Computer-Aided Reasoning for Software

CSE507

Solver-Aided Languages

courses.cs.washington.edu/courses/cse507/14au/

Emina Torlak
emina@cs.washington.edu

Today

Today

Last lecture

- Program synthesis

Today

Last lecture

- Program synthesis

Today

- The next N years: Solver-Aided Languages (?)

Today

Last lecture

- Program synthesis

Today

- The next N years: Solver-Aided Languages (?)

Reminders

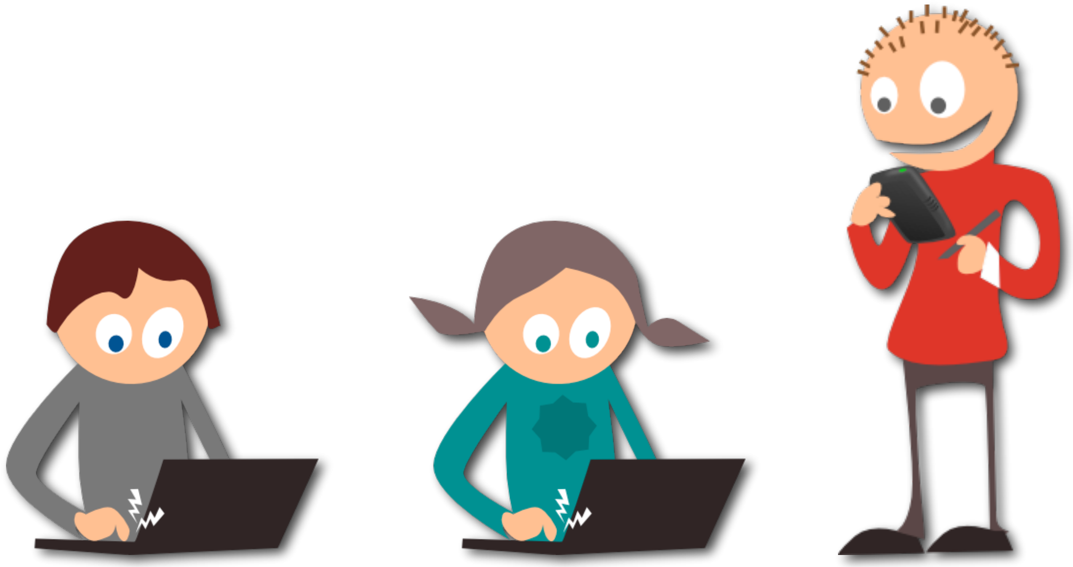
- Please fill out the course evaluation form (Dec 02-08)
- 8 min final presentations on Monday, Dec 08, 10:30am, MGH 254
- Final projects due on Monday, Dec 08, at 11pm

vision

a little programming for everyone

A little programming for everyone

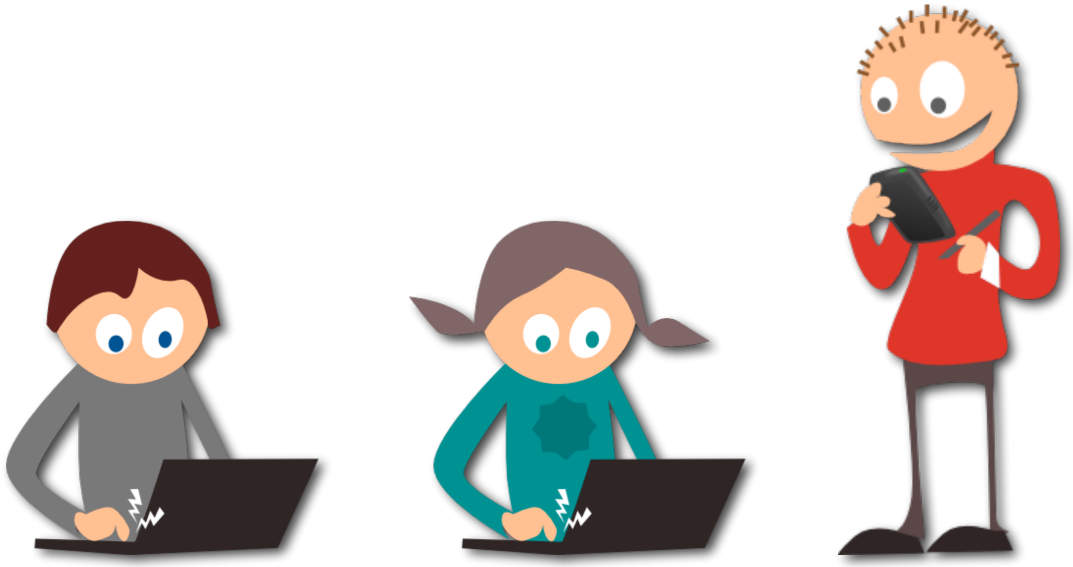
We all want to build programs …



A little programming for everyone

We all want to build programs …

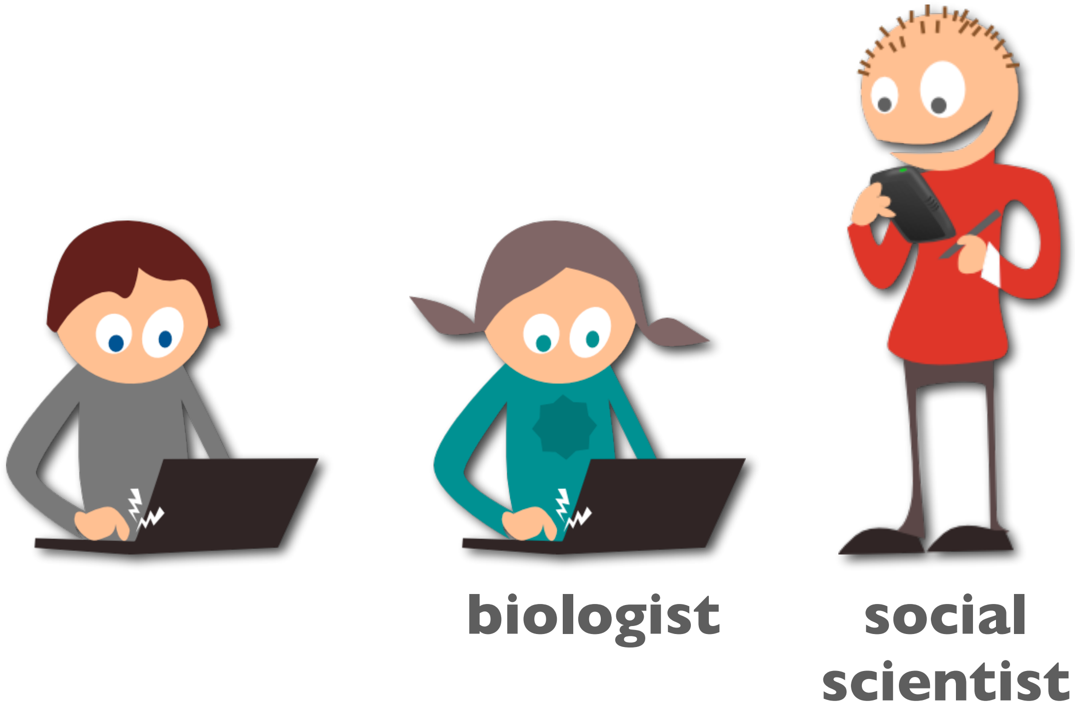
- spreadsheet data manipulation



A little programming for everyone

We all want to build programs …

- spreadsheet data manipulation
- models of cell fates



A little programming for everyone

We all want to build programs …

- spreadsheet data manipulation
- models of cell fates
- cache coherence protocols
- memory models

hardware designer

biologist

social scientist

A little programming for everyone

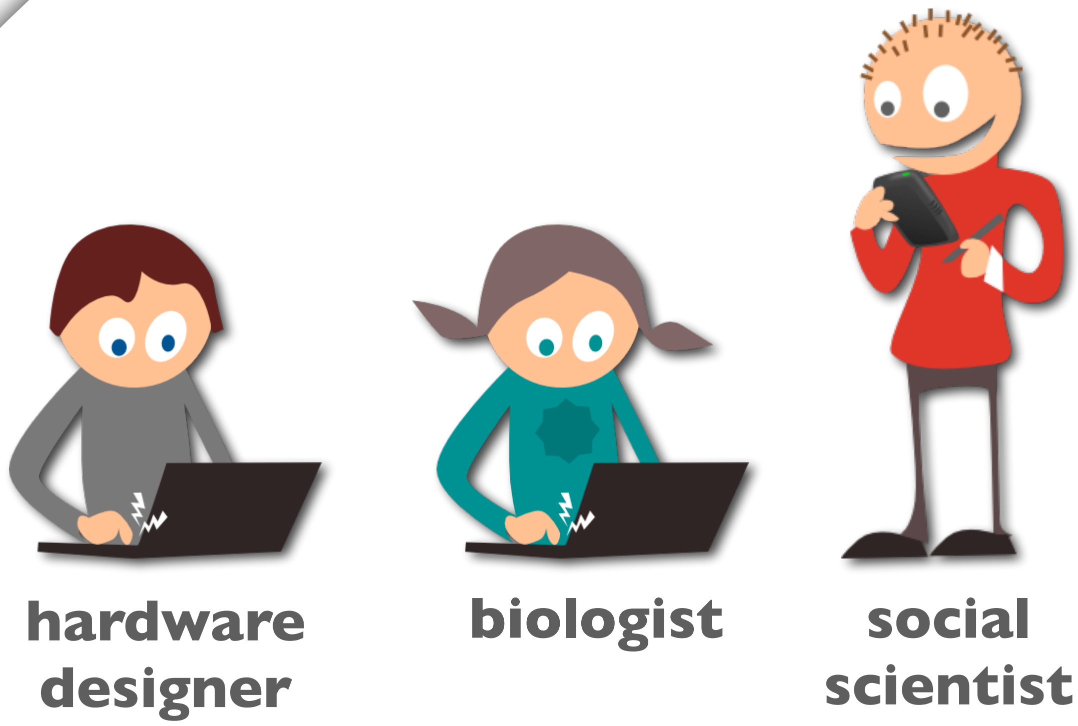
We all want to build programs …

- spreadsheet data manipulation [Flashfill, **POPL'11**]
- models of cell fates [SBL, **POPL'13**]
- cache coherence protocols [Transit, **PLDI'13**]
- memory models [MemSAT, **PLDI'10**]

time

code

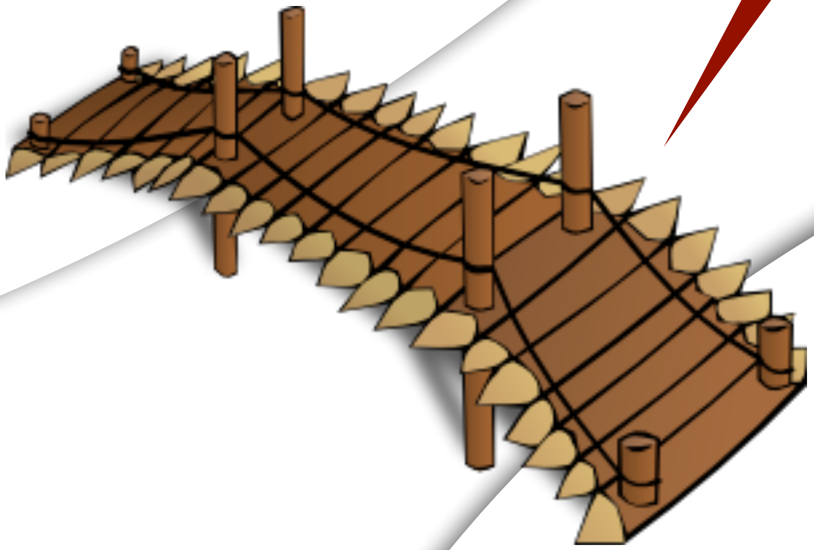
effort



A little programming for everyone

We all want to build programs …

- spreadsheet data manipulation
- models of cell fates
- cache coherence protocols
- memory models

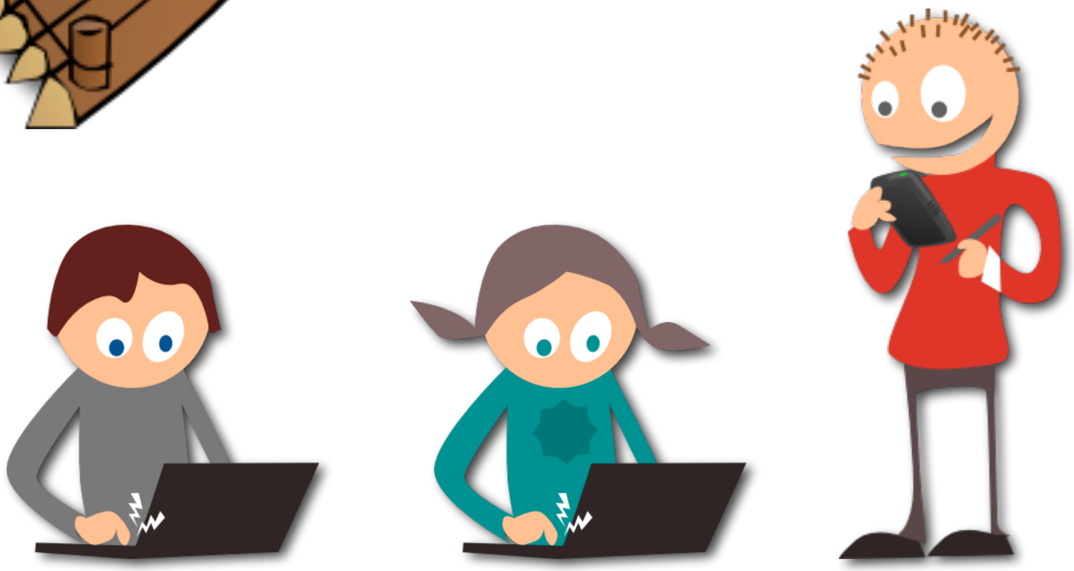


solver-aided languages

less time

less code

less effort



hardware designer

biologist

social scientist

A little history

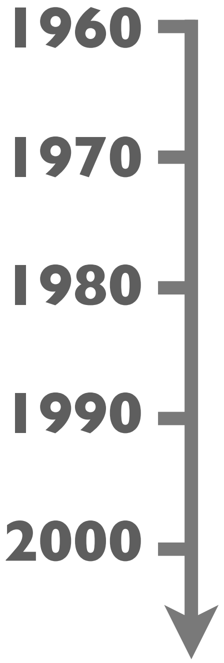
program logics (Floyd, Hoare, Dijkstra)

mechanization of logic (Milner, Pnueli)

mechanized tools (Clarke, Emerson, Sifakis)

better programs

A little history



software crisis

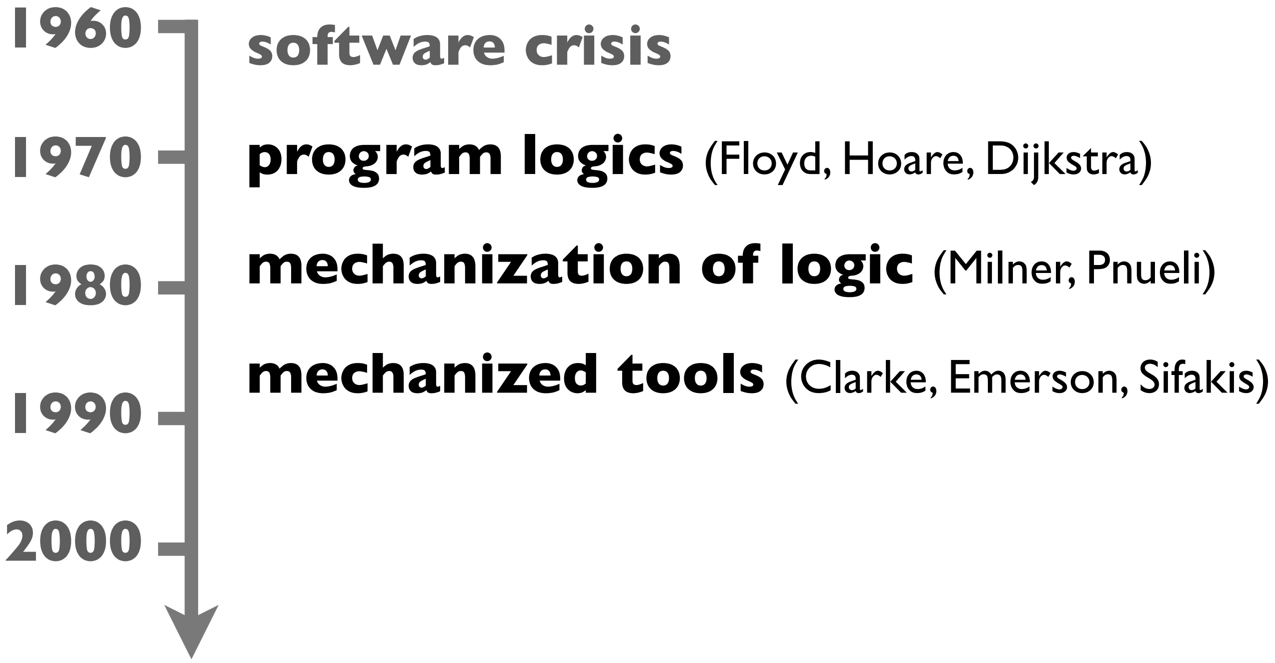
program logics (Floyd, Hoare, Dijkstra)

mechanization of logic (Milner, Pnueli)

mechanized tools (Clarke, Emerson, Sifakis)

better programs

A little history



1960 software crisis

1970 **program logics** (Floyd, Hoare, Dijkstra)

1980 **mechanization of logic** (Milner, Pnueli)

1990 **mechanized tools** (Clarke, Emerson, Sifakis)

2000

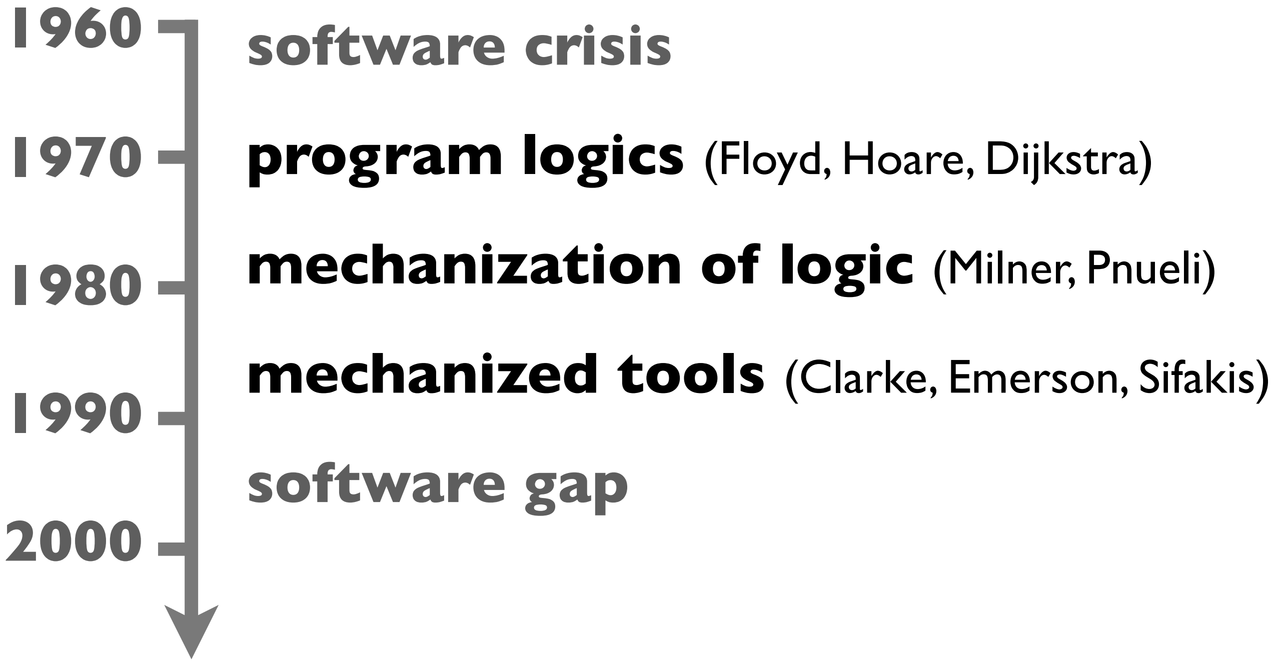
better programs

6TH SENSE [IBM]

ASTRÉE [AbsInt]

SLAM [MSR]

A little history



1960 software crisis

1970 **program logics** (Floyd, Hoare, Dijkstra)

1980 **mechanization of logic** (Milner, Pnueli)

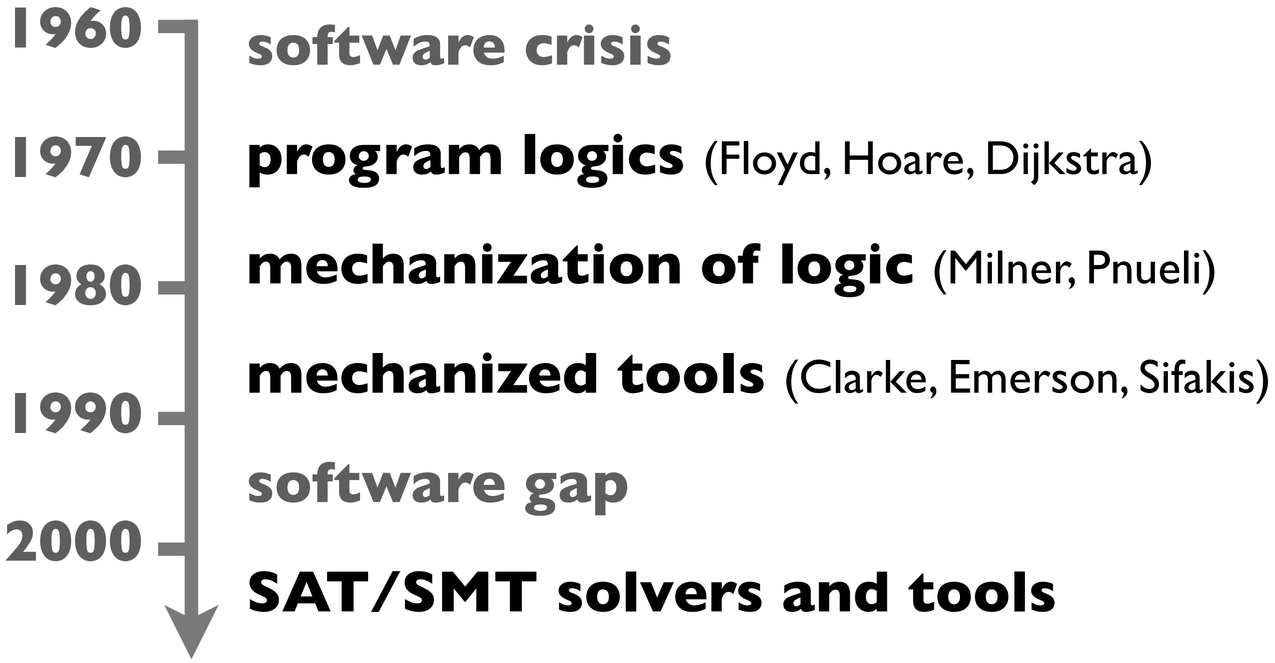
mechanized tools (Clarke, Emerson, Sifakis)

