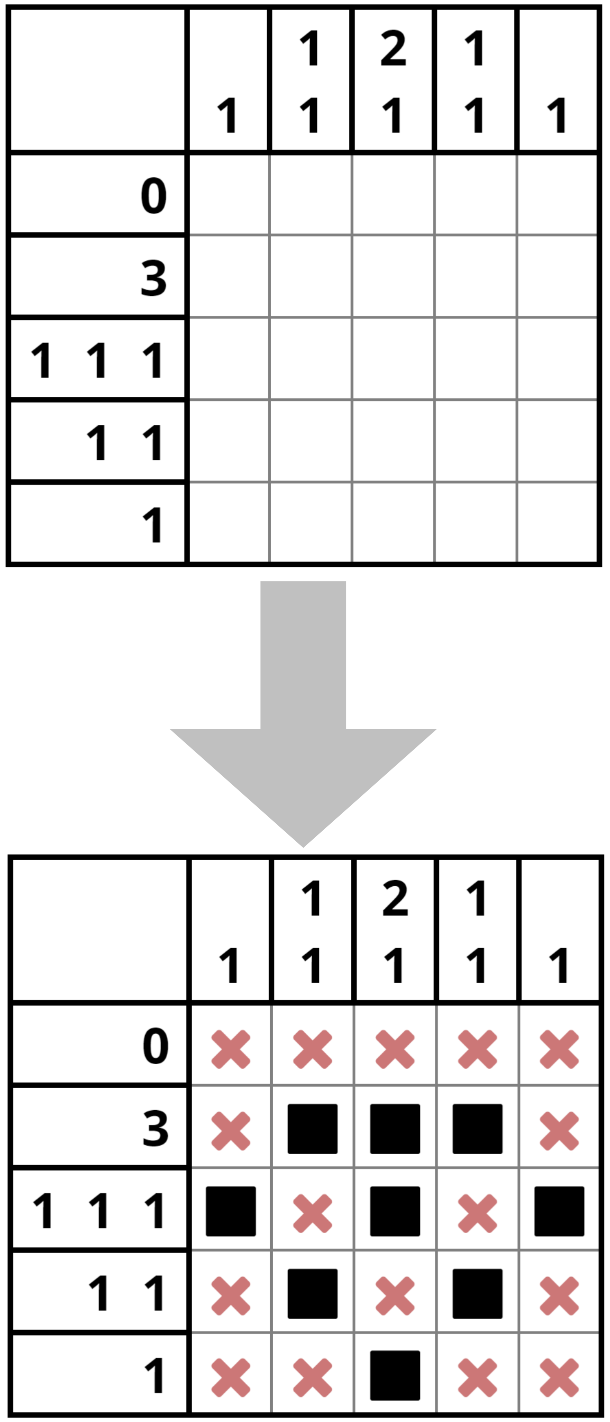
bug report

[Pernsteiner et al., CAV'16]

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



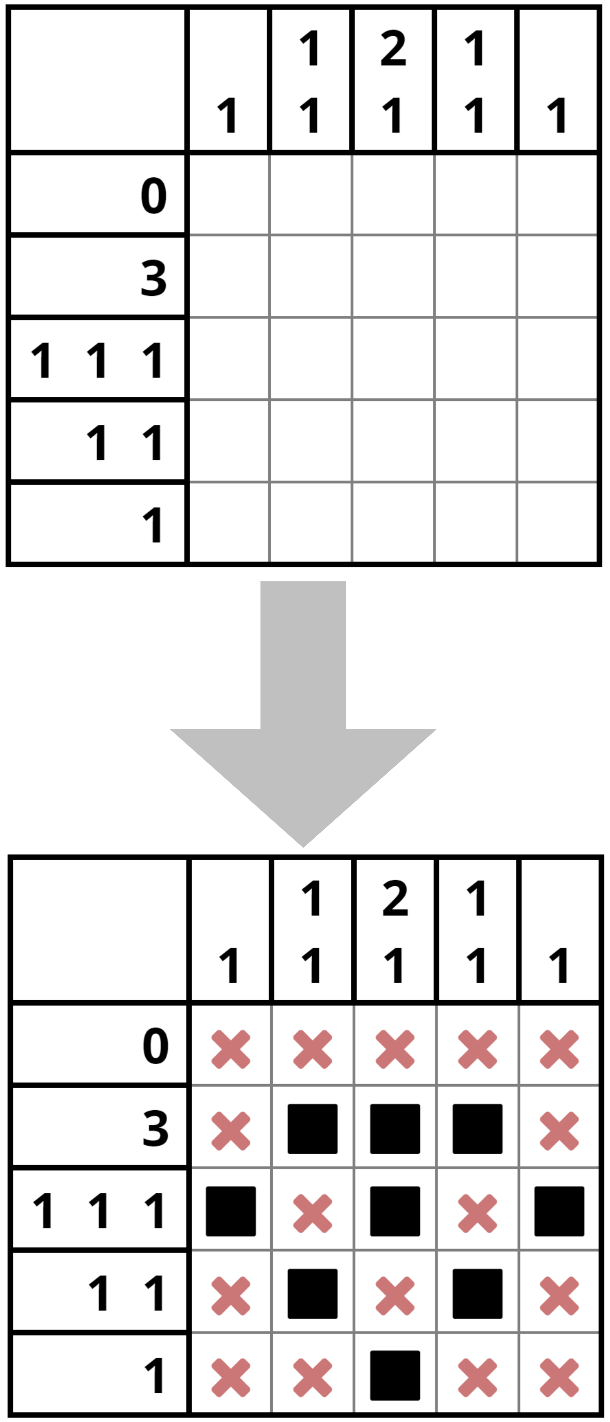
	1	1 1	2 1	1 1	1
0					
3					
1 1 1					
1 1					
1					

	1	1 1	2 1	1 1	1
0	✗	✗	✗	✗	✗
3	✗	■	■	■	✗
1 1 1	■	✗	■	✗	■
1 1	✗	■	✗	■	✗
1	✗	✗	■	✗	✗

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.