1990 software gap

2000

better programs

A little history



1960 **software crisis**

1970 **program logics** (Floyd, Hoare, Dijkstra)

1980 **mechanization of logic** (Milner, Pnueli)

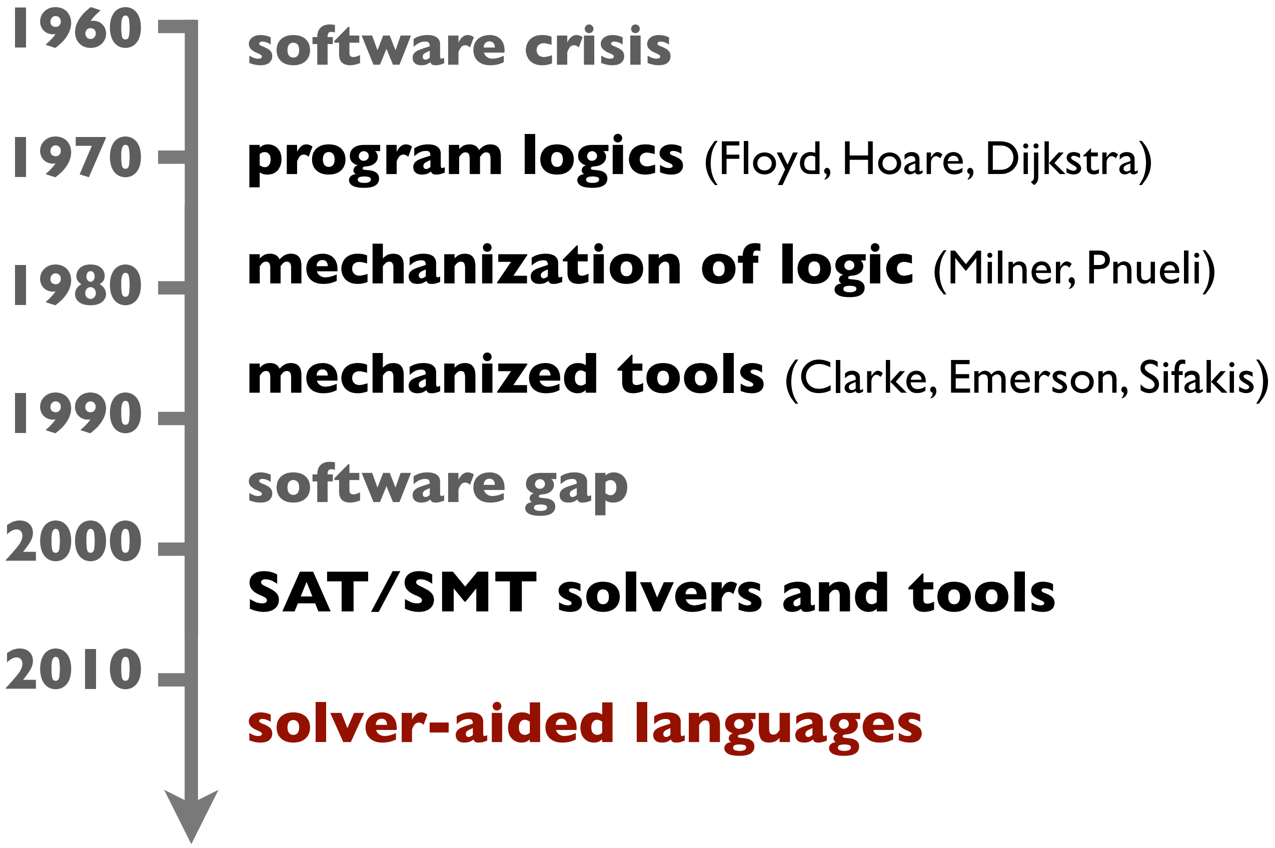
mechanized tools (Clarke, Emerson, Sifakis)

1990 **software gap**

2000 **SAT/SMT solvers and tools**

better programs

A little history



1960

software crisis

1970

program logics (Floyd, Hoare, Dijkstra)

1980

mechanization of logic (Milner, Pnueli)

mechanized tools (Clarke, Emerson, Sifakis)

1990

software gap

2000

SAT/SMT solvers and tools

2010

solver-aided languages

outline

outline

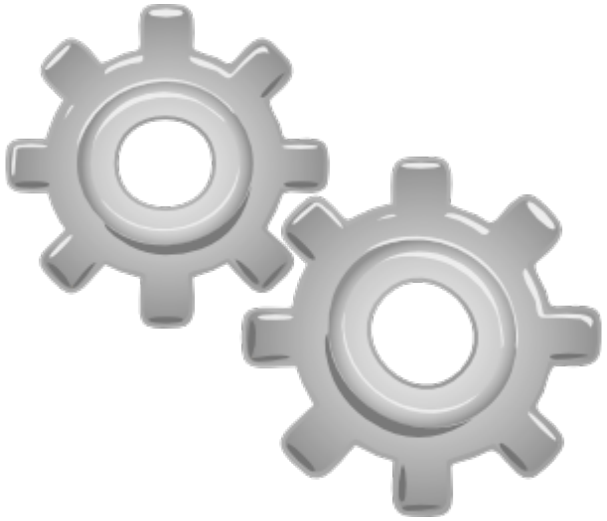
solver-aided tools

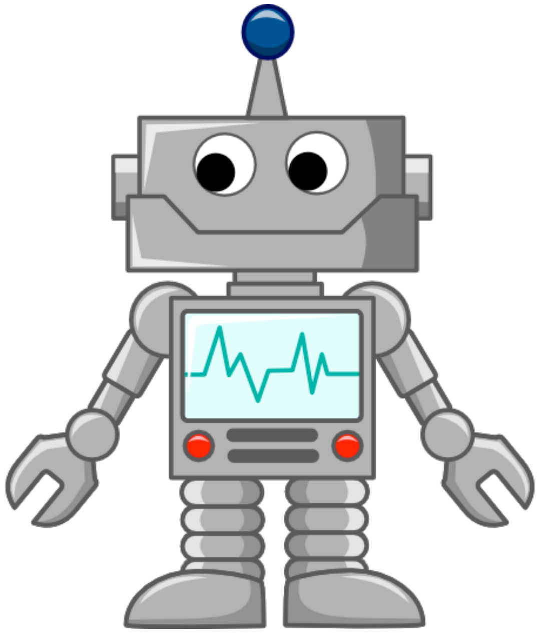
outline

solver-aided tools, languages

outline

solver-aided tools, languages and beyond

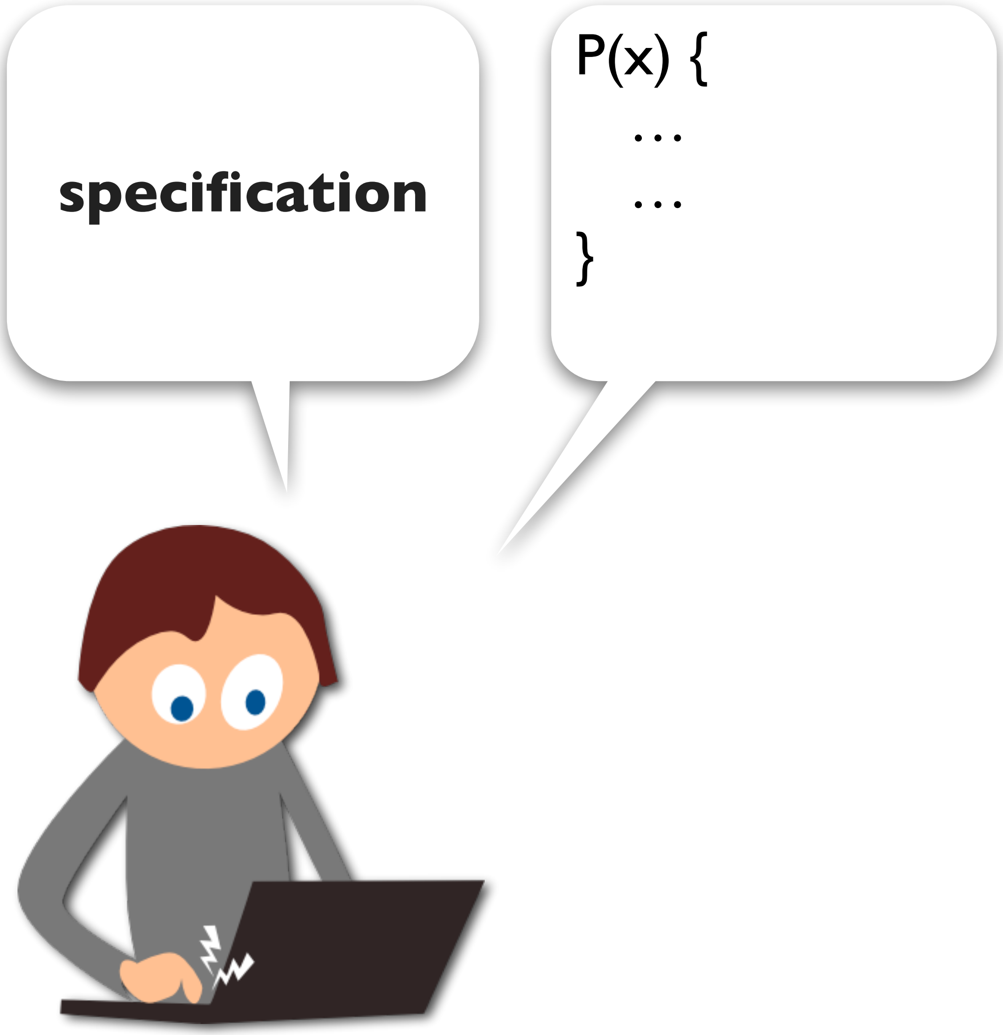




story

solver-aided tools

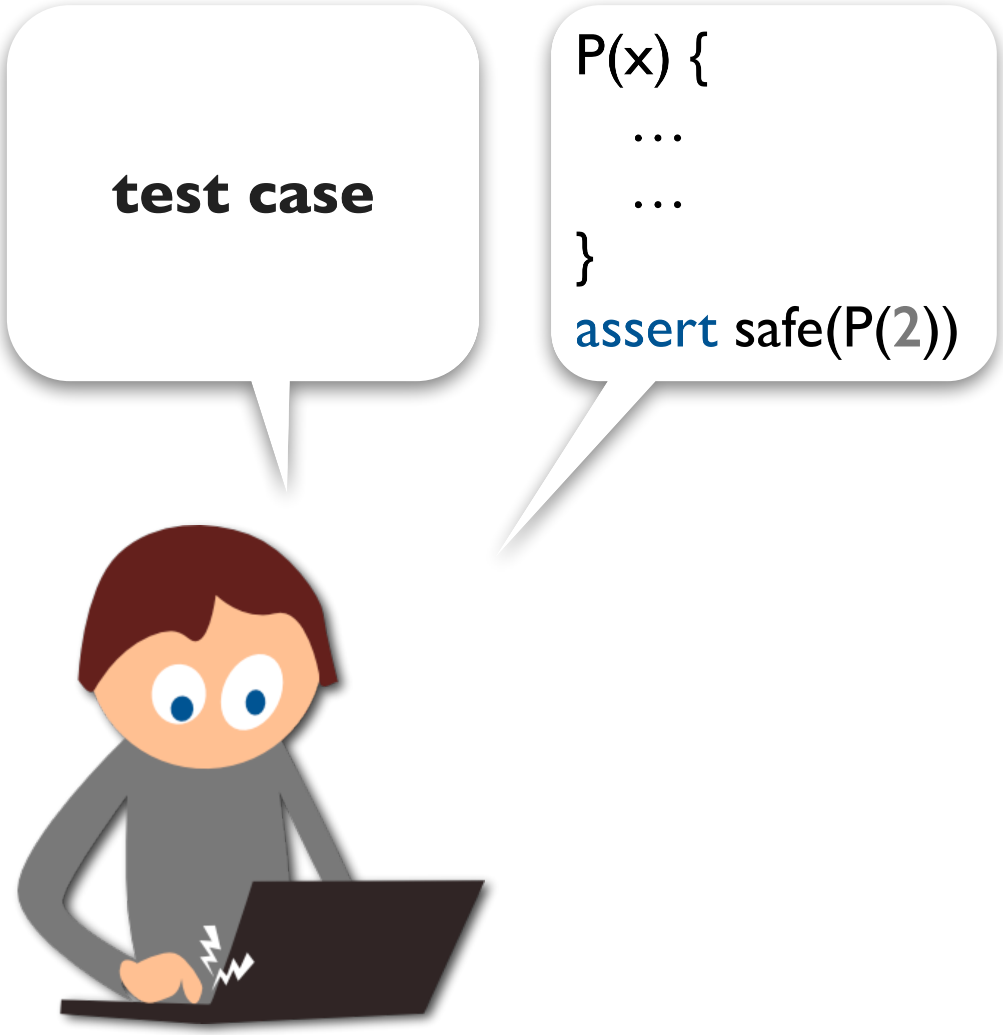
Programming …



specification

```
P(x) {
   …
   …
}
```

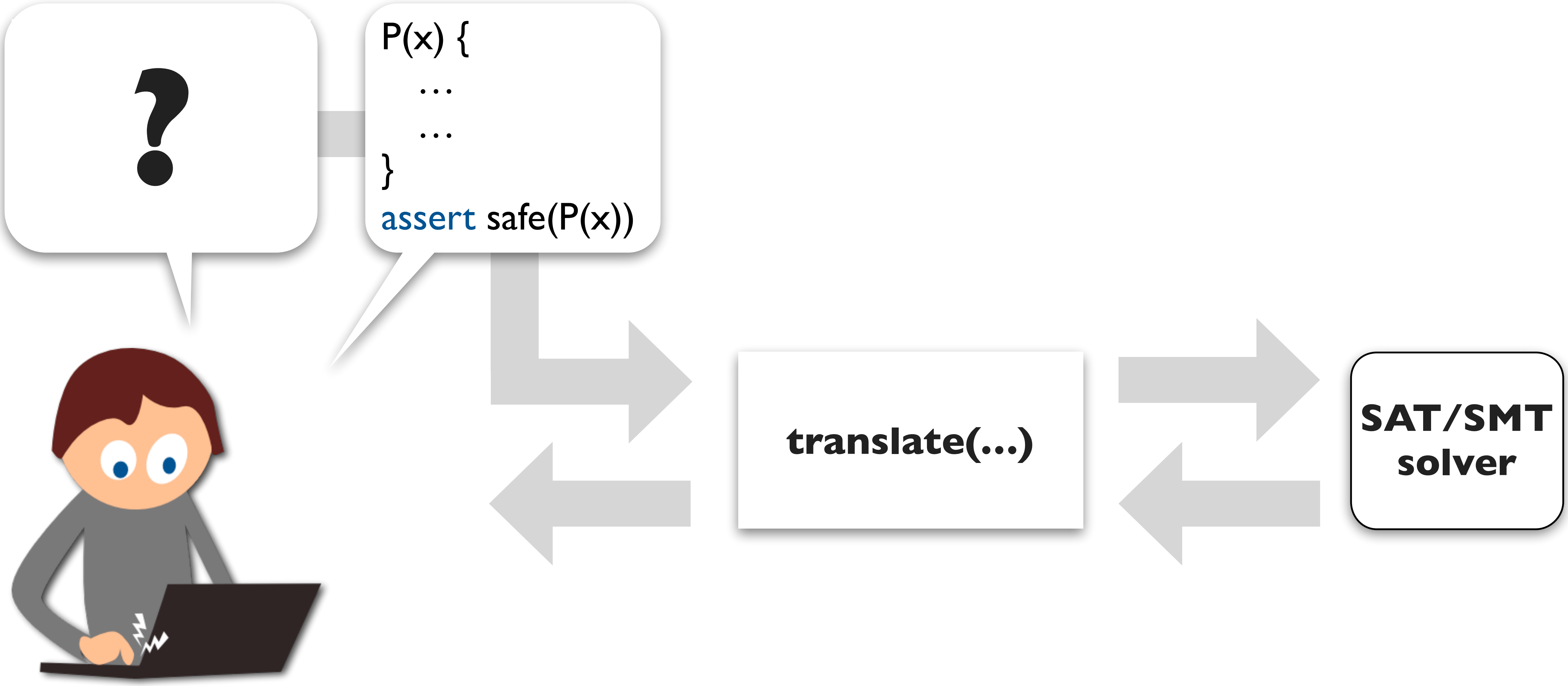
Programming ...



test case

```
P(x) {
    …
    …
}
assert safe(P(2))
```

Programming with a solver-aided tool



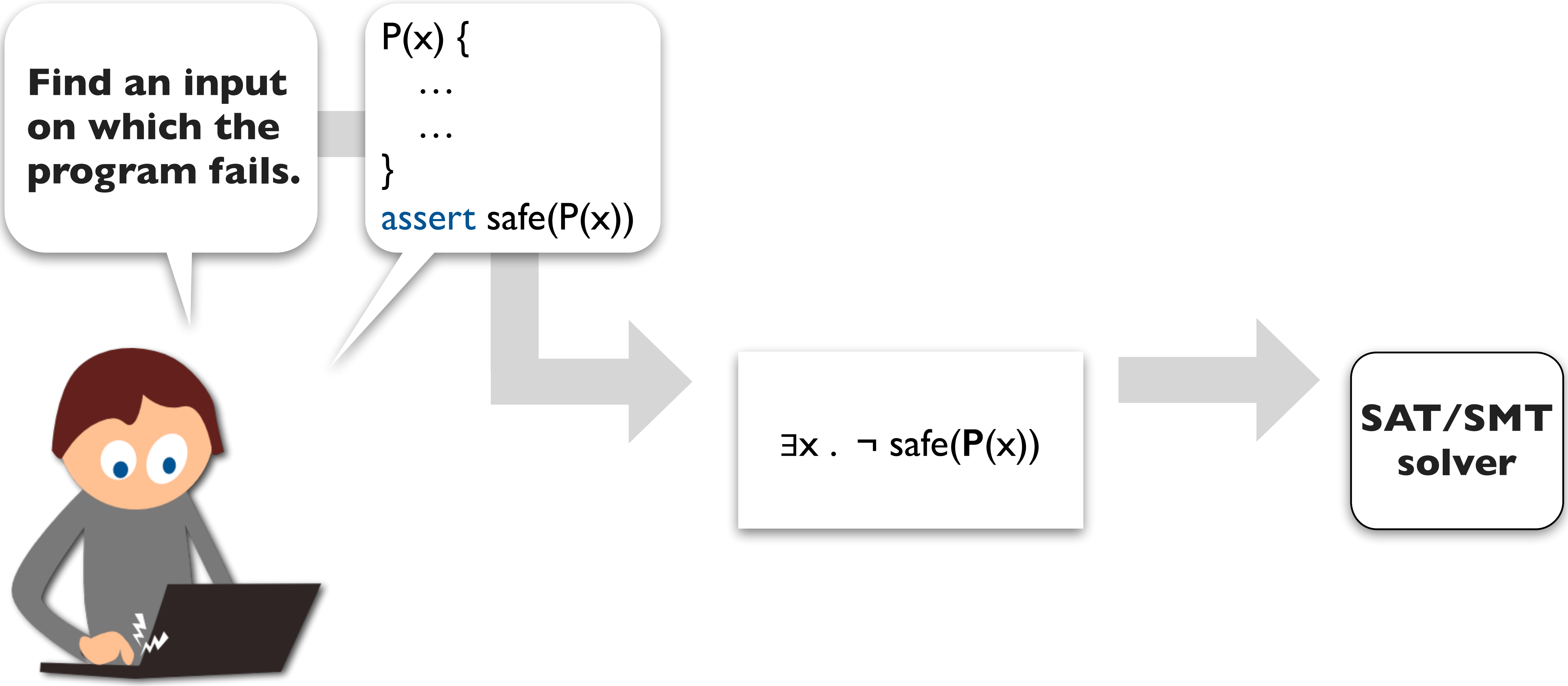
?

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

translate(…)

SAT/SMT solver

Solver-aided tools: verification



Find an input on which the program fails.

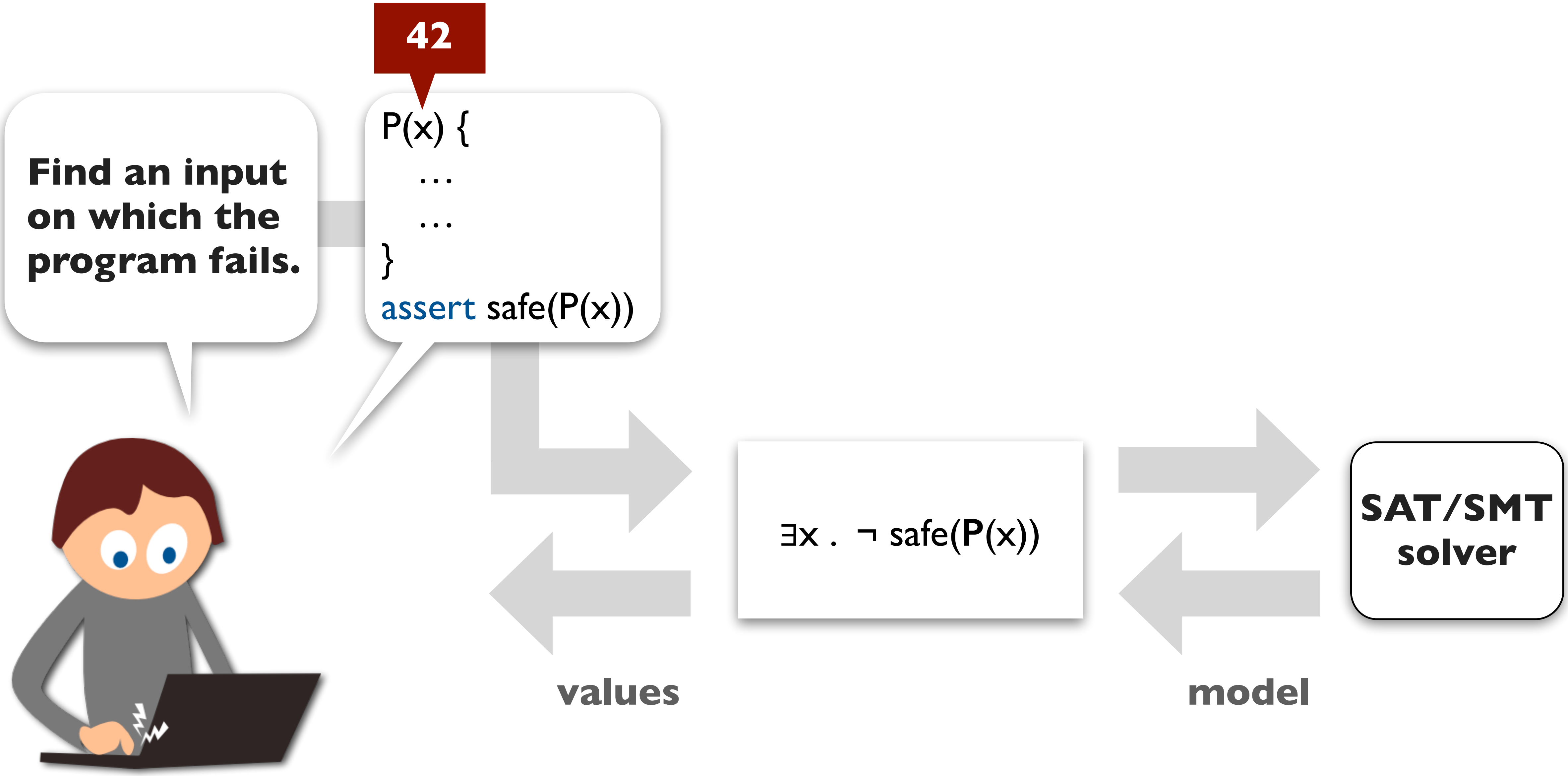
```
P(x) {
   …
   …
}
assert safe(P(x))
```

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

SAT/SMT solver

CBMC [Kroening et al., DAC'03]
Dafny [Leino, LPAR'10]
Miniatur [Vaziri et al., FSE'07]
Klee [Cadar et al., OSDI'08]

Solver-aided tools: verification



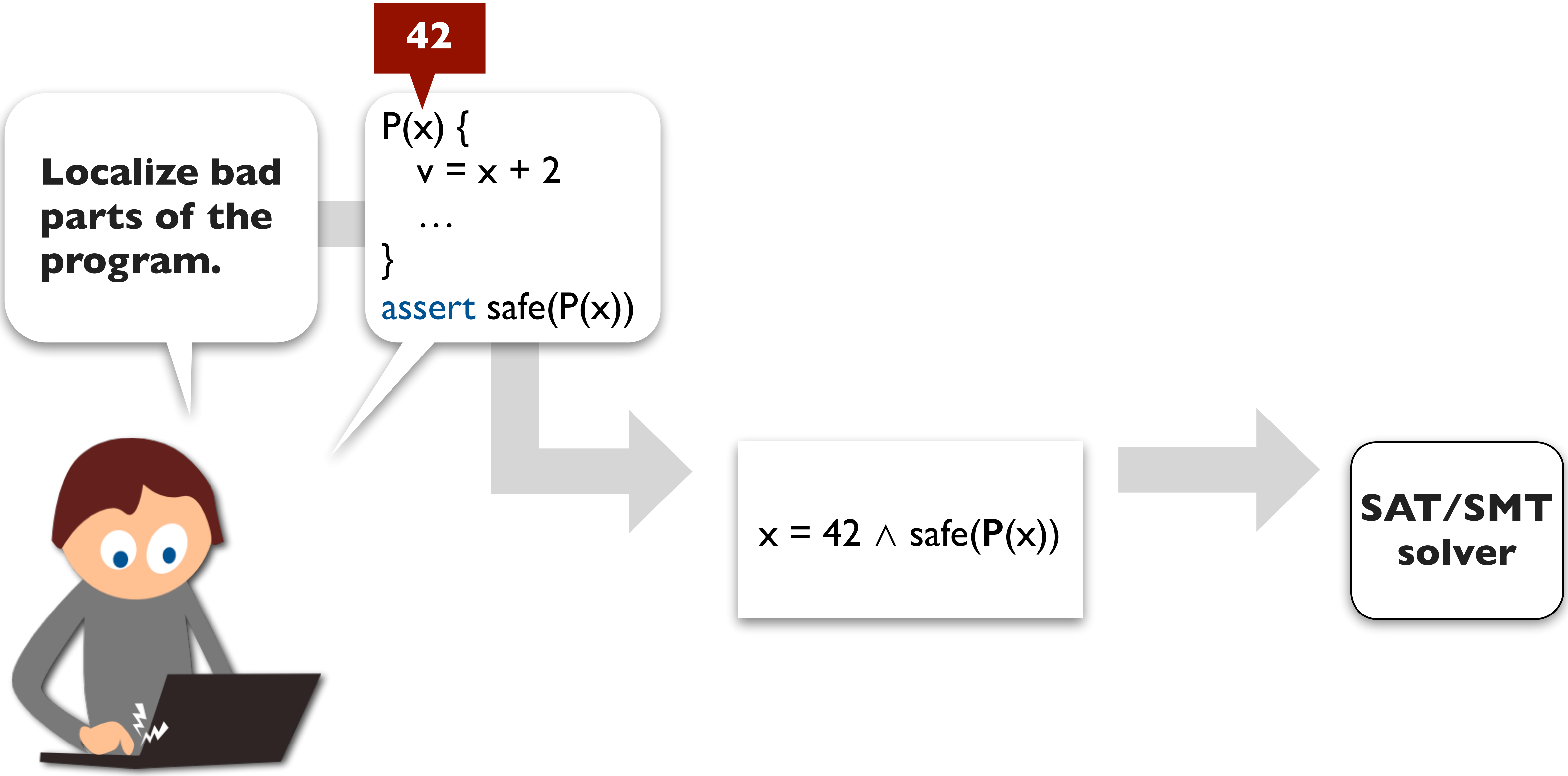
Find an input on which the program fails.

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

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

CBMC [Kroening et al., DAC'03]
Dafny [Leino, LPAR'10]
Miniatur [Vaziri et al., FSE'07]
Klee [Cadar et al., OSDI'08]

Solver-aided tools: debugging



Localize bad parts of the program.

42

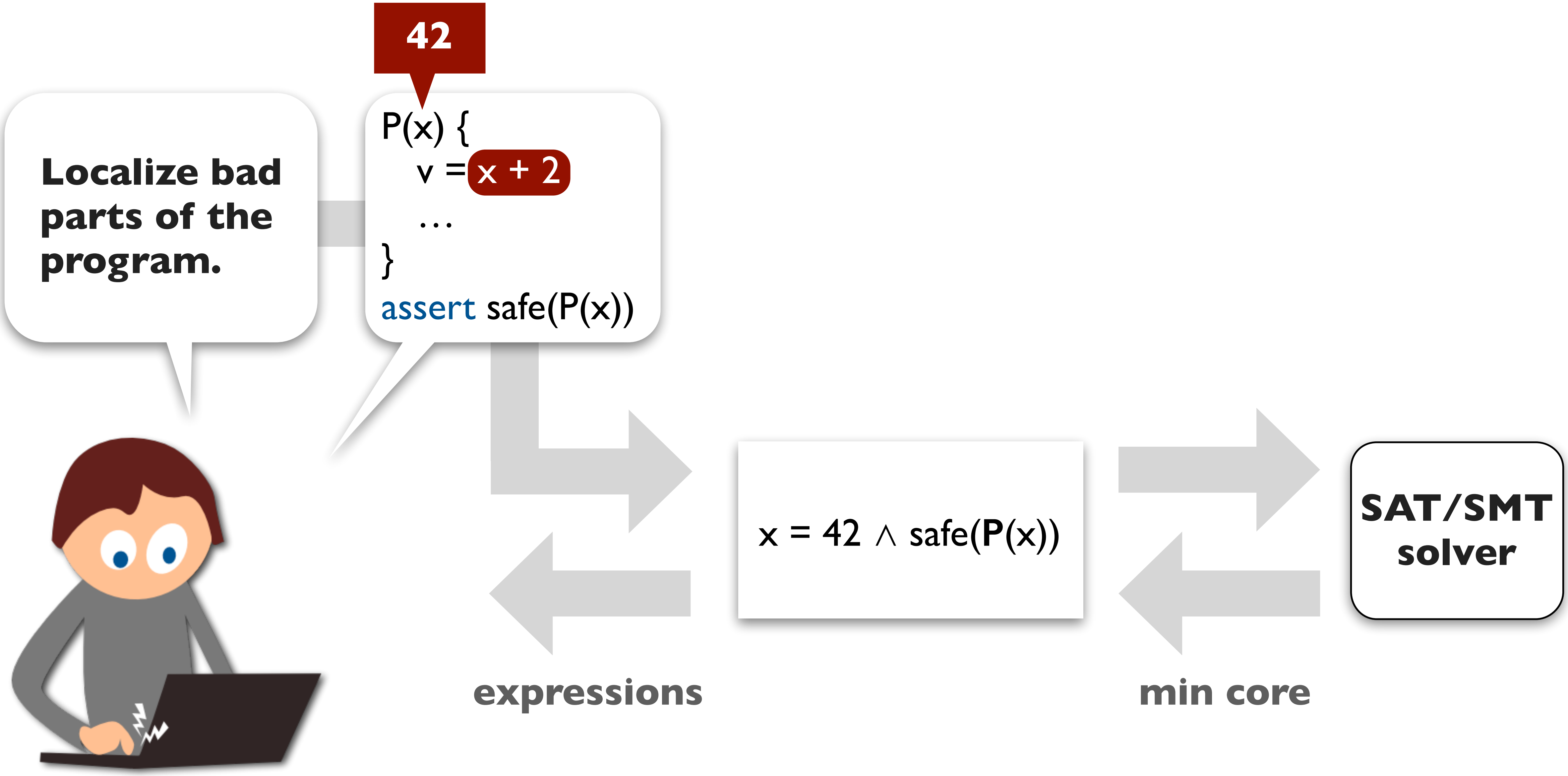
```
P(x) {
  v = x + 2
  …
}
assert safe(P(x))
```

x = 42 ∧ safe(P(x))

SAT/SMT solver

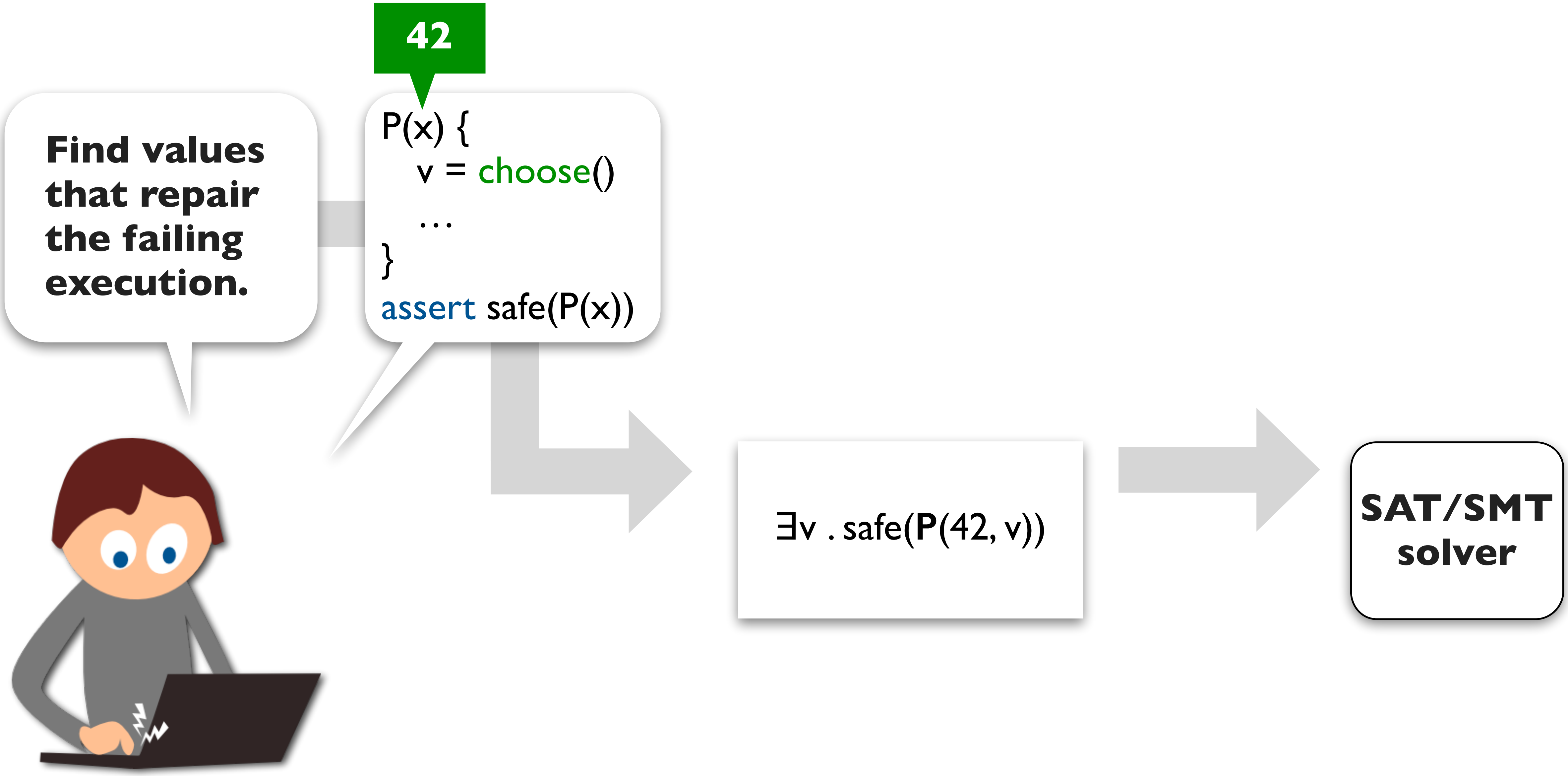
BugAssist [Jose & Majumdar, PLDI’11]
Angelina [Chandra et al., ICSE’11]

Solver-aided tools: debugging



BugAssist [Jose & Majumdar, PLDI'11]
Angelina [Chandra et al., ICSE'11]

Solver-aided tools: angelic execution



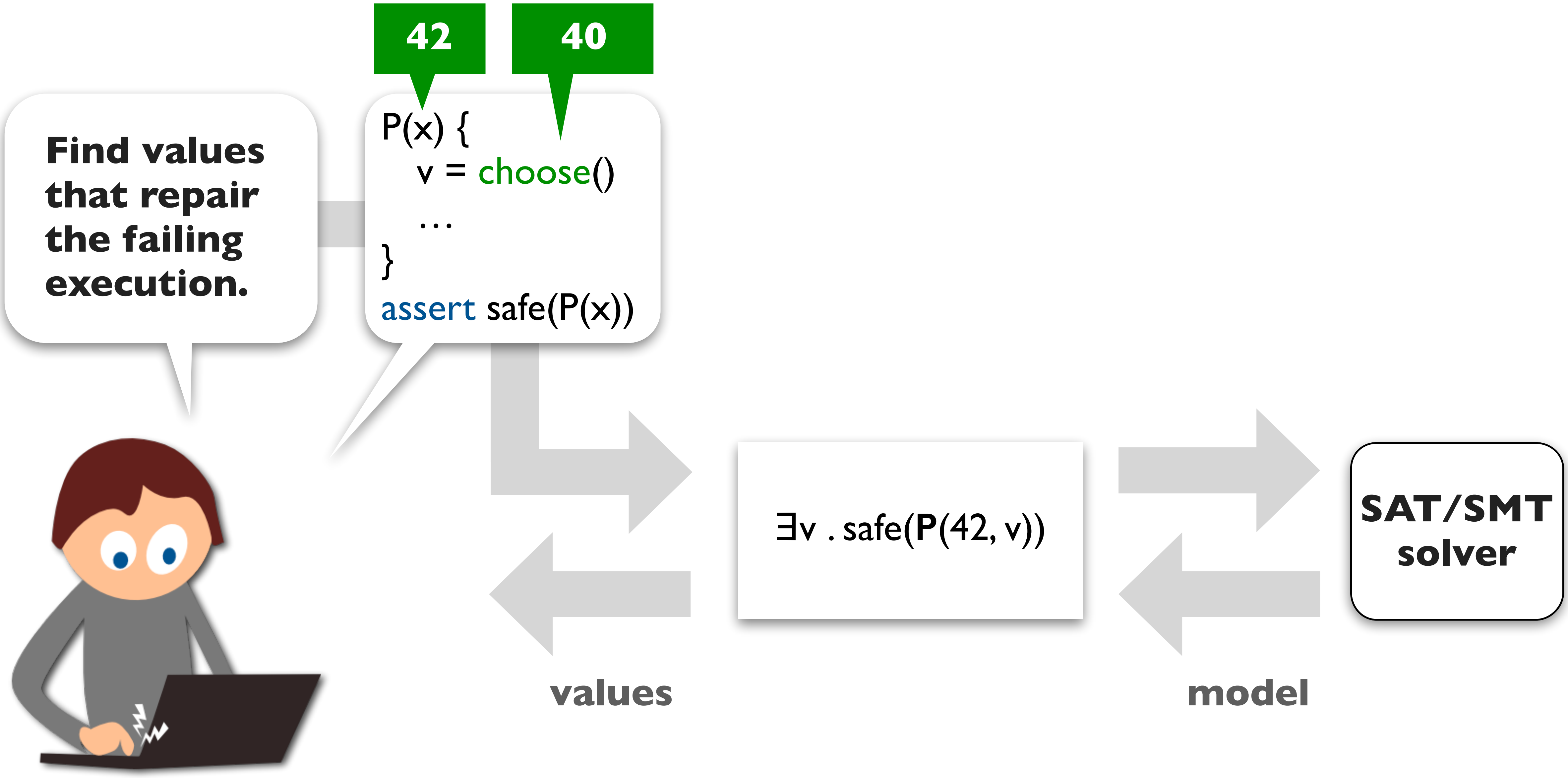
Find values that repair the failing execution.

```
P(x) {
  v = choose()
  …
}
assert safe(P(x))
```

$\exists v$. safe(P(42, v))

Kaplan [Koksal et al, POPL'12]
PBnJ [Samimi et al., ECOOP'10]
Squander [Milicevic et al., ICSE'11]

Solver-aided tools: angelic execution



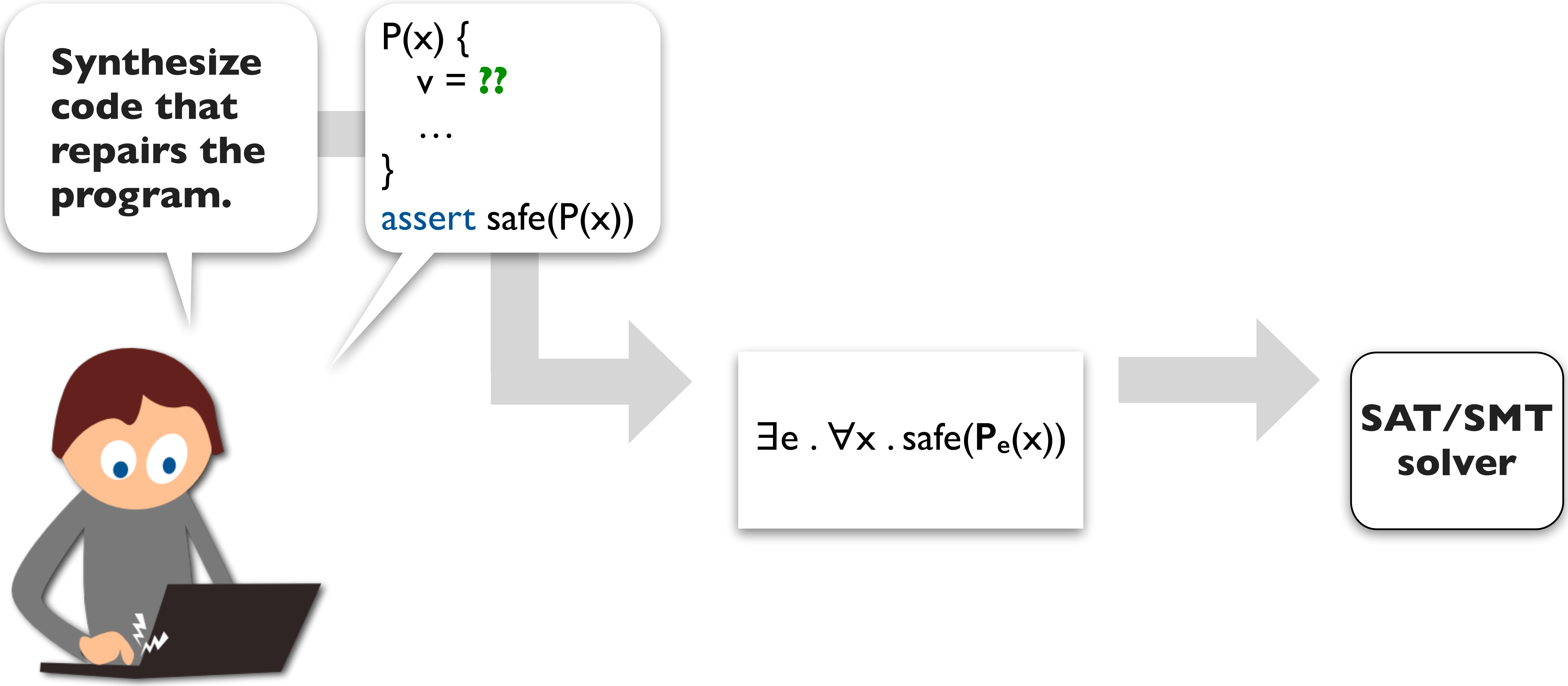
Find values that repair the failing execution.

```
P(x) {
  v = choose()
  …
}
assert safe(P(x))
```

$\exists v$. safe(P(42, v))

Kaplan [Koksal et al, POPL'12]
PBnJ [Samimi et al., ECOOP'10]
Squander [Milicevic et al., ICSE'11]

Solver-aided tools: synthesis



Synthesize code that repairs the program.