Synthesizing strategies for games and education



	1	1 1	2 1	1 1	1
0					
3					
1 1 1					
1 1					
1					

	1	1 1	2 1	1 1	1
0	✗	✗	✗	✗	✗
3	✗	■	■	■	✗
1 1 1	■	✗	■	✗	■
1 1	✗	■	✗	■	✗
1	✗	✗	■	✗	✗

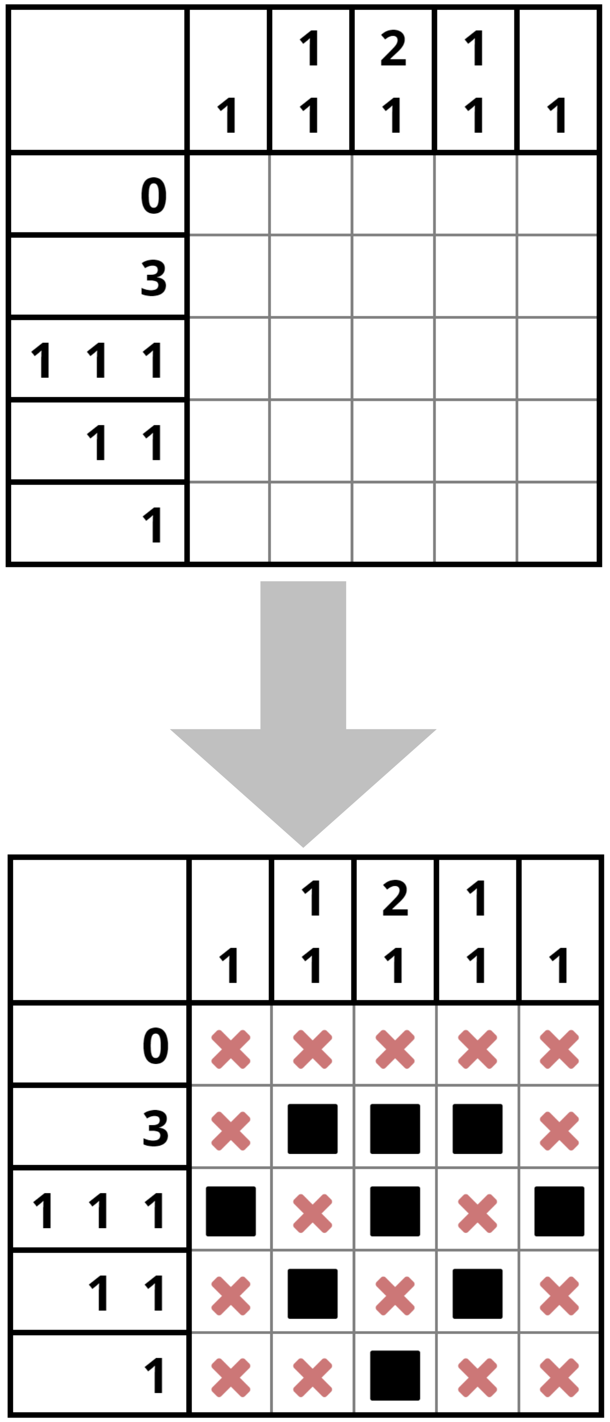
Nonograms game mechanics:

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.

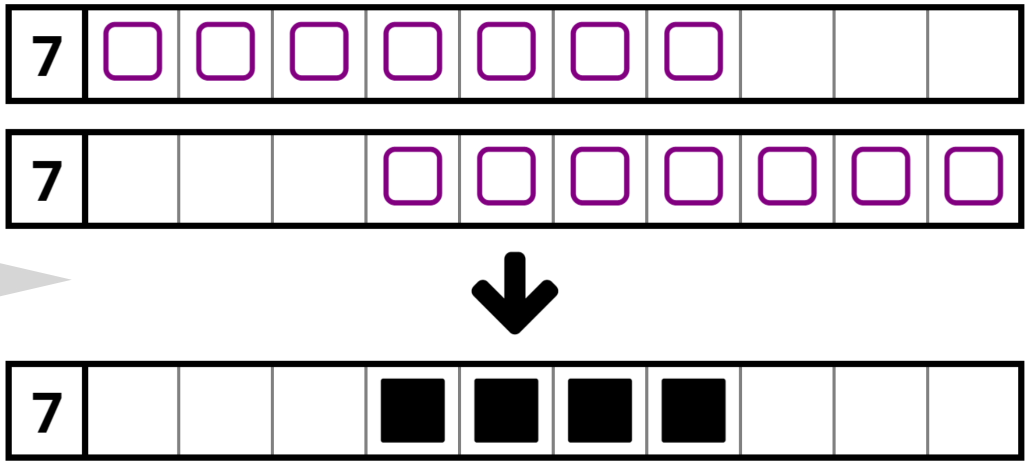
Synthesizing strategies for games and education



	1	1 1	2 1	1 1	1
0					
3					
1 1 1					
1 1					
1					

	1	1 1	2 1	1 1	1
0	✕	✕	✕	✕	✕
3	✕	■	■	■	✕
1 1 1	■	✕	■	✕	■
1 1	✕	■	✕	■	✕
1	✕	✕	■	✕	✕