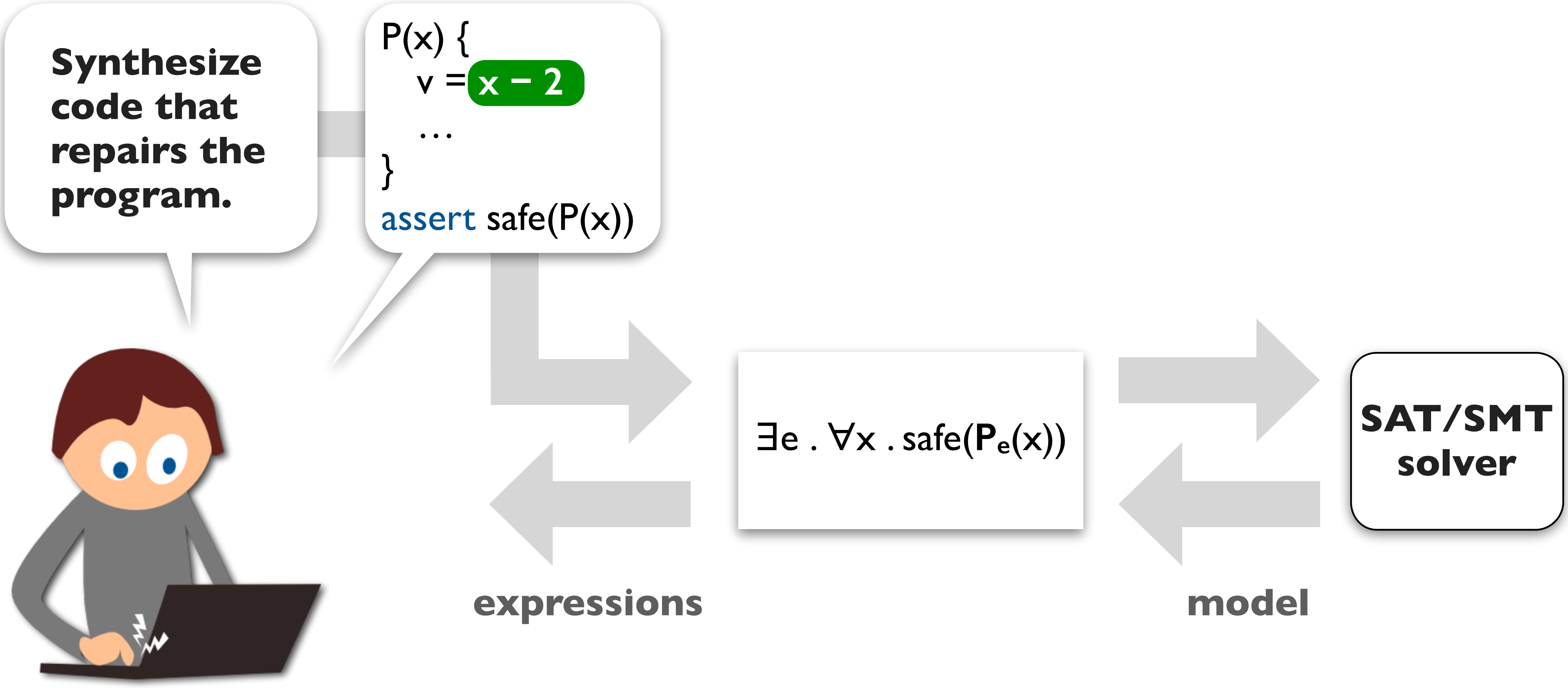
```
P(x) {
  v = ??
    …
}
assert safe(P(x))
```

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

SAT/SMT solver

Sketch [Solar-Lezama et al., ASPLOS'06]
Comfusy [Kuncak et al., CAV'10]

Solver-aided tools: synthesis



Synthesize code that repairs the program.

```
P(x) {
   v = x − 2
   …
}
assert safe(P(x))
```

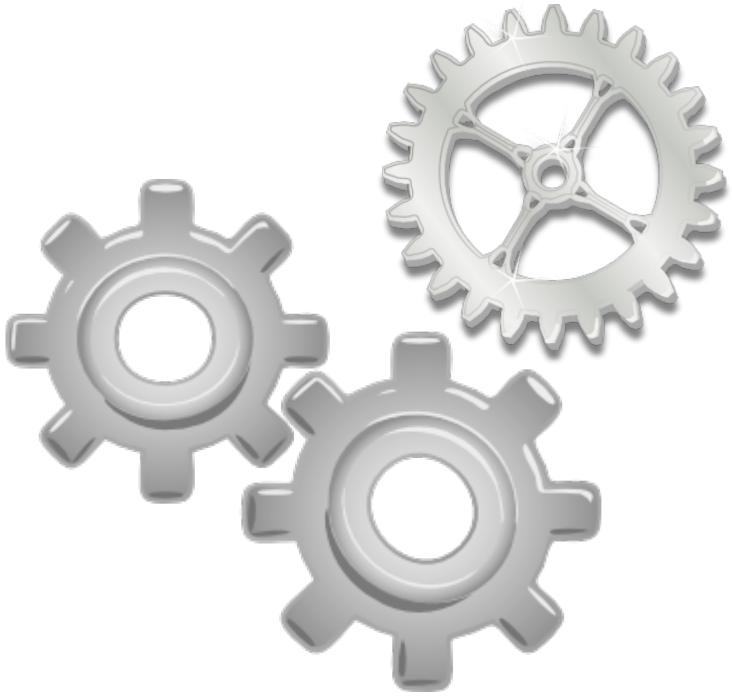
$\exists e \; . \; \forall x \; . \; \mathrm{safe}(\mathbf{P}_e(x))$

SAT/SMT solver

Sketch [Solar-Lezama et al., ASPLOS'06]
Comfusy [Kuncak et al., CAV'10]

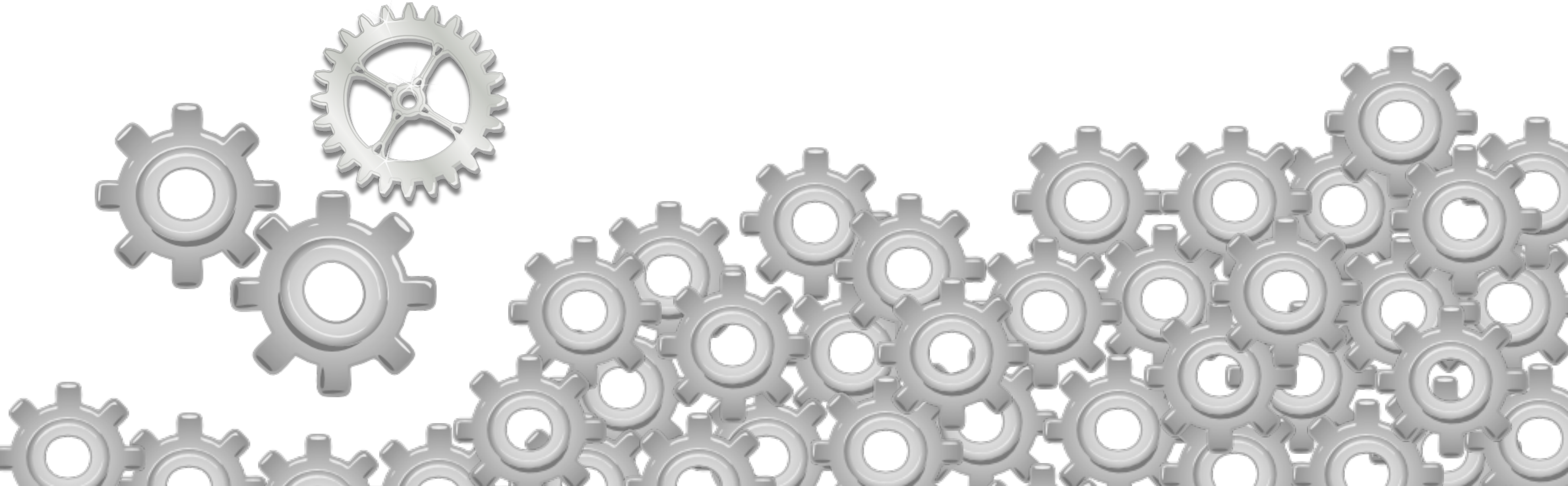
wanted

more solver-aided tools …

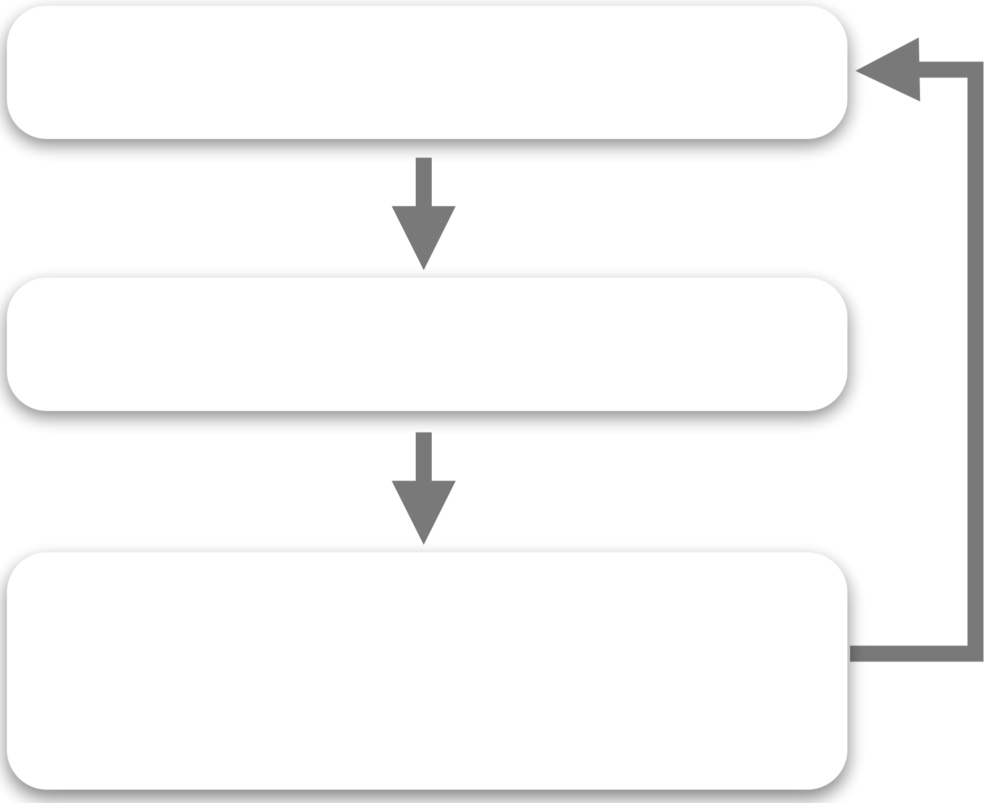


wanted

more solver-aided tools …

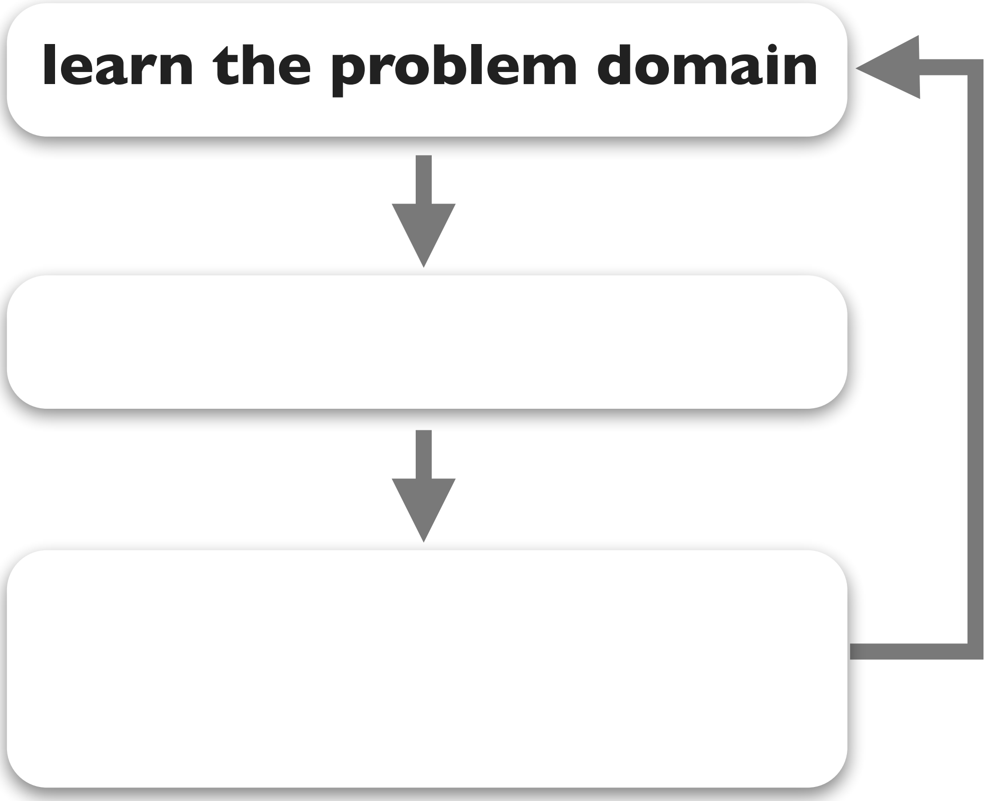


Building solver-aided tools: state-of-the-art

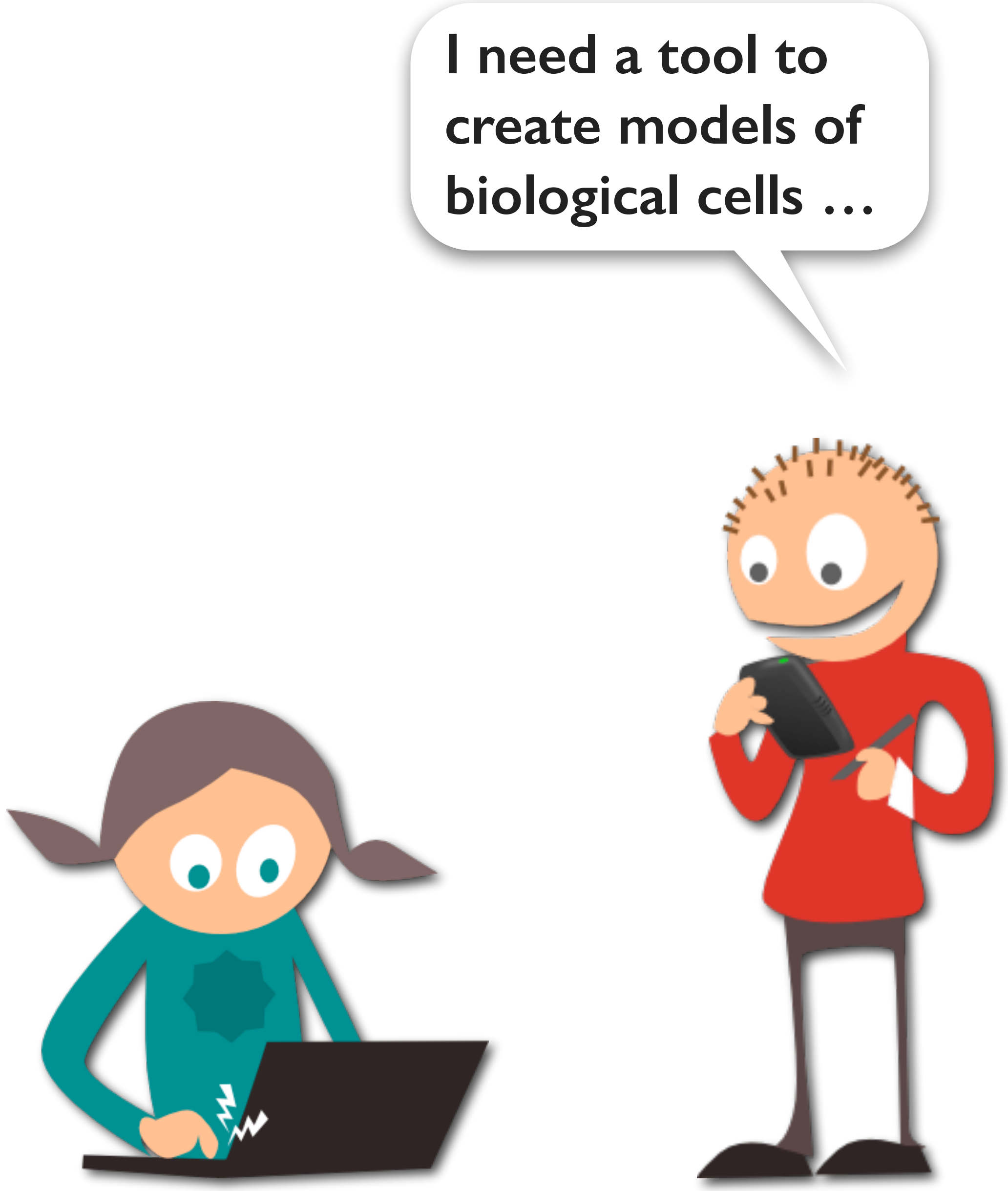


tools expert

Building solver-aided tools: state-of-the-art



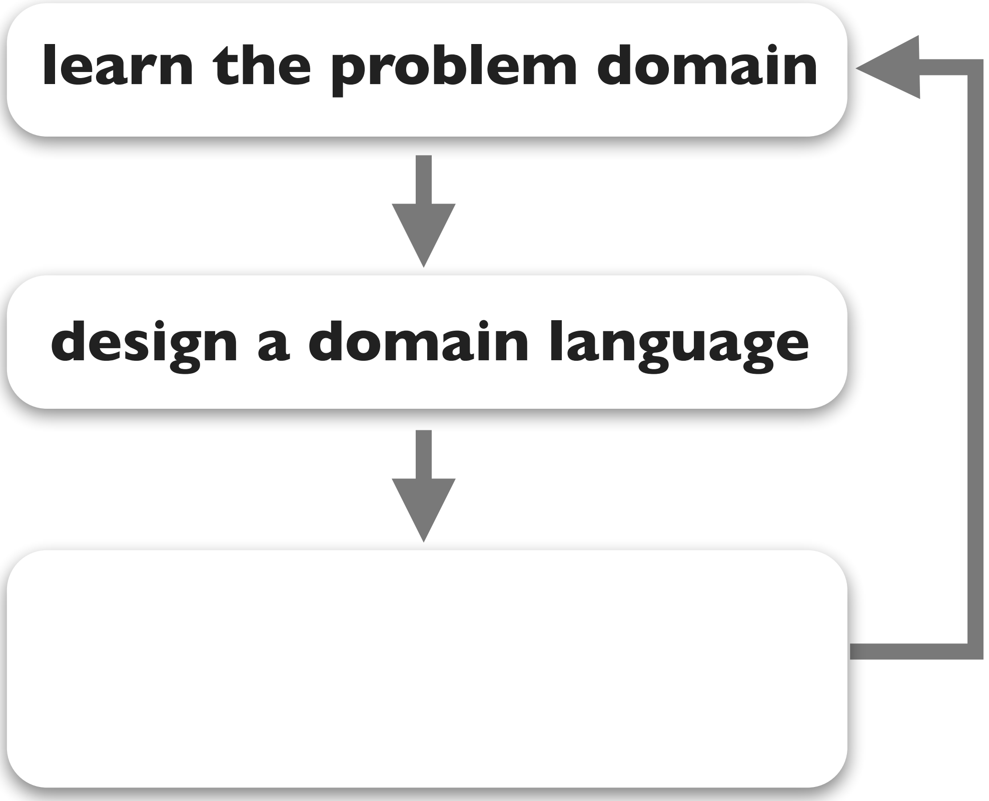
learn the problem domain



tools expert

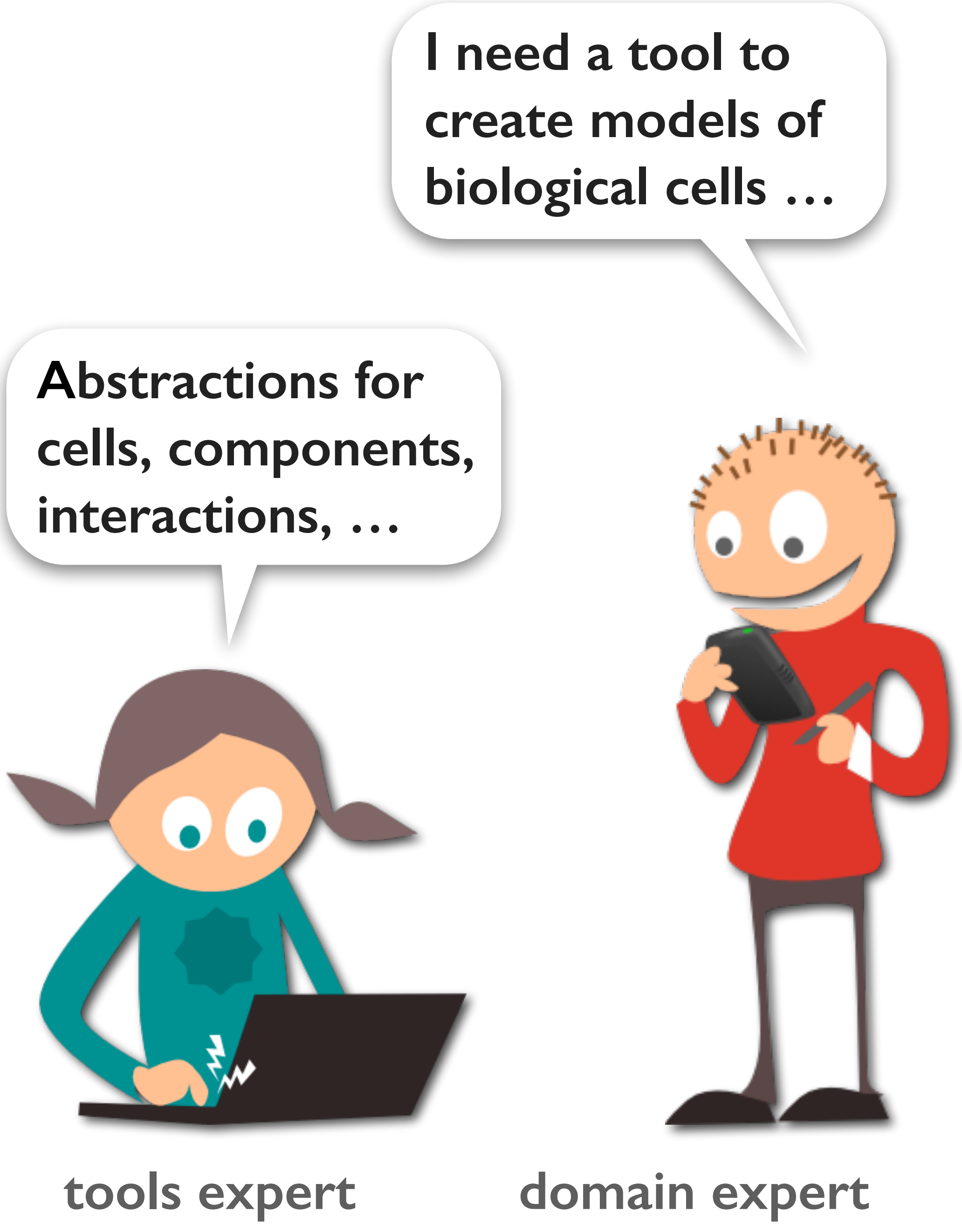
domain expert

Building solver-aided tools: state-of-the-art

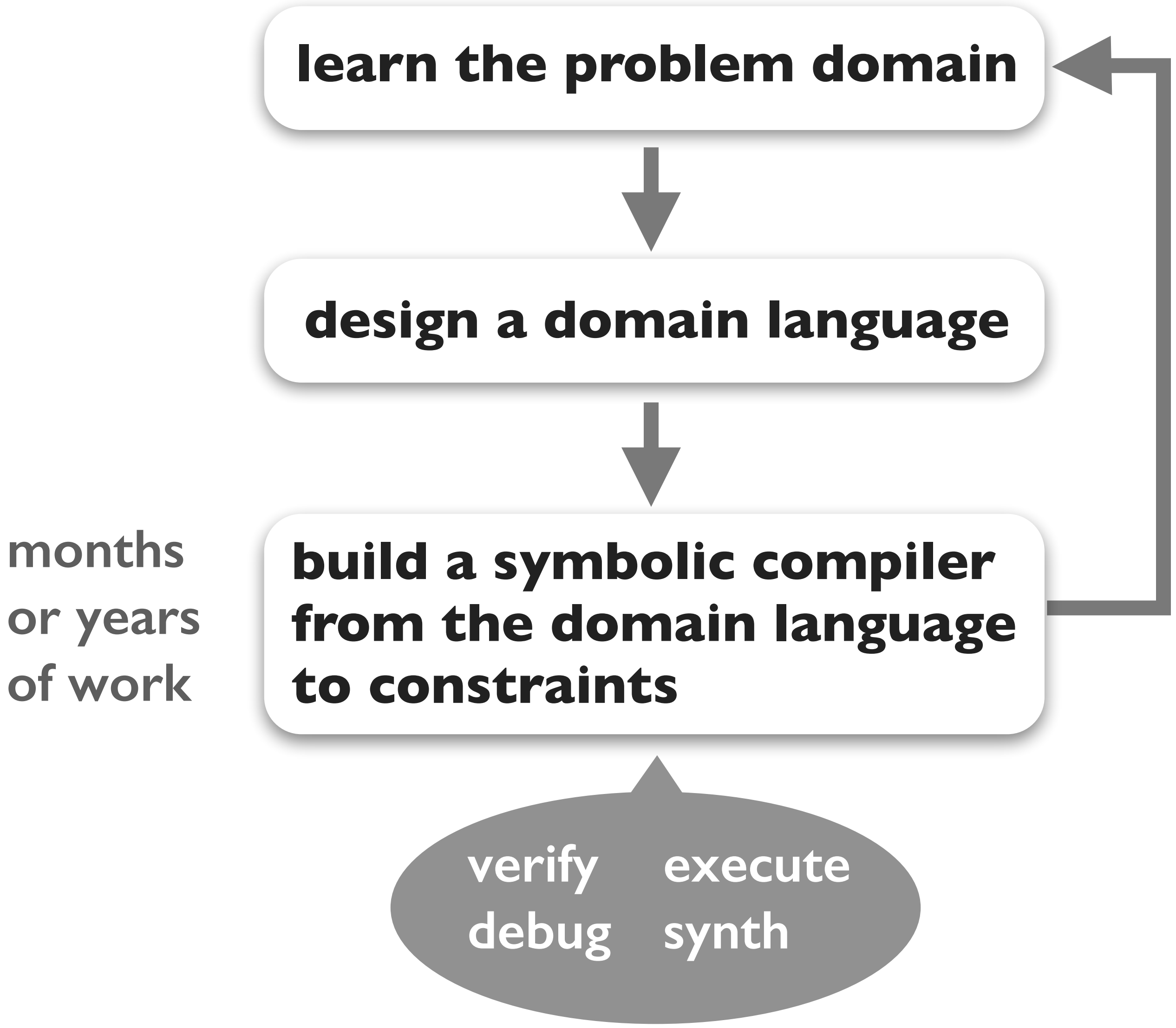


learn the problem domain

design a domain language


I need a tool to create models of biological cells …
Abstractions for cells, components, interactions, …
tools expert
domain expert