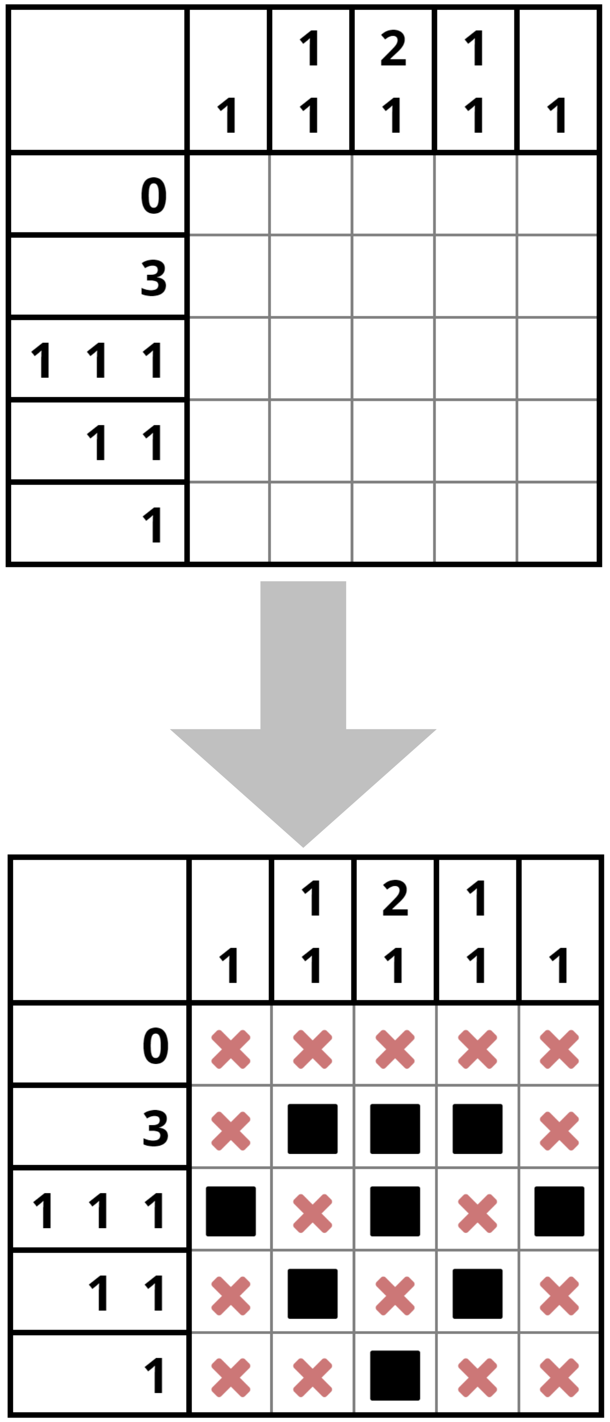
The 'big hint" strategy.



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.

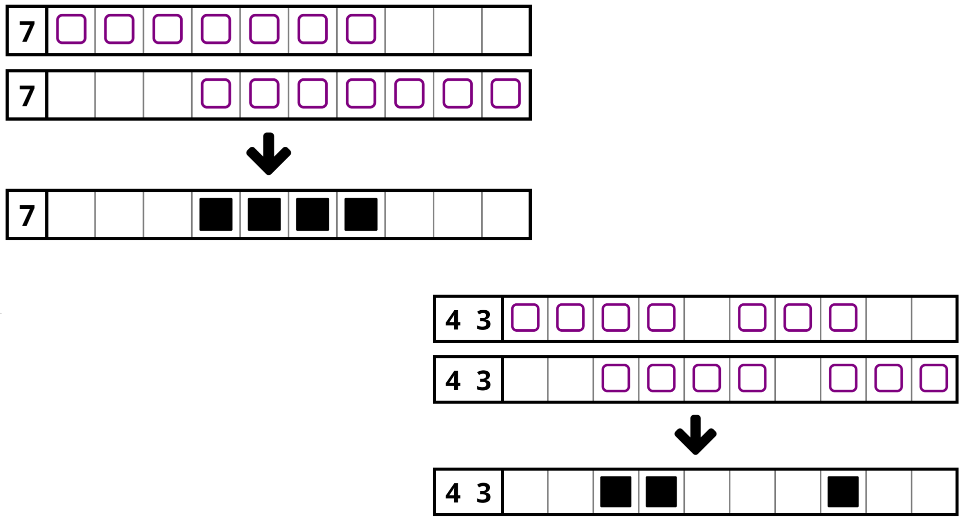
Synthesizing strategies for games and education



	1	1 1	2 1	1 1	1
0					
3					
1 1 1					
1 1					
1					

	1	1 1	2 1	1 1	1
0	✕	✕	✕	✕	✕
3	✕	■	■	■	✕
1 1 1	■	✕	■	✕	■
1 1	✕	■	✕	■	✕
1	✕	✕	■	✕	✕

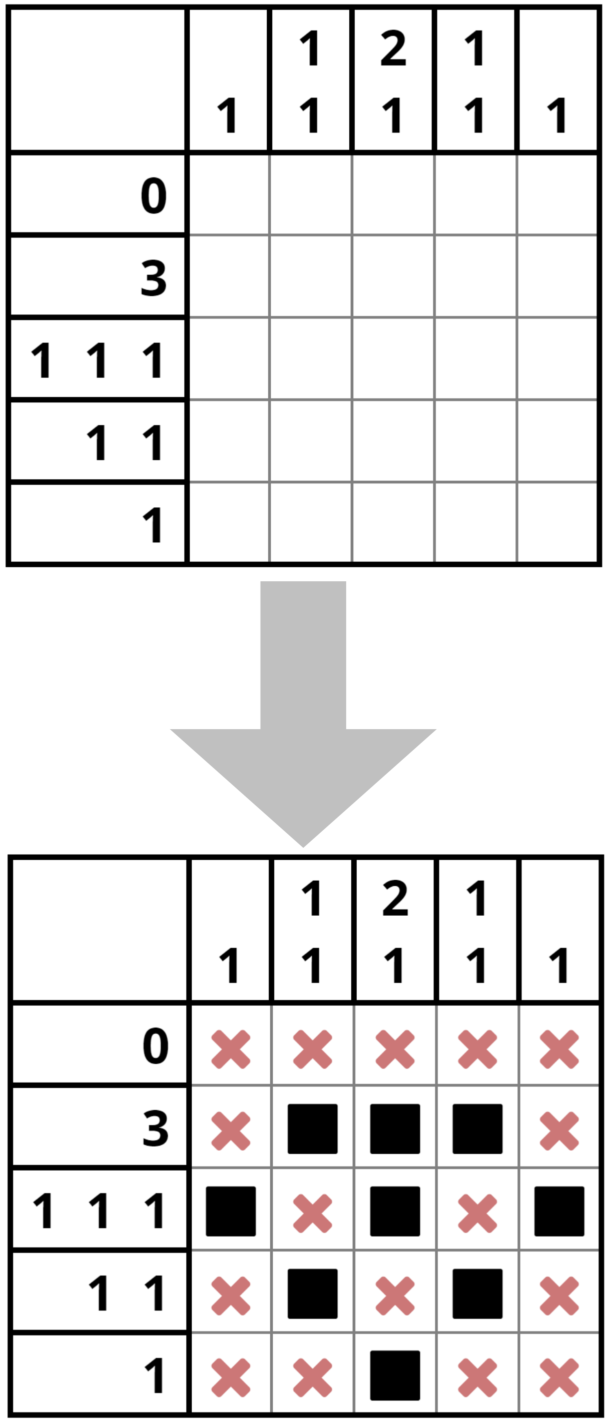
The 'big hint" strategy.



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.

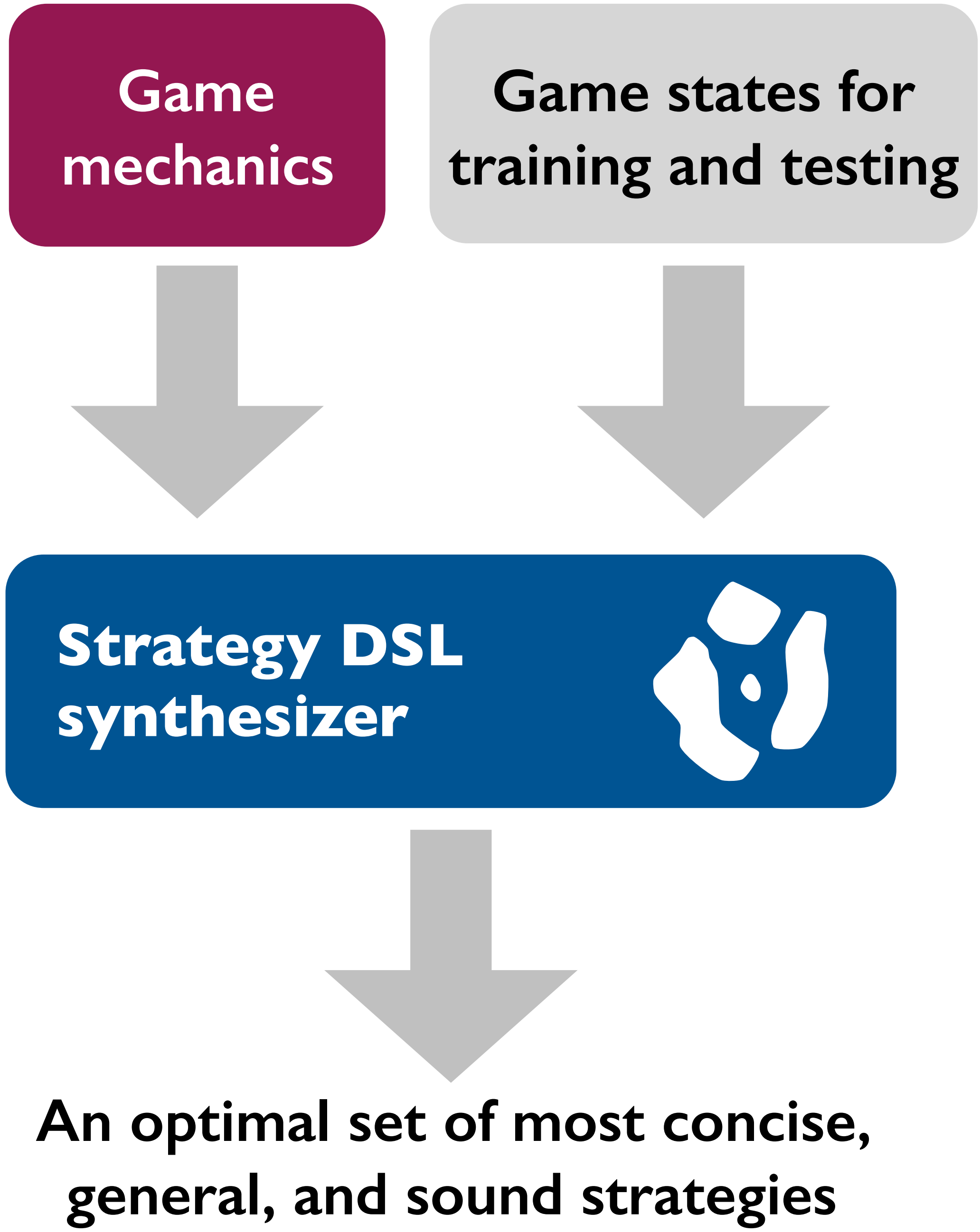
Finding these strategies is a key challenge in game design, and is usually done through human testing.

Synthesizing strategies for games and education



	1	1 1	2 1	1 1	1
0					
3					
1 1 1					
1 1					
1					

	1	1 1	2 1	1 1	1
0	✗	✗	✗	✗	✗
3	✗	■	■	■	✗
1 1 1	■	✗	■	✗	■
1 1	✗	■	✗	■	✗
1	✗	✗	■	✗	✗