Building solver-aided tools: state-of-the-art



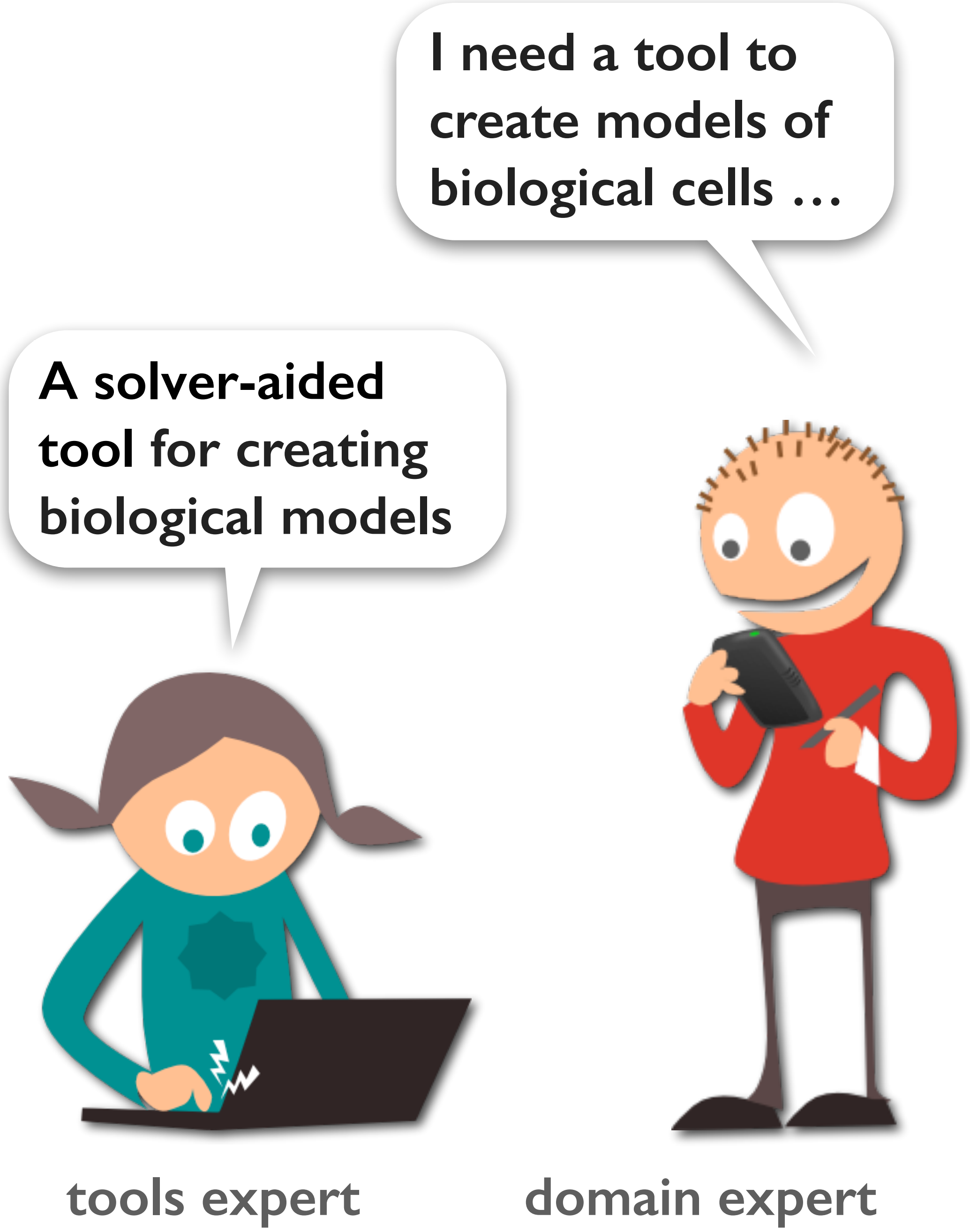
learn the problem domain

design a domain language

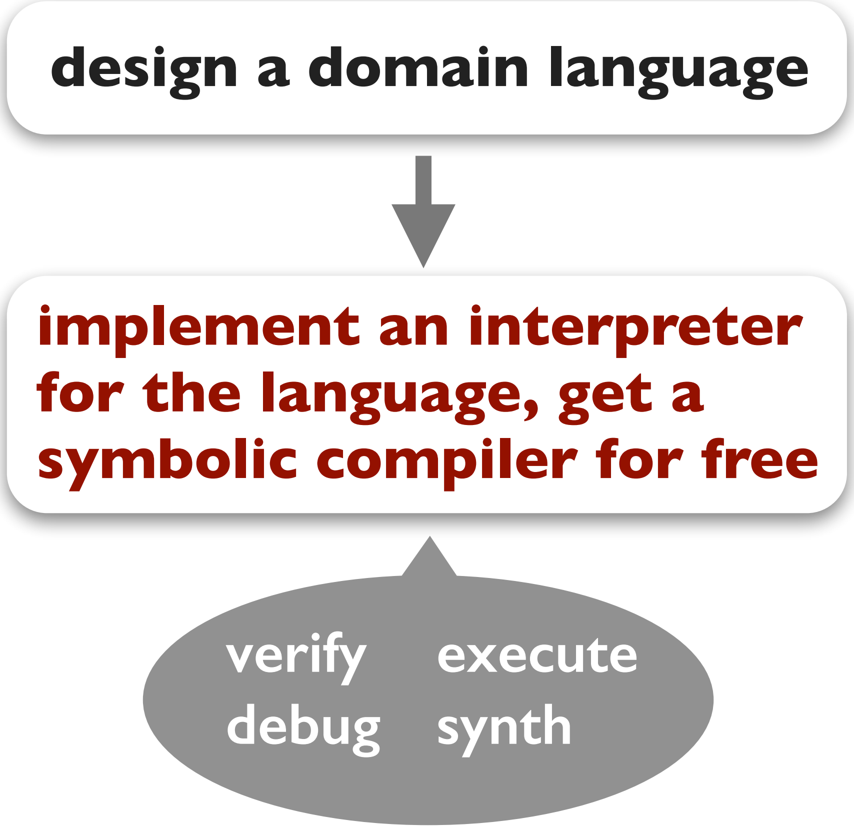
months or years of work

build a symbolic compiler from the domain language to constraints

verify execute debug synth


I need a tool to create models of biological cells …
A solver-aided tool for creating biological models
tools expert
domain expert

Can we do better?



design a domain language

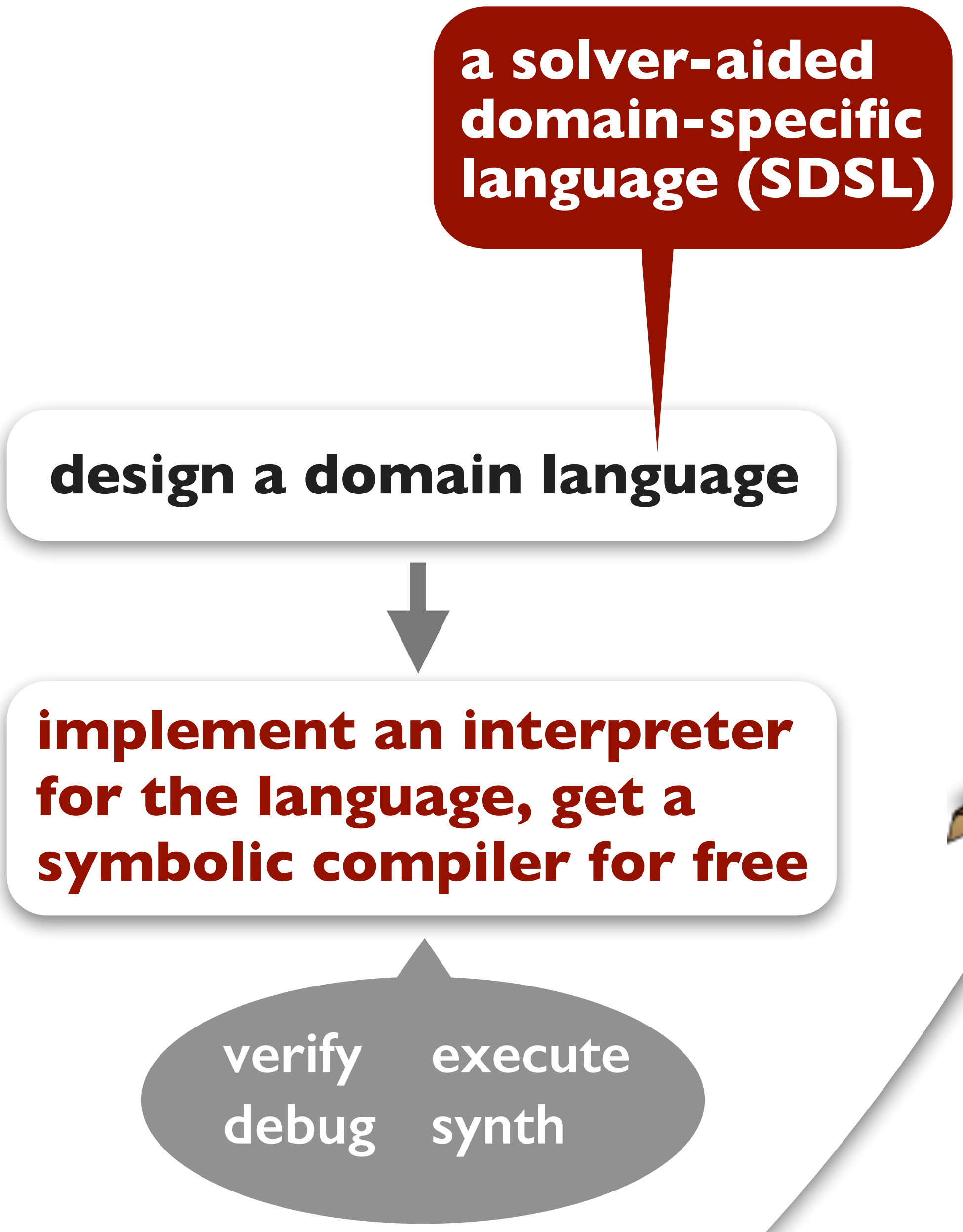
weeks

implement an interpreter for the language, get a symbolic compiler for free

verify execute
debug synth

domain expert

Can we do better?



a solver-aided domain-specific language (SDSL)

design a domain language

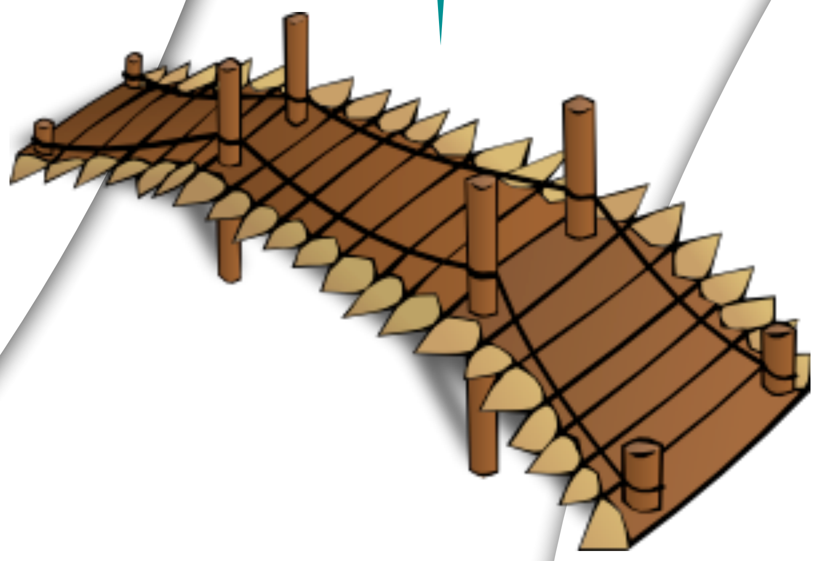
weeks

implement an interpreter for the language, get a symbolic compiler for free

verify execute

debug synth

a solver-aided host language

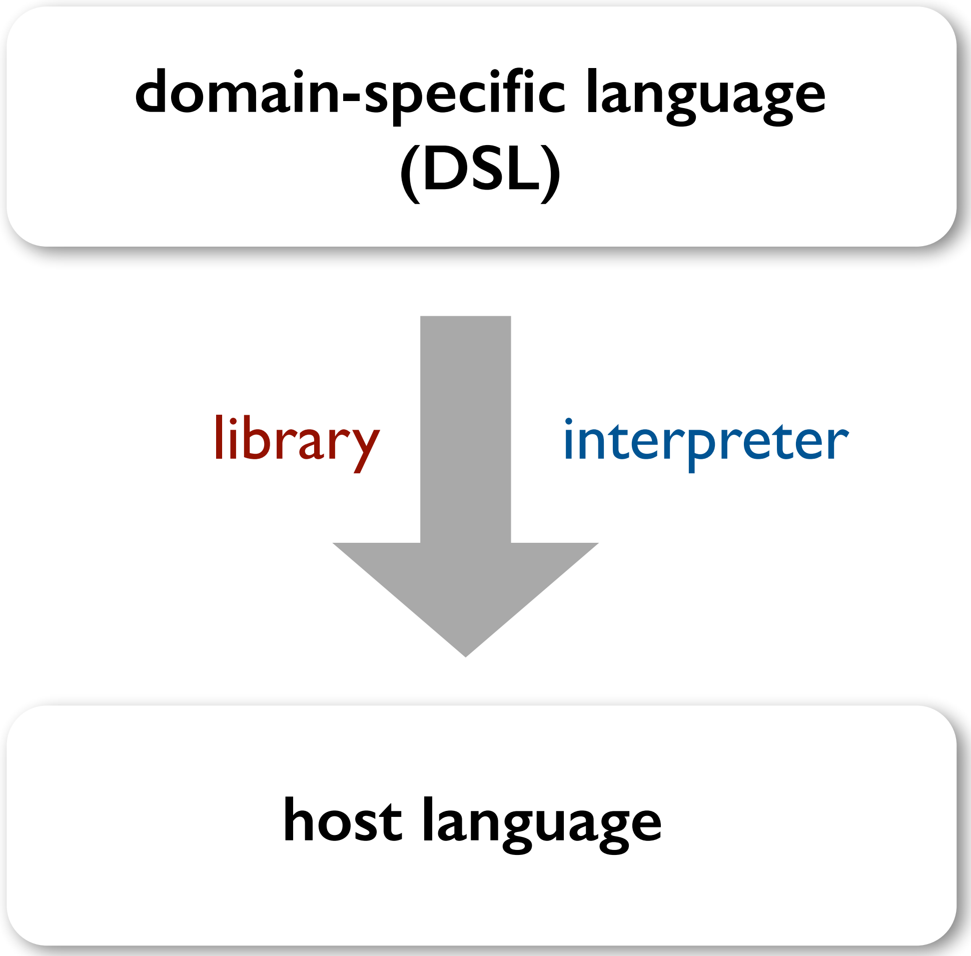


domain expert

design

solver-aided languages

Layers of languages



domain-specific language (DSL)

library

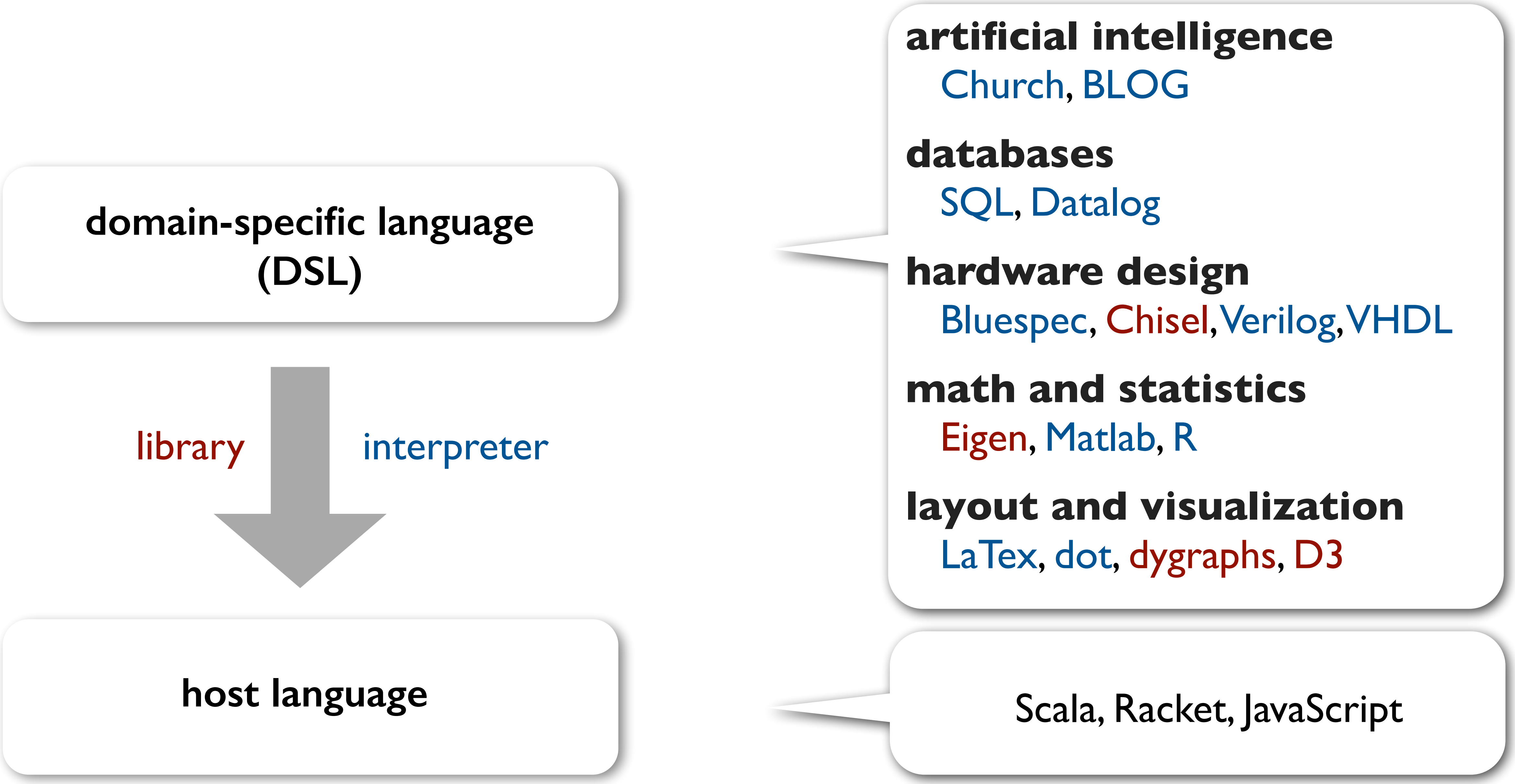
interpreter

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 languages



domain-specific language
(DSL)

library

interpreter

host language

artificial intelligence
Church, BLOG

databases
SQL, Datalog

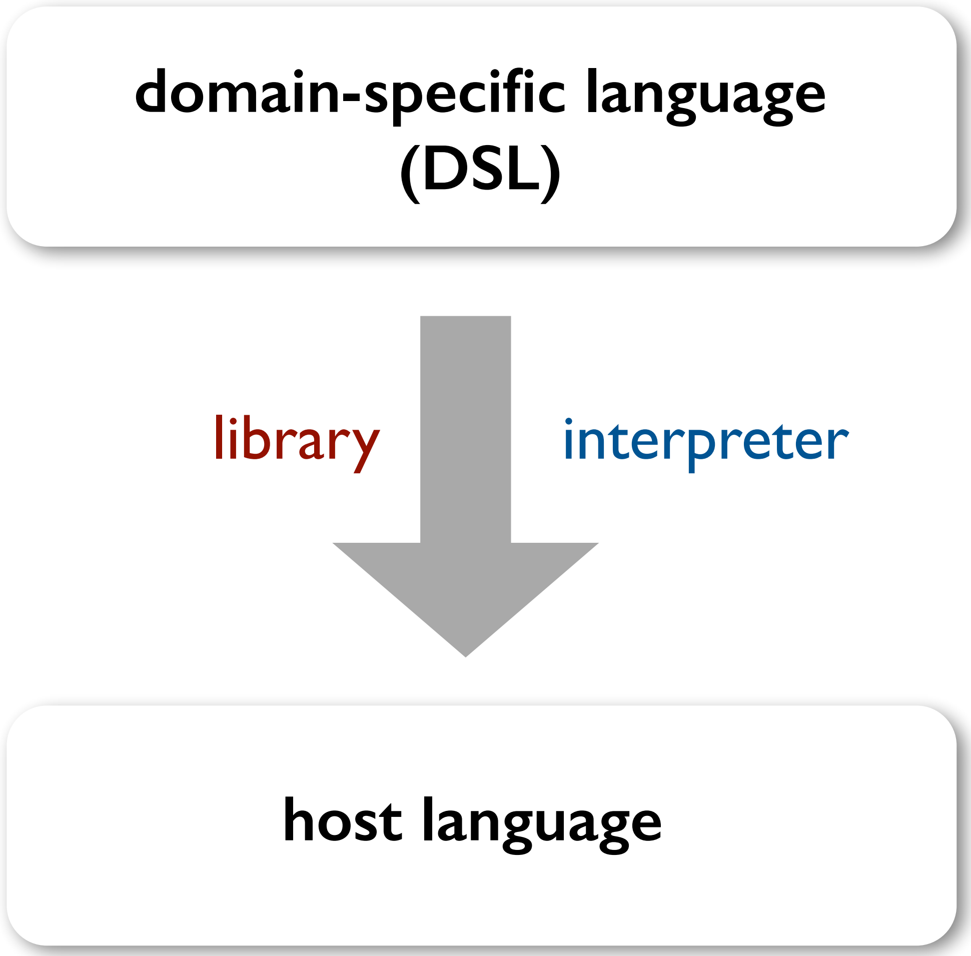
hardware design
Bluespec, Chisel, Verilog, VHDL

math and statistics
Eigen, Matlab, R

layout and visualization
LaTex, dot, dygraphs, D3

Scala, Racket, JavaScript

Layers of languages



domain-specific language (DSL)

library

interpreter

host language

Eigen / Matlab

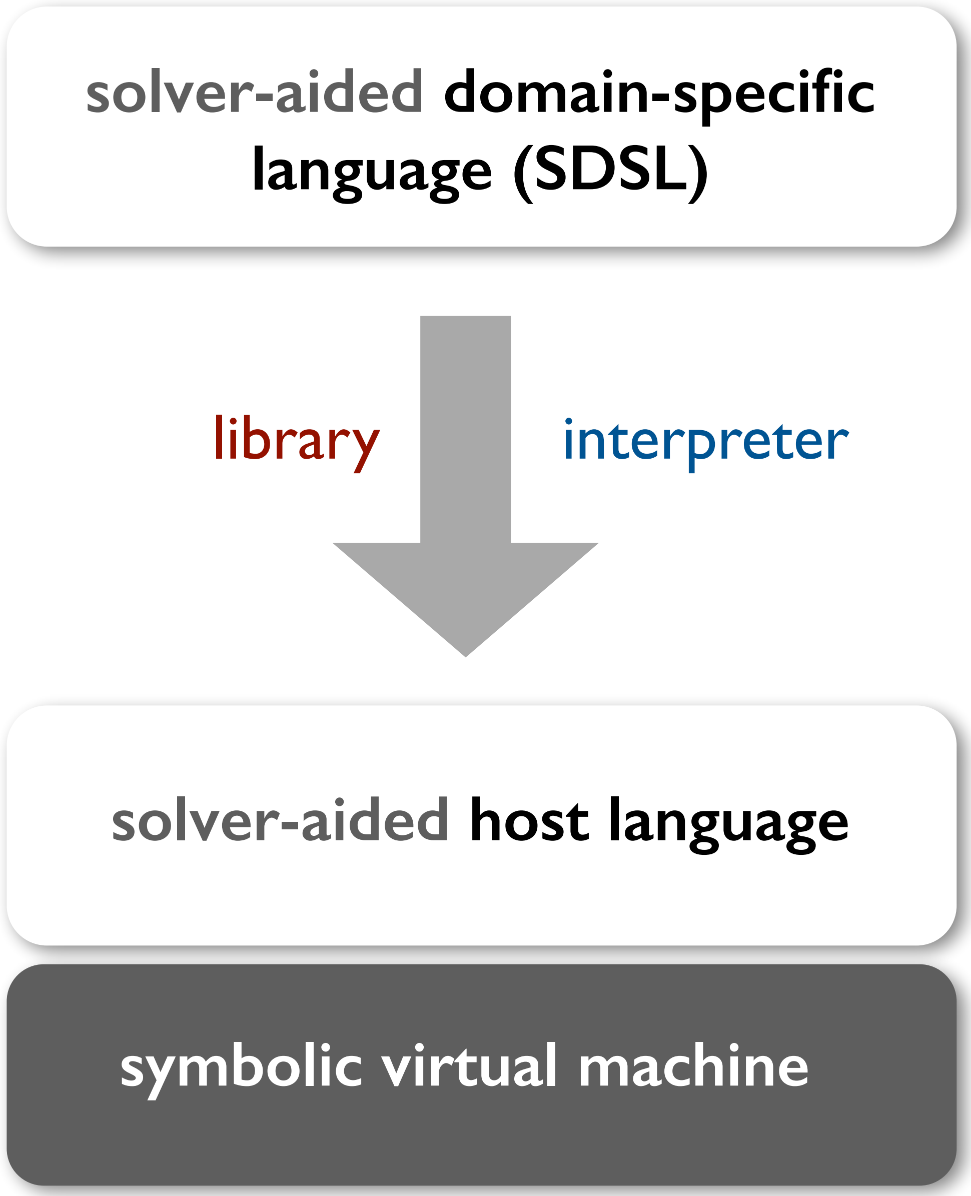
```
C = A * B
```

[associativity]

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)**

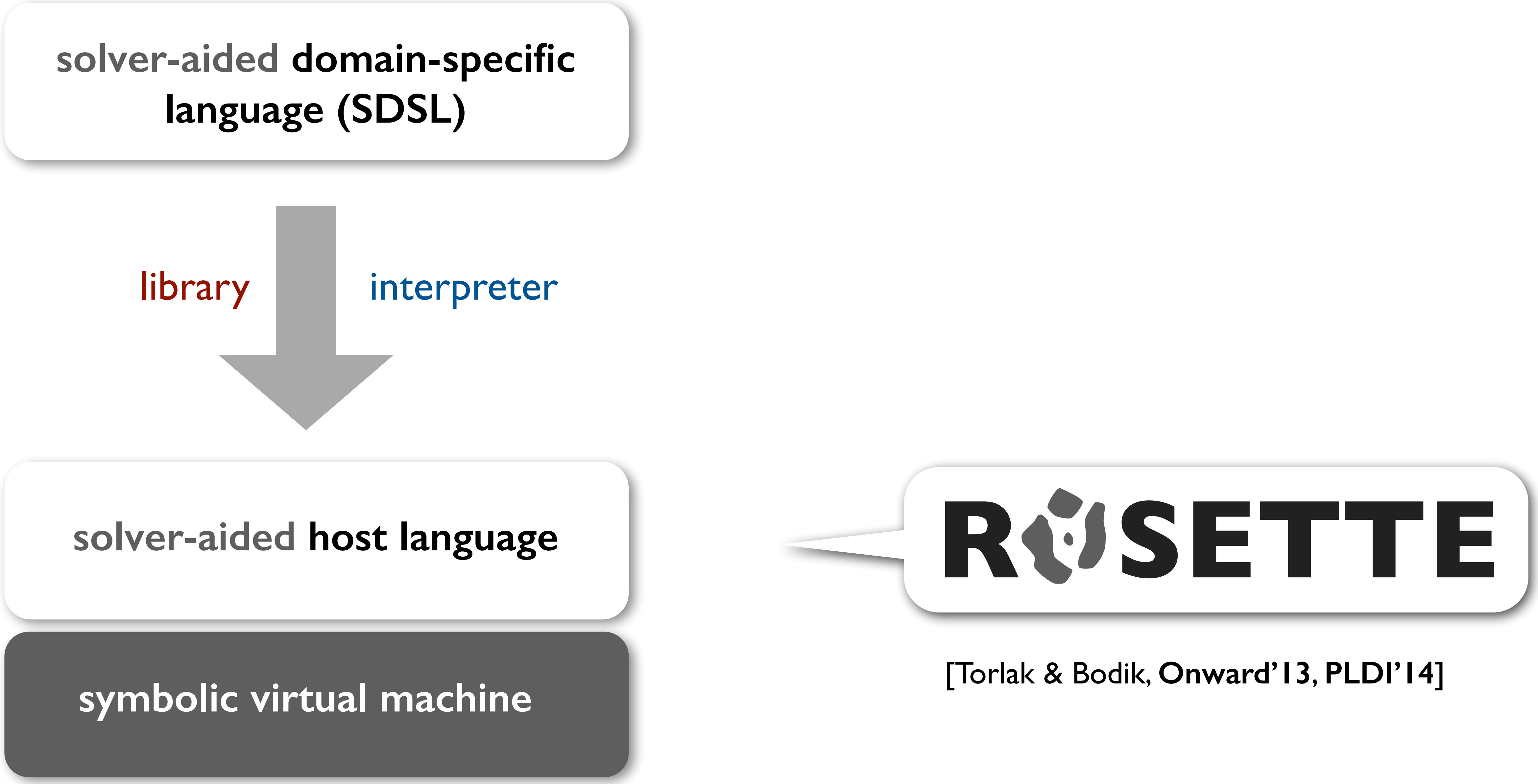
library

interpreter

solver-aided **host language**

symbolic virtual machine

Layers of solver-aided languages



solver-aided domain-specific language (SDSL)

library

interpreter

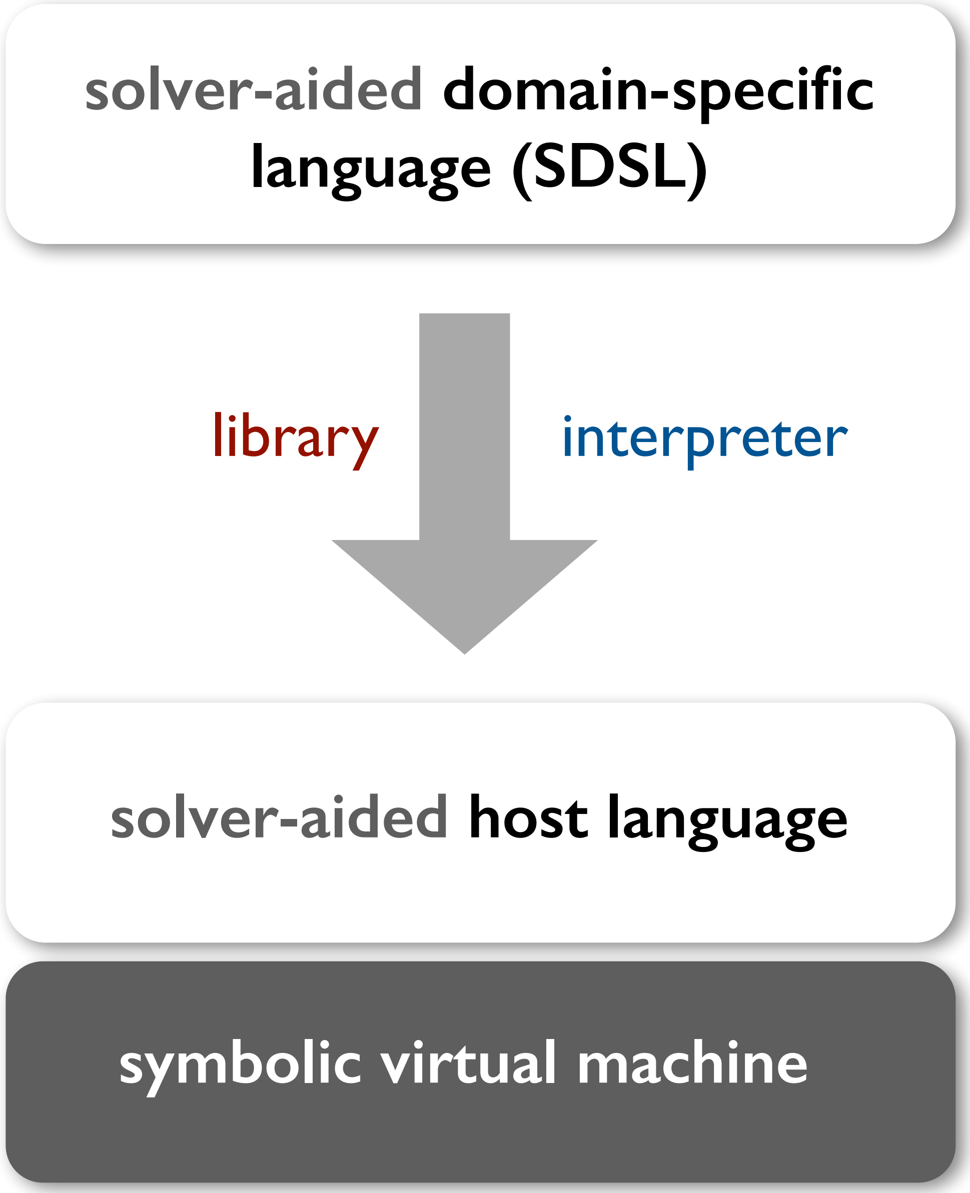
solver-aided host language

symbolic virtual machine

ROSETTE

[Torlak & Bodik, **Onward'13, PLDI'14**]

Layers of solver-aided languages



solver-aided domain-specific language (SDSL)

library

interpreter

solver-aided host language

symbolic virtual machine

spatial programming
Chlorophyll

data-parallel programming
SynthCL

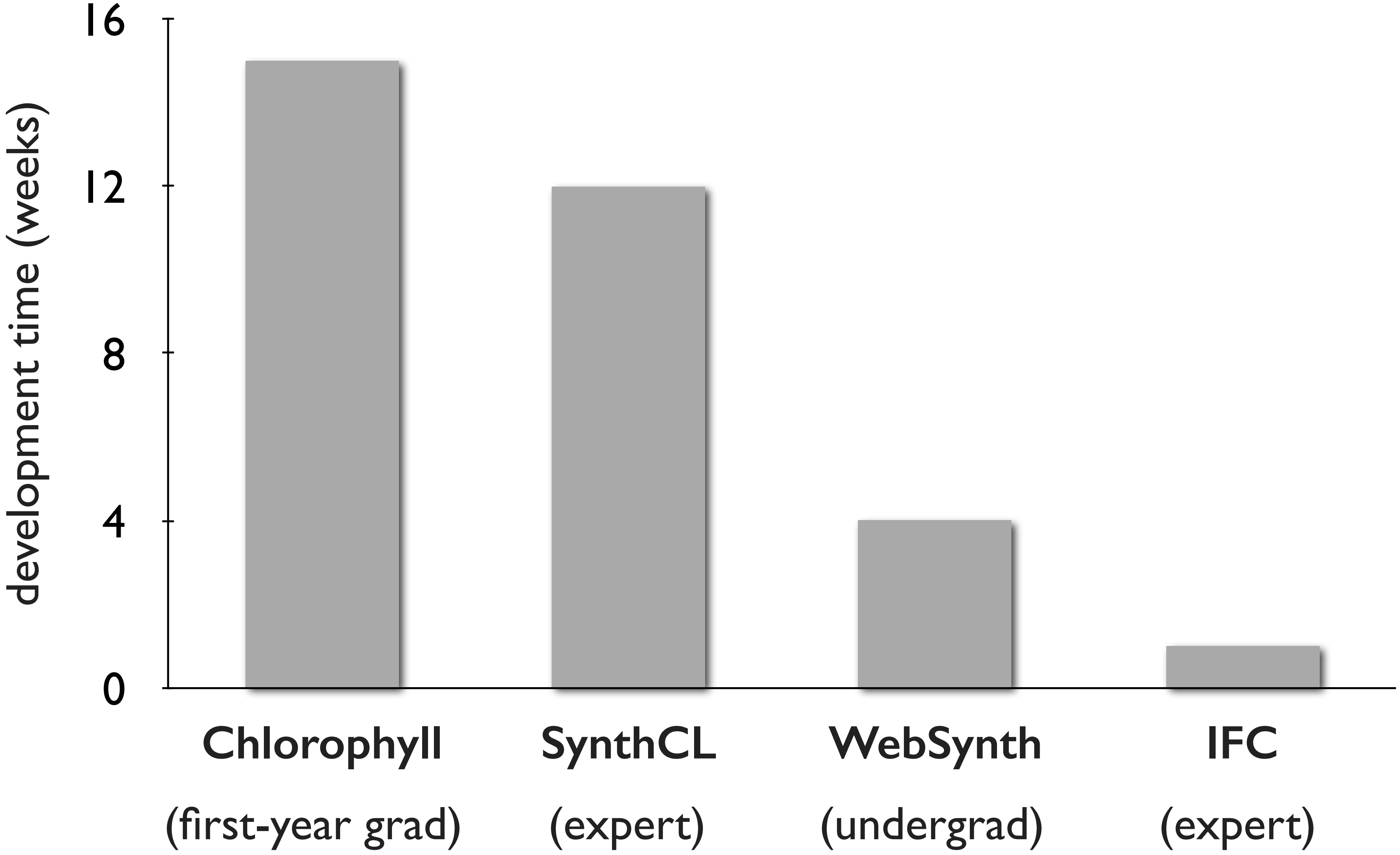
web scraping
WebSynth

secure stack machines
IFC

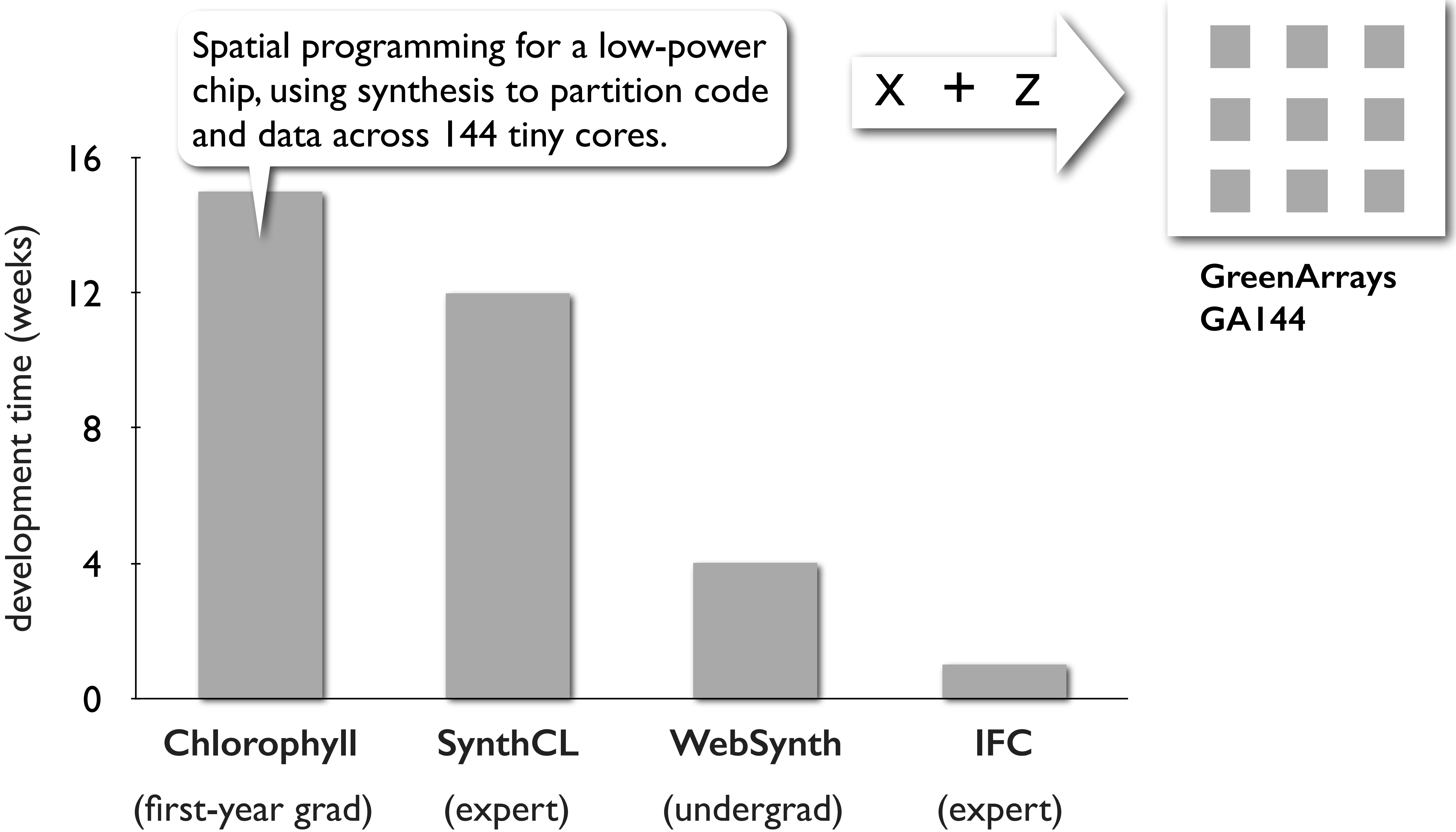
ROSETTE

[Torlak & Bodik, **Onward'13, PLDI'14**]

SDSLs developed with ROSETTE



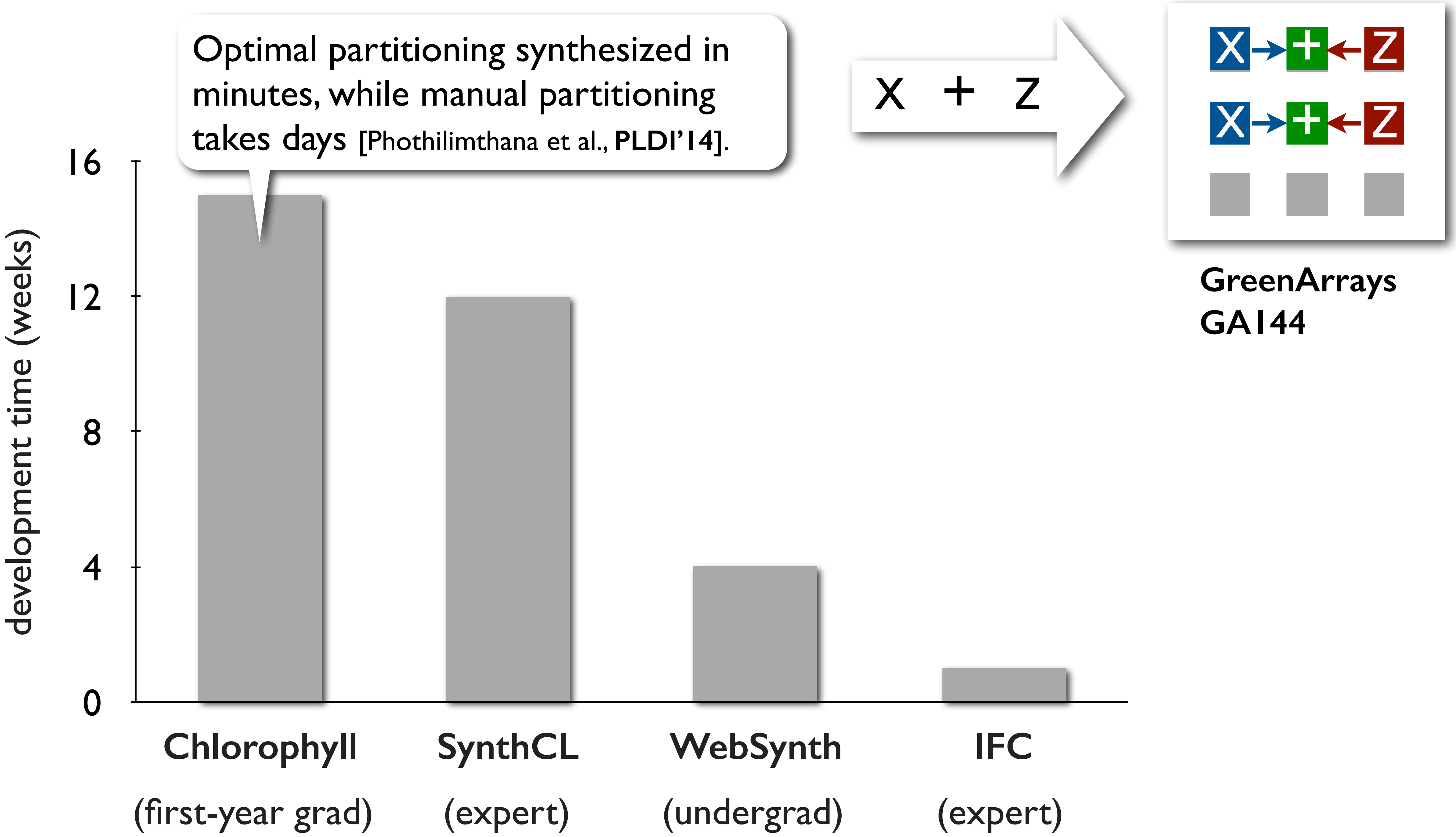
SDSLs developed with ROSETTE



Spatial programming for a low-power chip, using synthesis to partition code and data across 144 tiny cores.