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

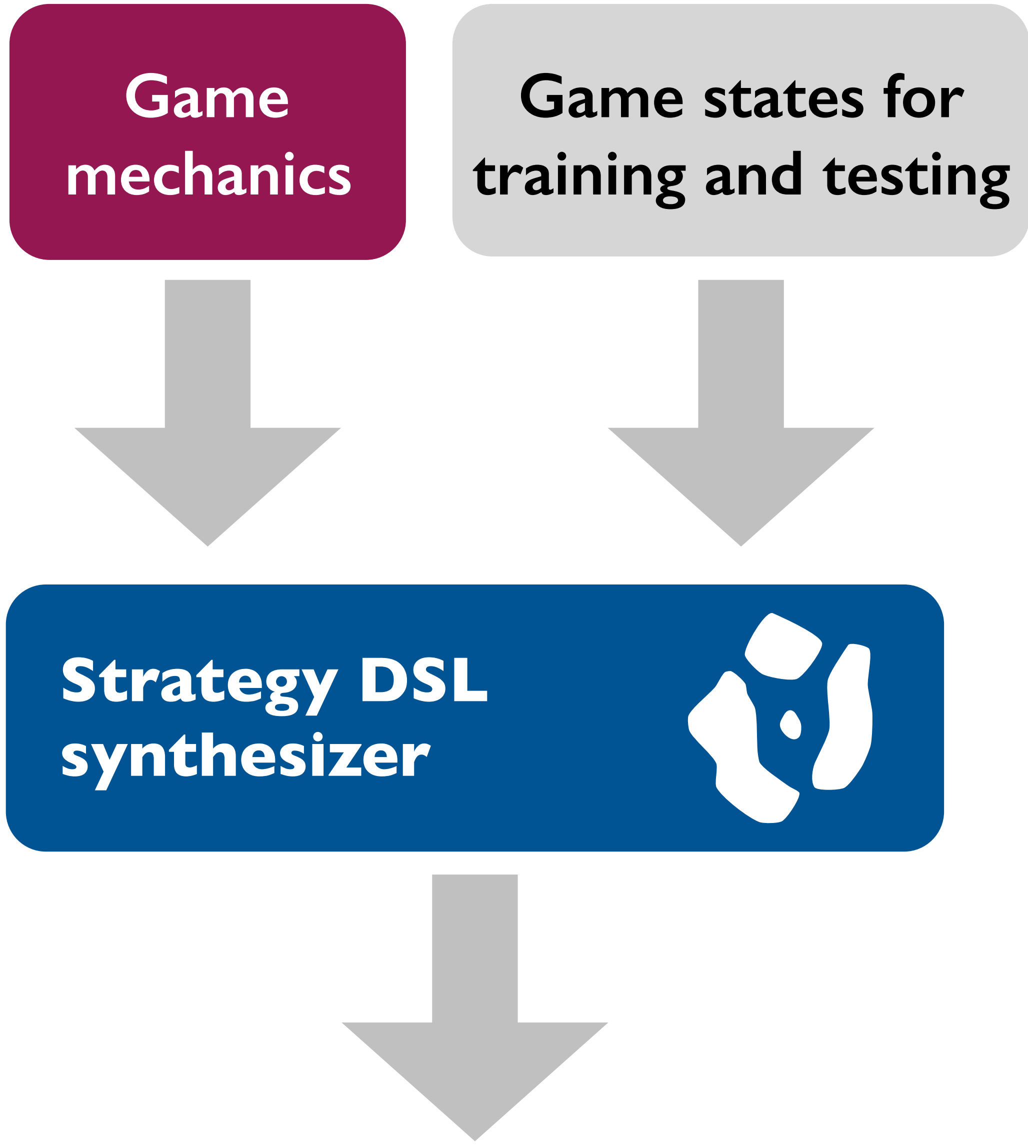
Strategy DSL synthesizer

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

Prototyped in a few weeks and synthesized real strategies.

Eric Butler

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

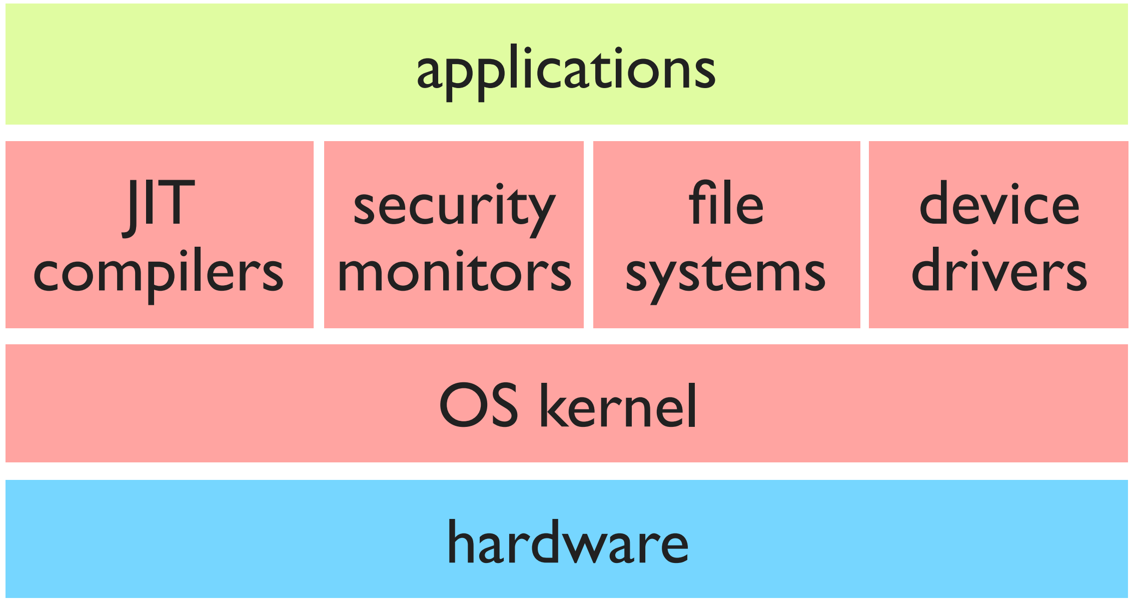
[Butler et al., FDG'17, VMCAI'18]

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.



applications

JIT compilers

security monitors

file systems

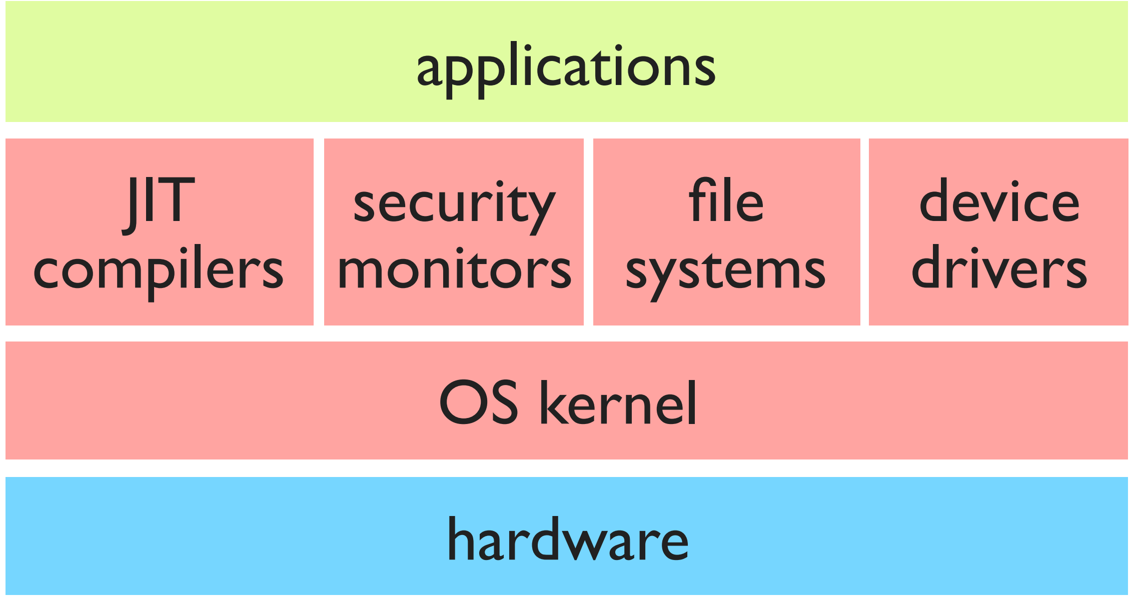
device drivers

OS kernel

hardware

Verifying systems software

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



applications

JIT compilers

security monitors

file systems

device drivers

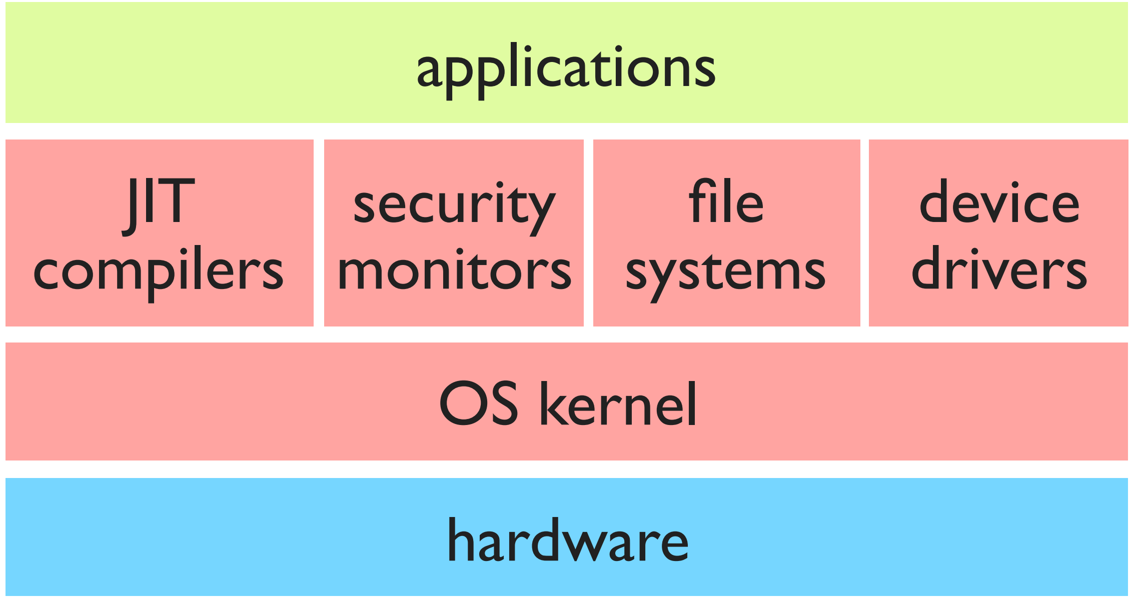
OS kernel

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.

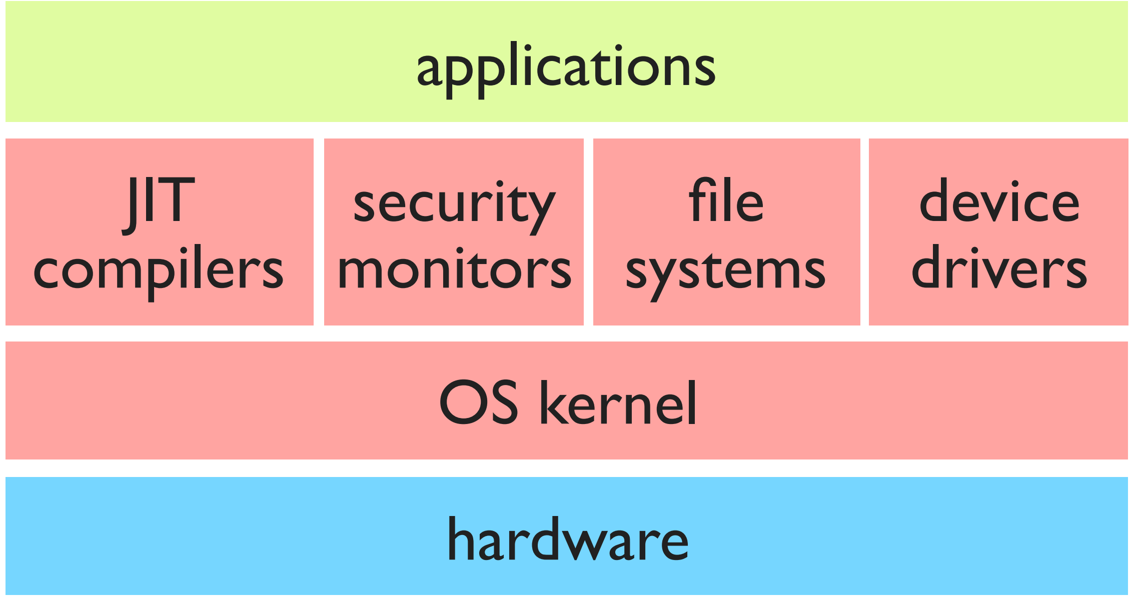


Verifying OS components is hard: e.g., the Komodo security monitor took 2 person-years to prove, with a proof-to-implementation ratio of 6:1.