GreenArrays GA144

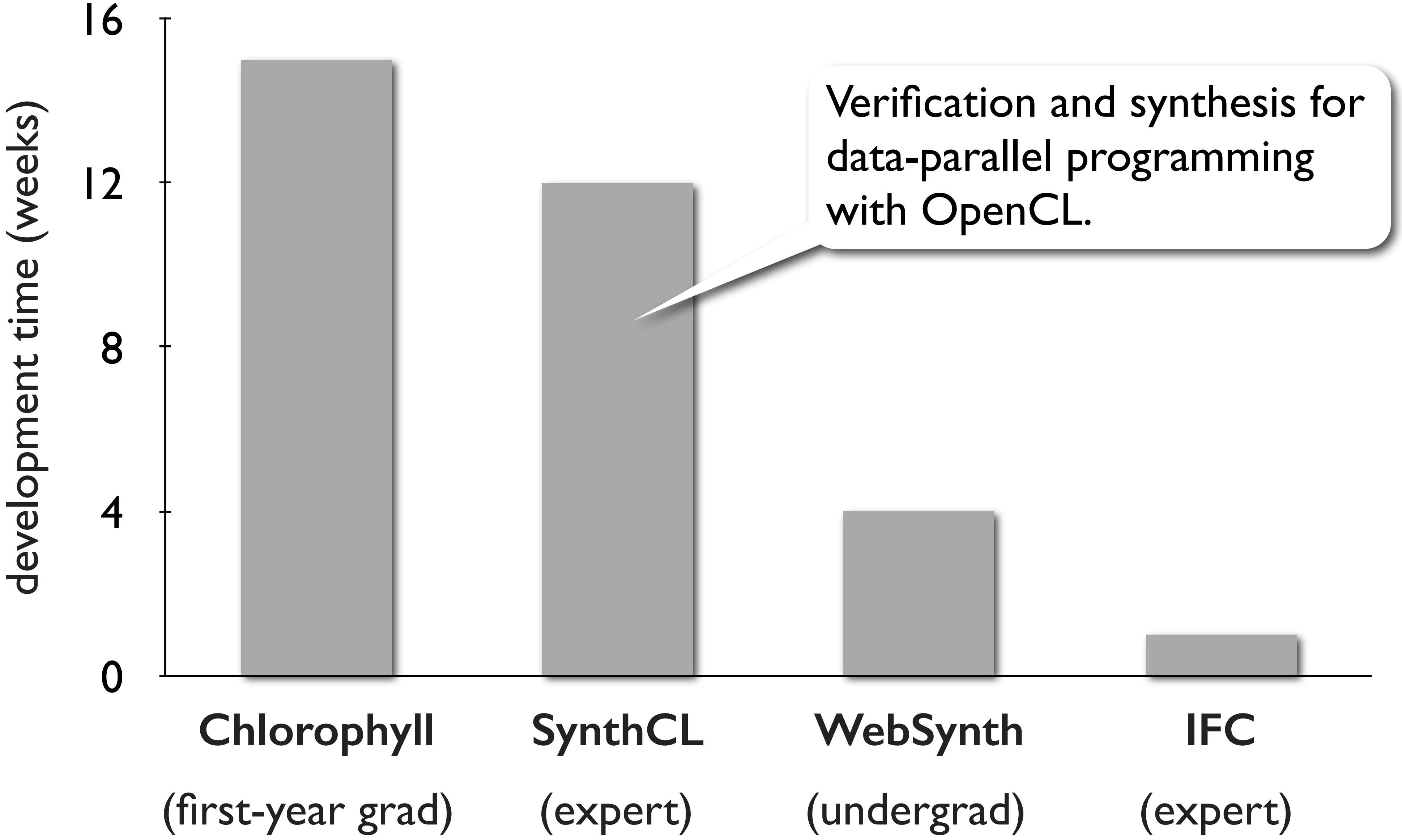
SDSLs developed with ROSETTE



Optimal partitioning synthesized in minutes, while manual partitioning takes days [Phothilimthana et al., **PLDI'14**].

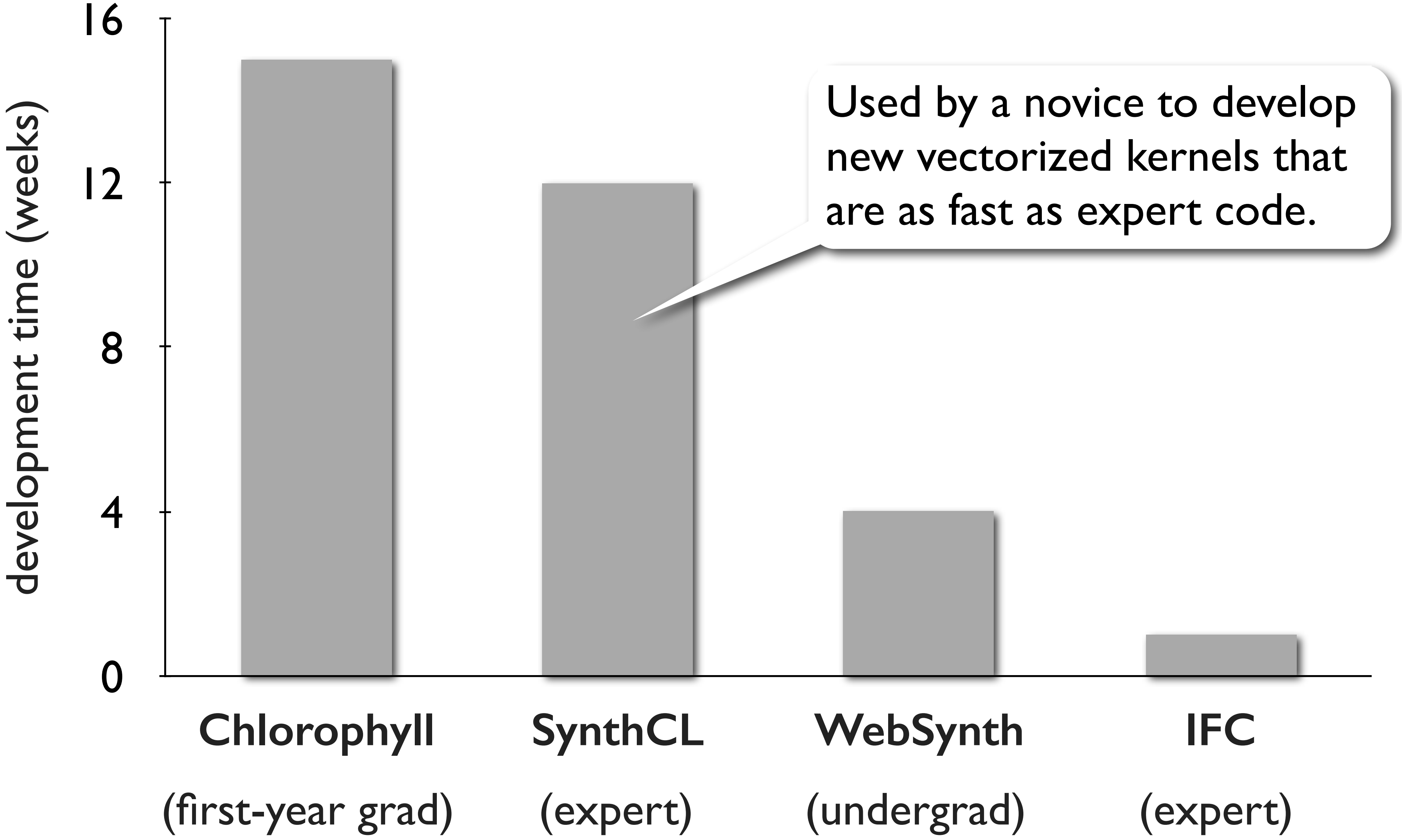
GreenArrays GA144

SDSLs developed with ROSETTE



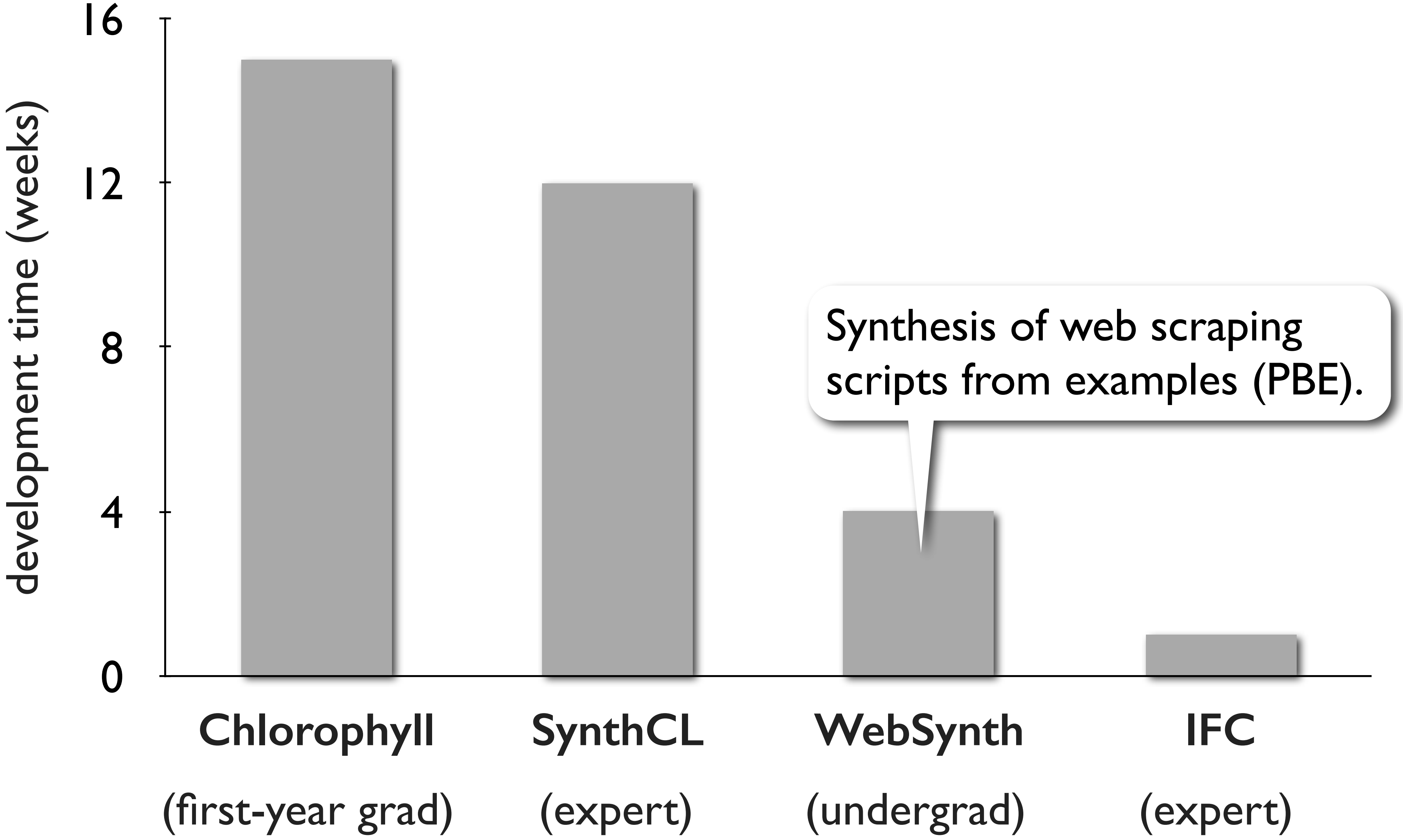
Verification and synthesis for data-parallel programming with OpenCL.

SDSLs developed with ROSETTE



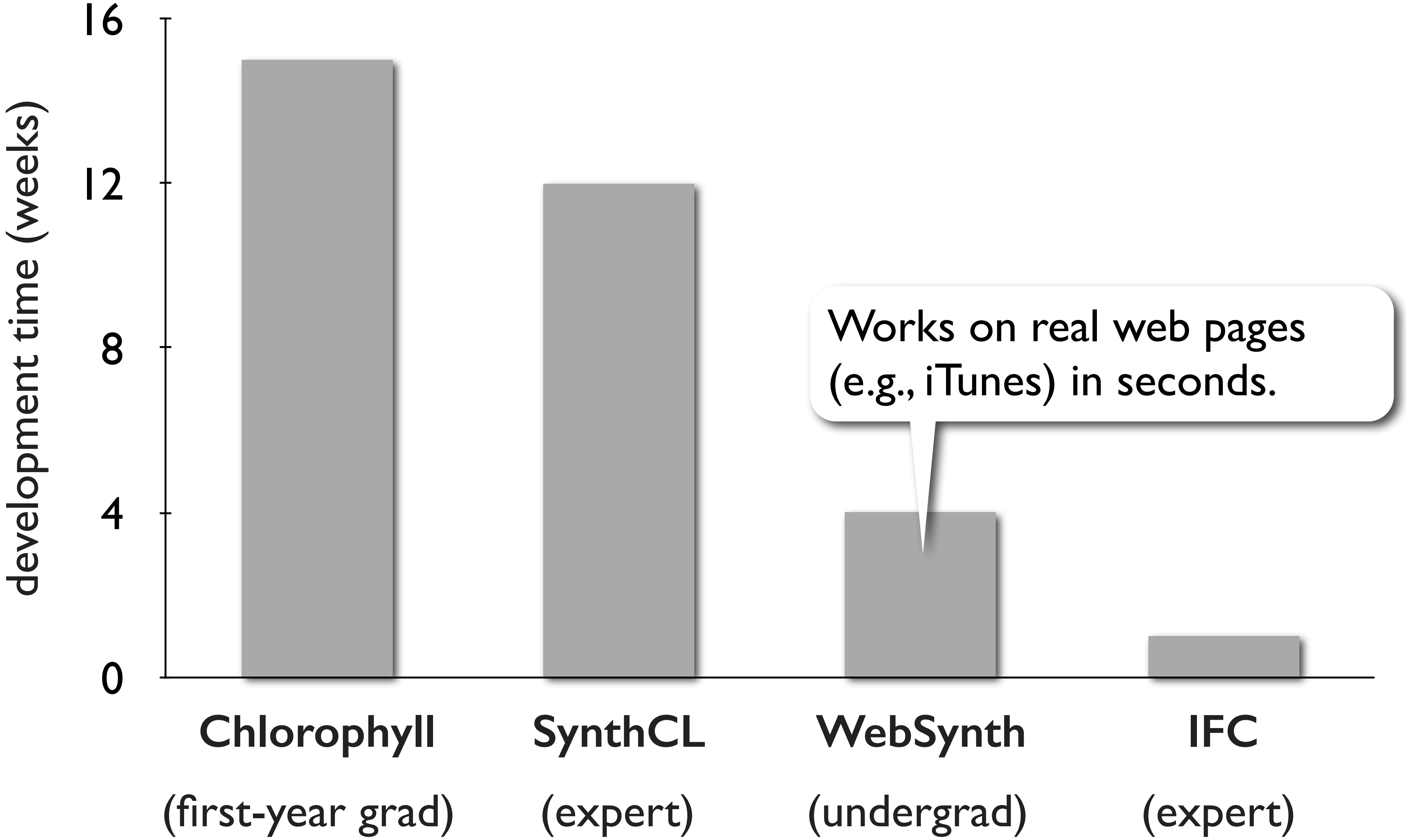
Used by a novice to develop new vectorized kernels that are as fast as expert code.

SDSLs developed with ROSETTE



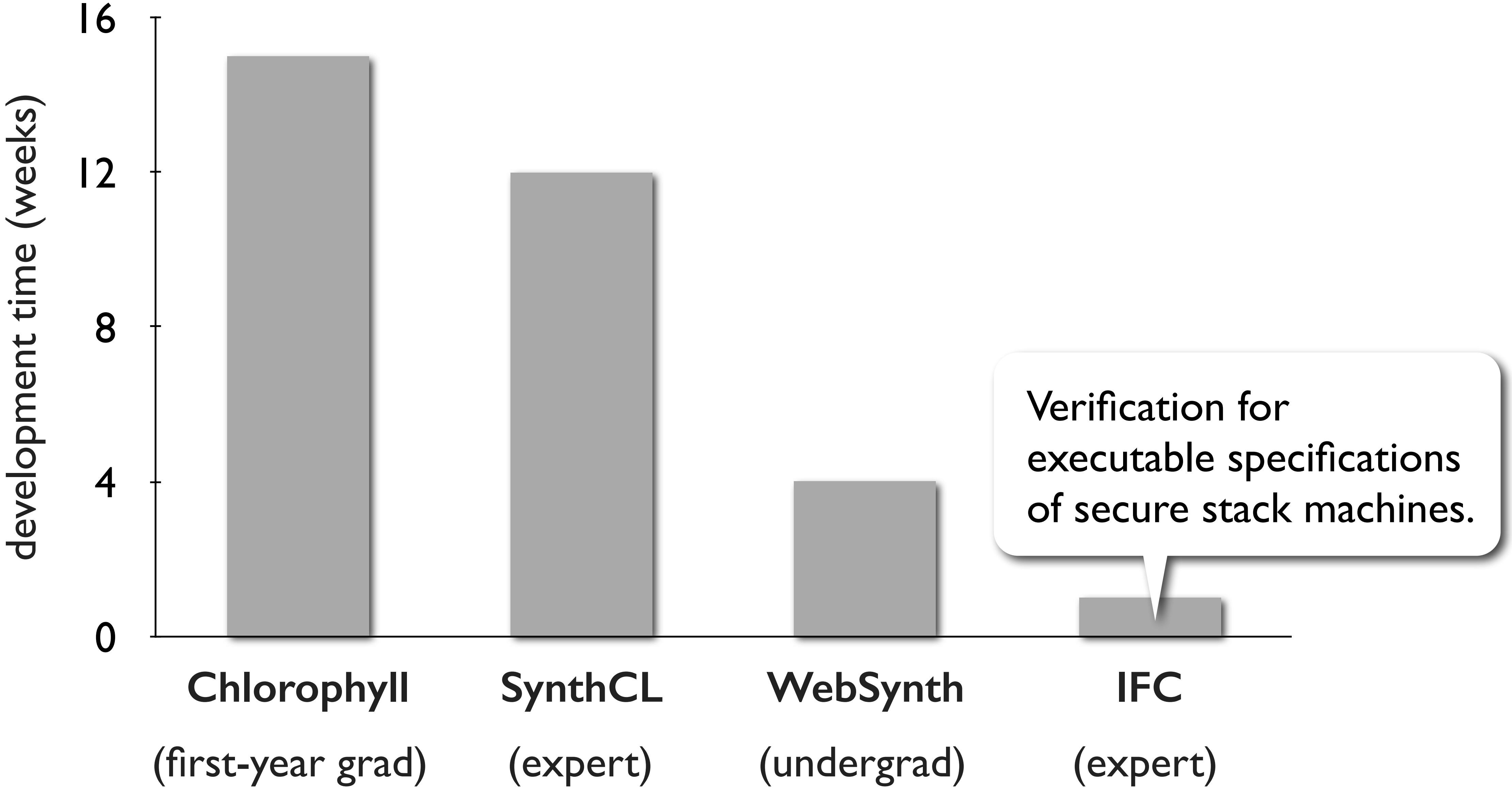
Synthesis of web scraping scripts from examples (PBE).

SDSLs developed with ROSETTE



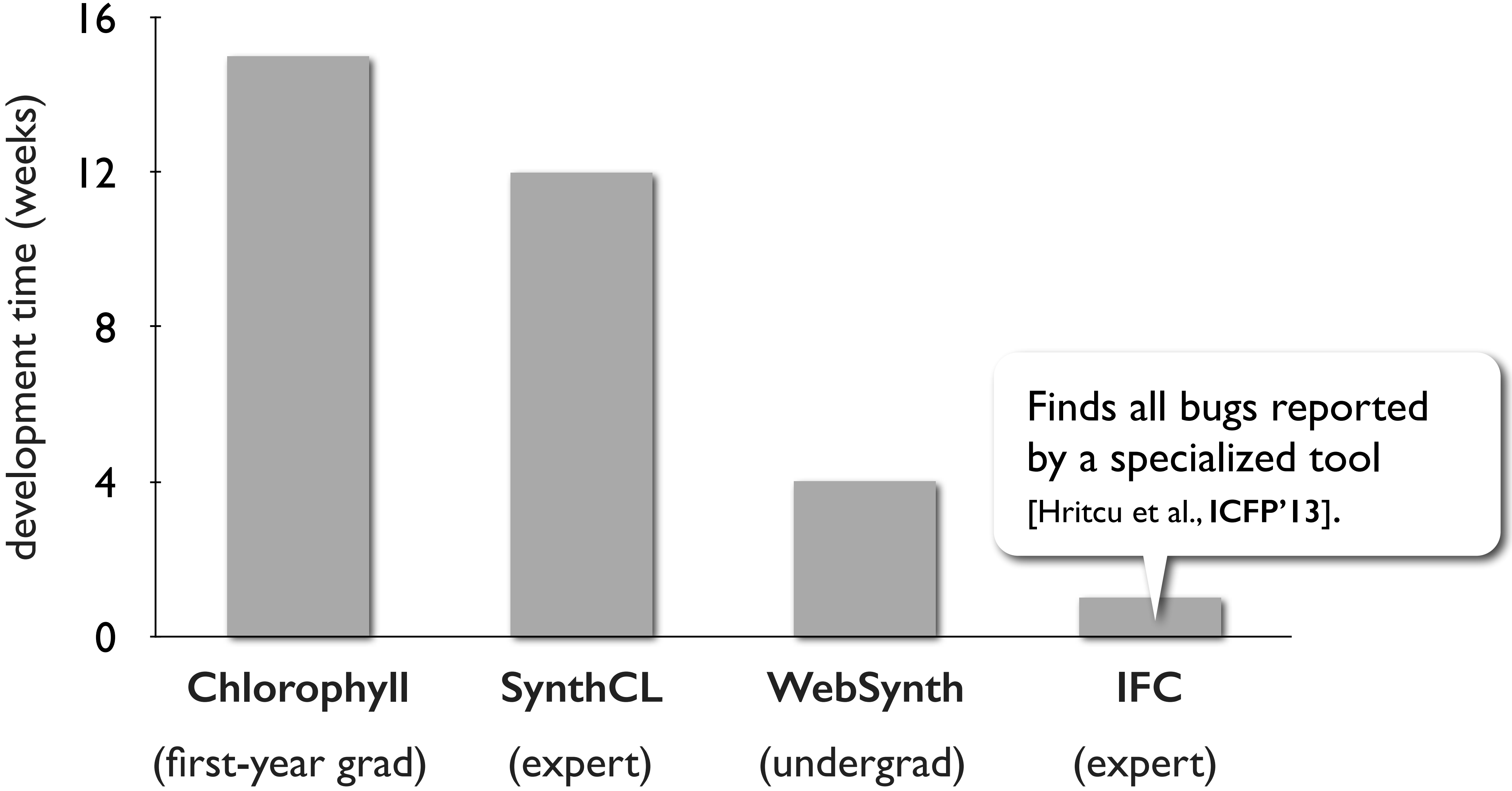
Works on real web pages (e.g., iTunes) in seconds.