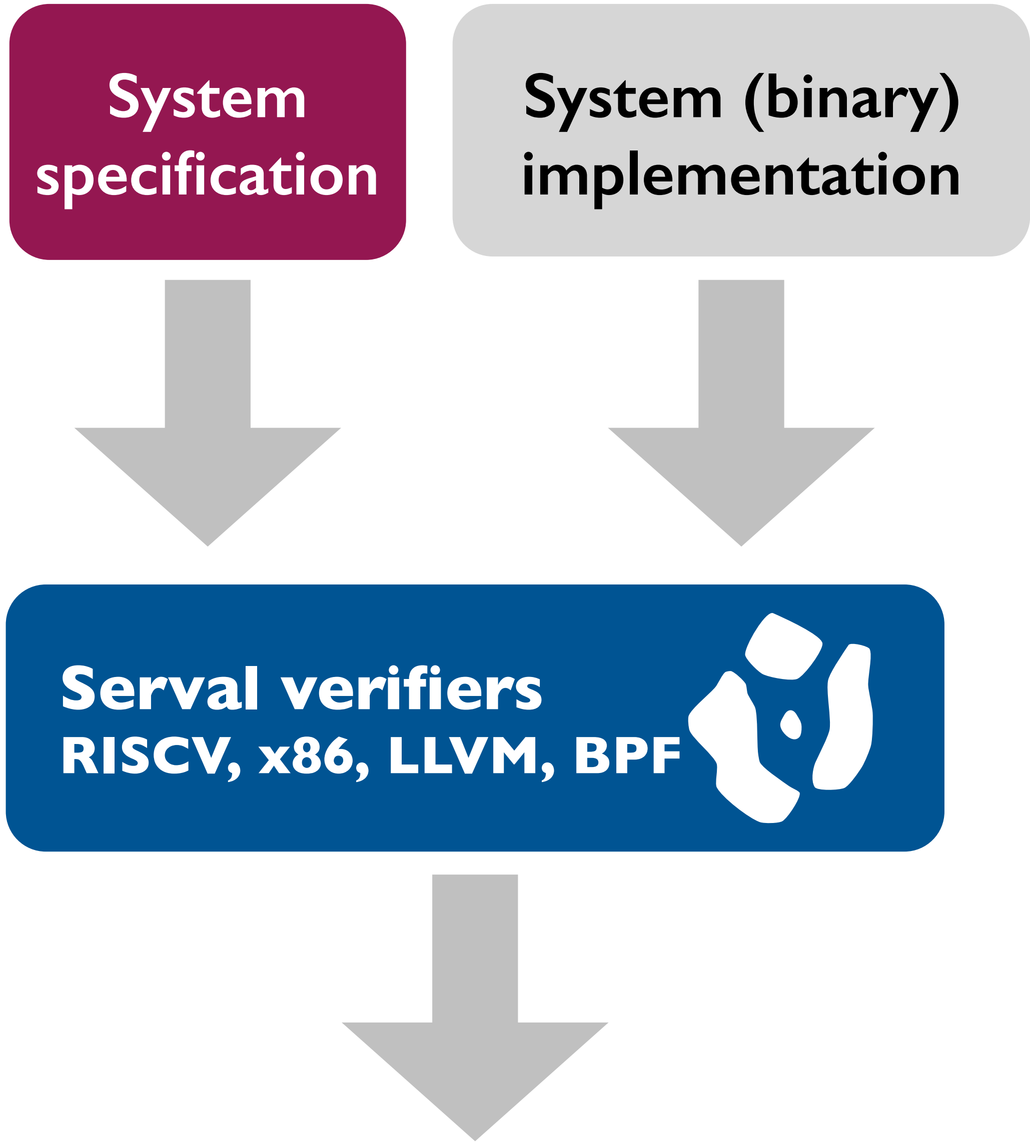
Bugs in OS components are bad news for reliability, security, and performance of computer systems.

Verifying systems software

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

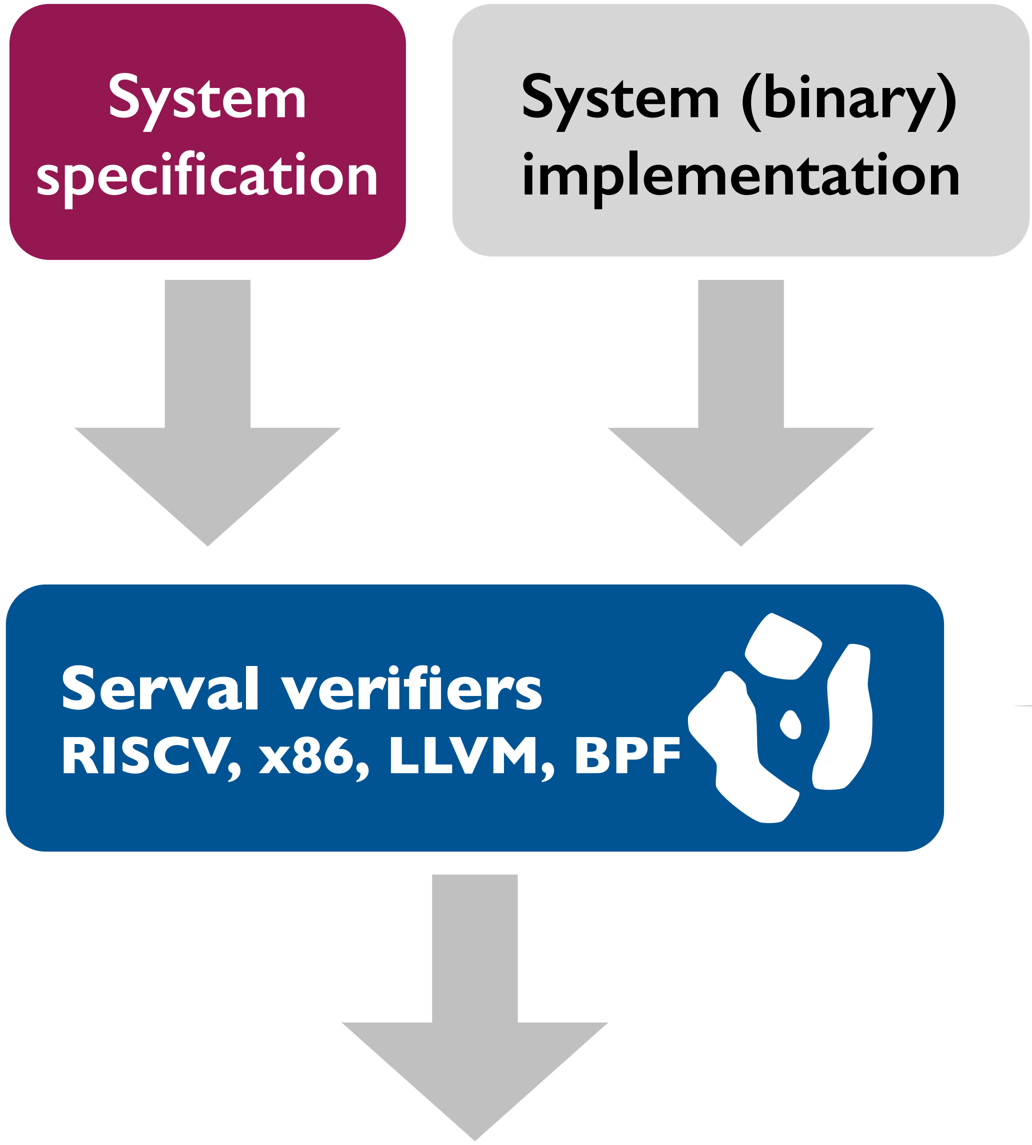


Bugs in OS components are bad news for reliability, security, and performance of computer systems.



Bug report or correctness guarantee

Verifying systems software



System specification

System (binary) implementation

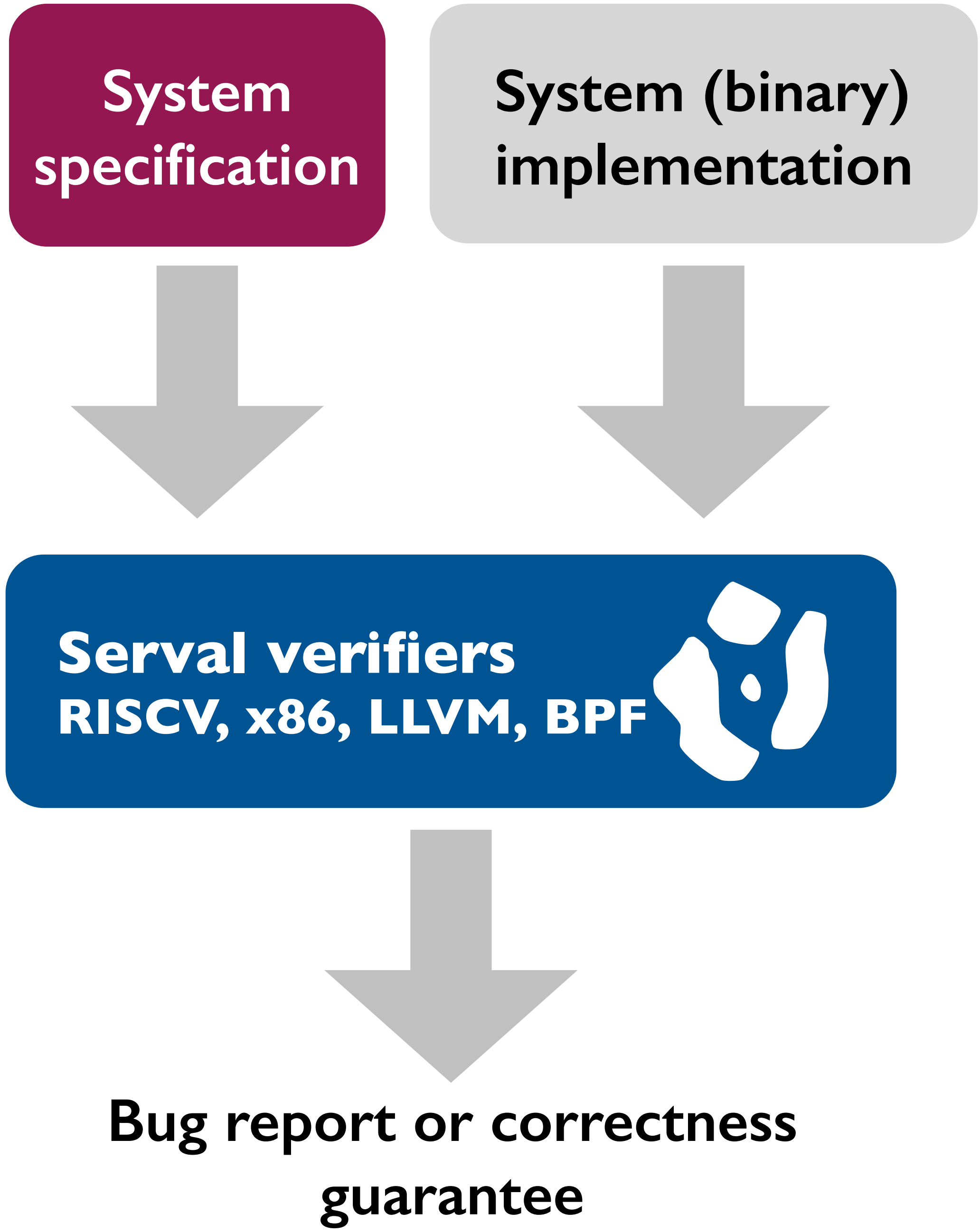
Serval verifiers
RISCV, x86, LLVM, BPF

Bug report or correctness guarantee

Each verifier took a couple of weeks to build!

Luke Nelson

Verifying systems software



System specification

System (binary) implementation

Serval verifiers
RISCV, x86, LLVM, BPF

Bug report or correctness guarantee

[Nelson et al., SOSP'19]

Verified three existing **security monitors** (CertiKOS, Komodo, Keystone) fully automatically.

Found **15 new bugs** in the **Linux BPF JITs** for RISCV64 and x86-32, all confirmed and fixed by developers.

Summary

Today

- Going pro with solver-aided programming.

Next lecture

- Getting started with SAT solving!