SDSLs developed with ROSETTE



Verification for executable specifications of secure stack machines.

SDSLs developed with ROSETTE



Finds all bugs reported by a specialized tool [Hritcu et al., **ICFP'13**].

Anatomy of a solver-aided host language

Modern descendent of Scheme with macro-based metaprogramming.

Racket

Anatomy of a solver-aided host language

```
(define-symbolic id type)

(assert expr)

(verify expr)
(debug [expr] expr)
(solve expr)
(synthesize [expr] expr)
```

A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvge(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 = bvge(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.

test **debug**
verify **synth**

A tiny example SDSL

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

test debug
verify synth

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 = bvge(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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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: ROSETTE

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

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

interpret

```
(define bvmax
 `((2 bvge 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: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 bvge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

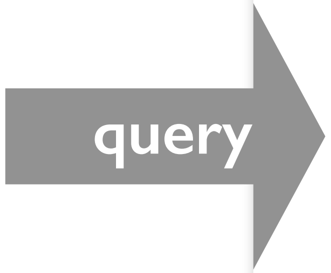
- pattern matching
- dynamic evaluation
- first-class & higher-order procedures
- side effects

```
(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 = bvge(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 n0 n1 number?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 constants of type number and binds them to variables n0 and n1.

```
(define-symbolic n0 n1 number?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

> verify(bvmax, max)
```

query

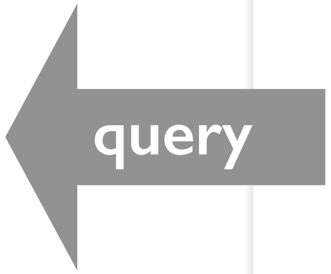
Symbolic values can be used just like concrete values of the same type.

```
(define-symbolic n0 n1 number?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

> verify(bvmax, max)
(0, -2)
```



```
(define-symbolic n0 n1 number?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

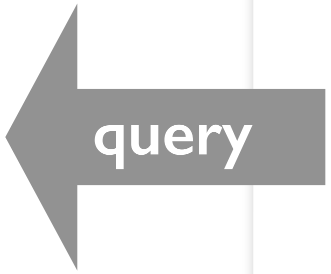
(verify *expr*) searches for a concrete interpretation of symbolic constants that causes *expr* to fail.

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 n0 n1 number?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

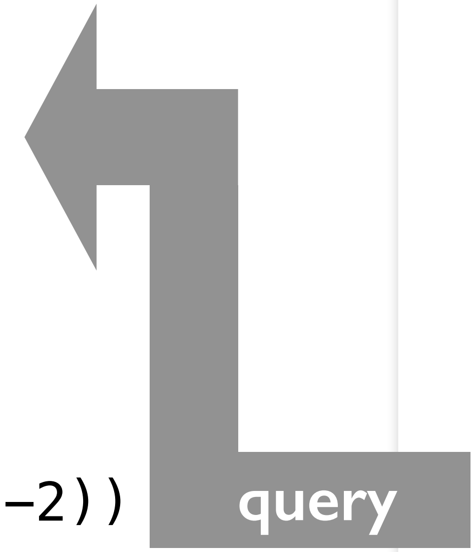
```
> debug(bvmax, max, (0, -2))
```

query

```
(define inputs (list 0 -2))
(debug [input-register?]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 inputs (list 0 -2))
(debug [input-register?]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

> synthesize(bvmax, max)
```

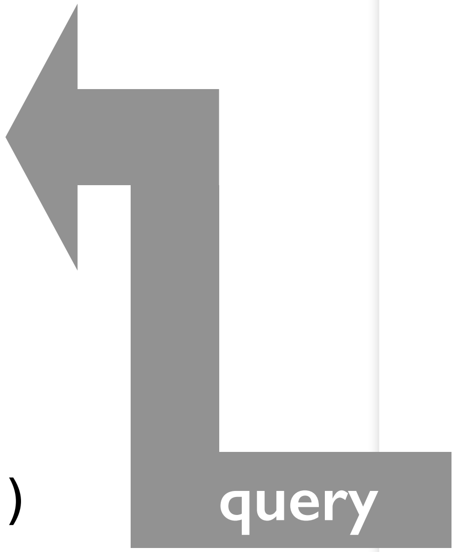
query

```
(define-symbolic n0 n1 number?)
(define inputs (list n0 n1))
(synthesize [inputs]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs)))))
```

A tiny example SDSL: ROSETTE

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

> synthesize(bvmax, max)
```

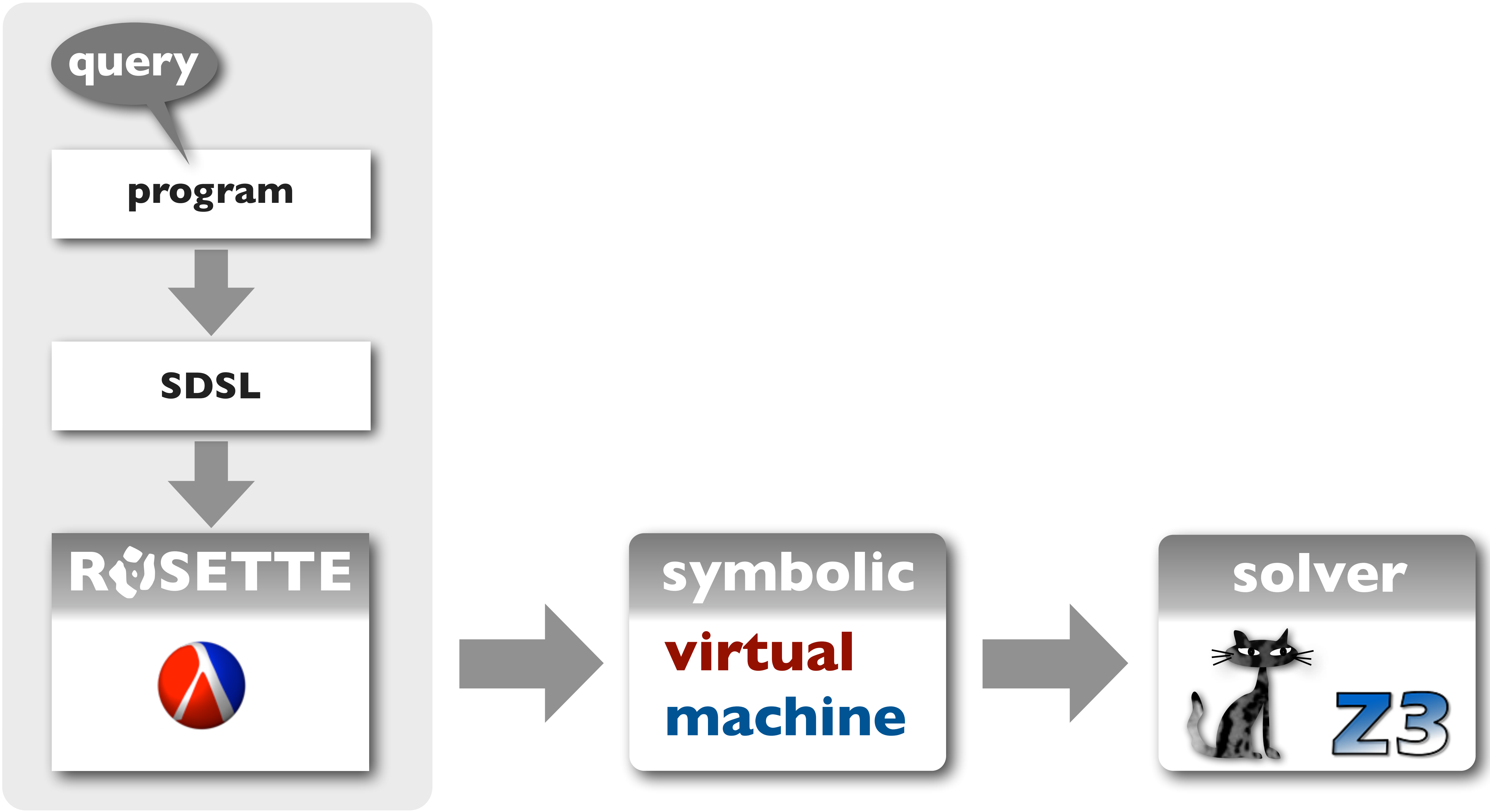


```
(define-symbolic n0 n1 number?)
(define inputs (list n0 n1))
(synthesize [inputs]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

tech

symbolic virtual machine (SVM)

How it all works: a big picture view



query

program

SDSL

ROSETTE

symbolic

virtual

machine

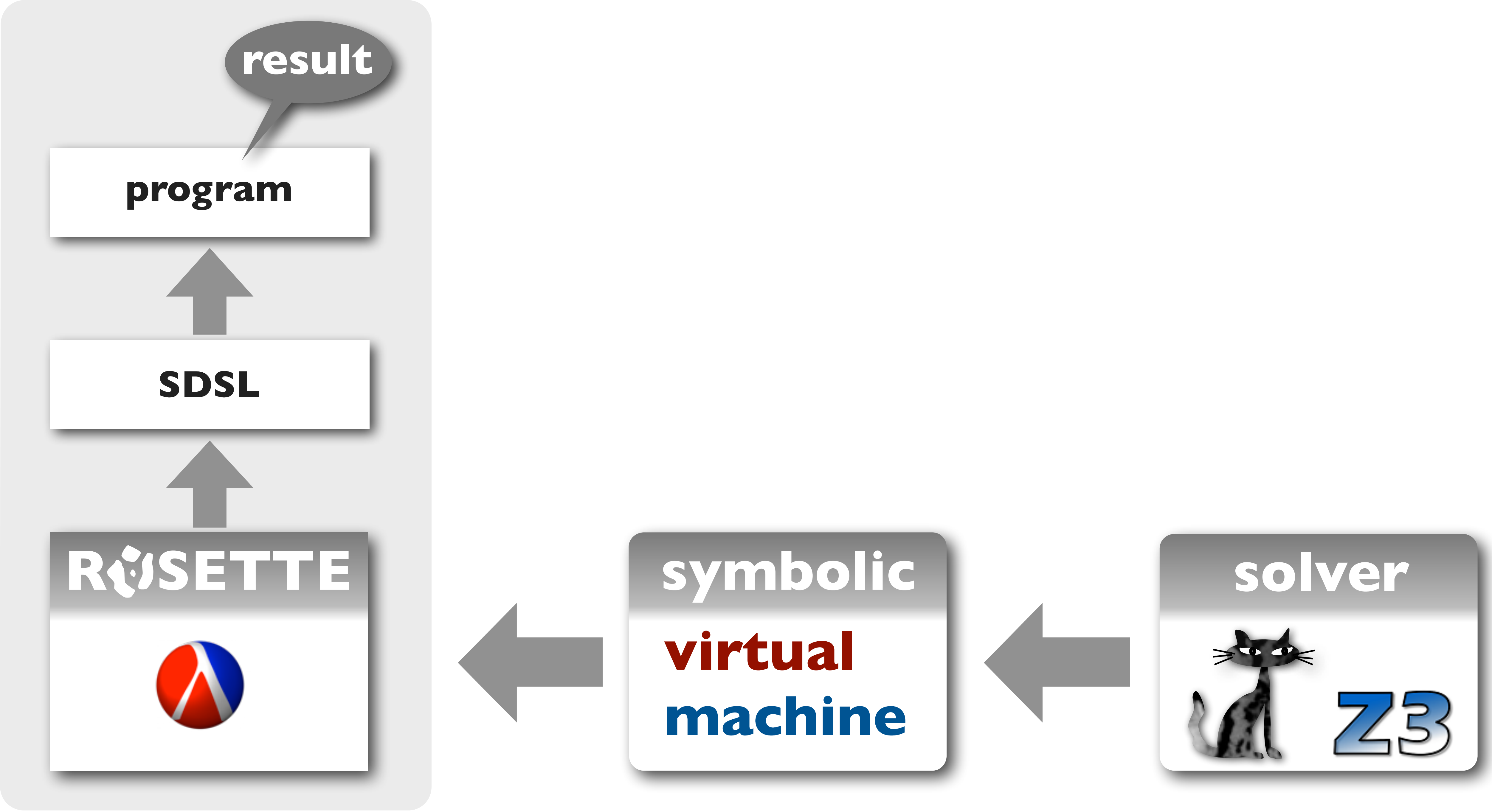
solver

Z3

[Torlak & Bodik, **Onward'13**]

[Torlak & Bodik, **PLDI'14**]

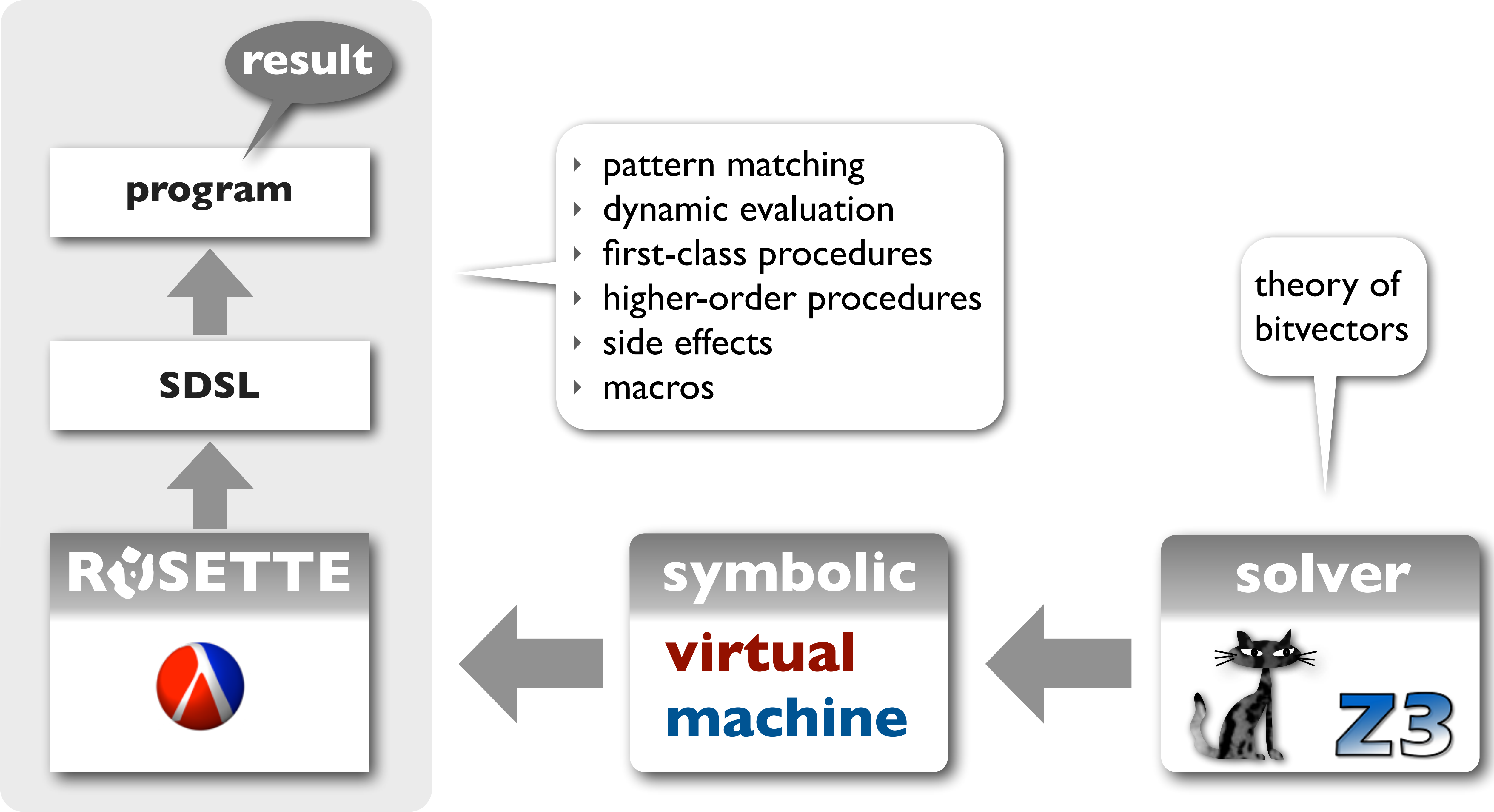
How it all works: a big picture view



[Torlak & Bodik, **Onward'13**]

[Torlak & Bodik, **PLDI'14**]

How it all works: a big picture view



result

program

SDSL

ROSETTE

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

symbolic

virtual machine

theory of bitvectors

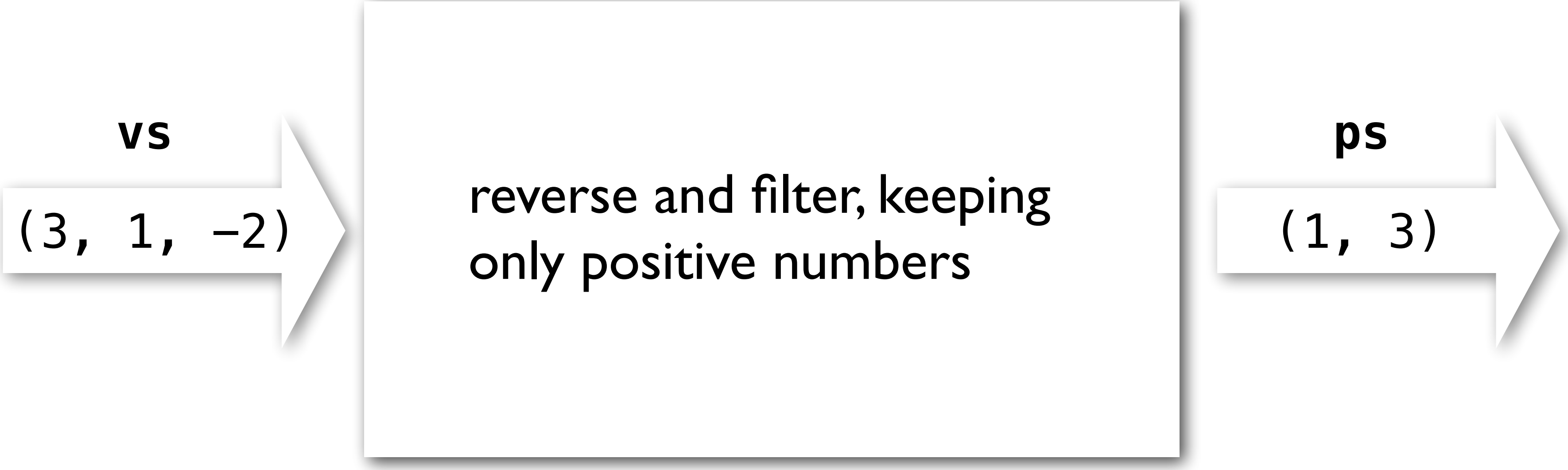
solver

Z3

[Torlak & Bodik, **Onward'13**]

[Torlak & Bodik, **PLDI'14**]

Translation to constraints by example



`vs`

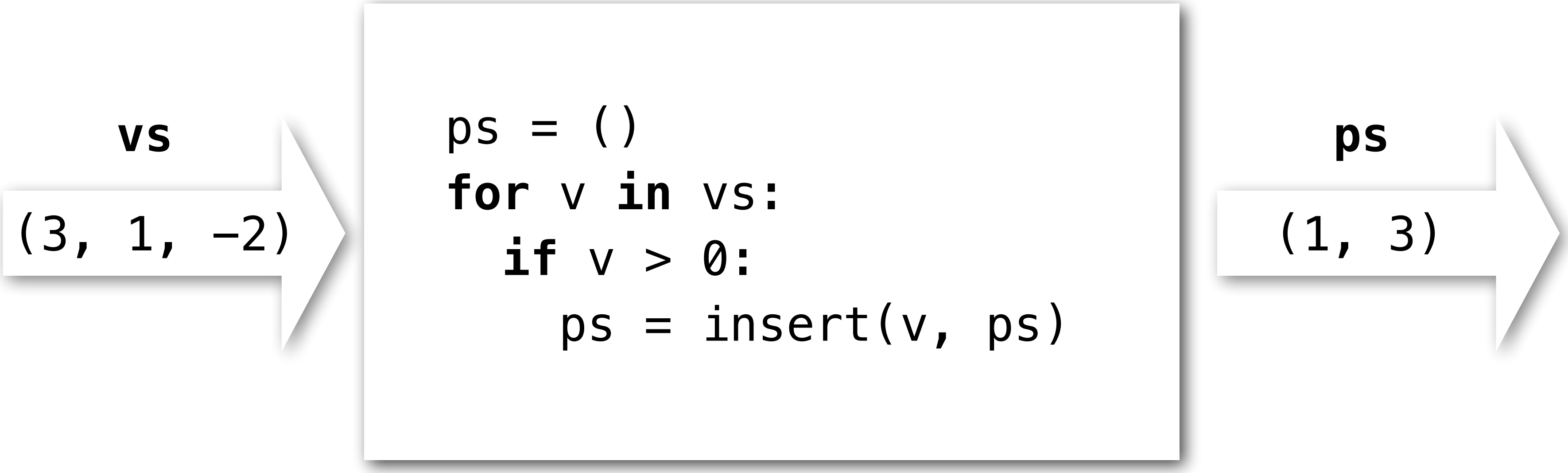
`(3, 1, −2)`

reverse and filter, keeping only positive numbers

`ps`

`(1, 3)`

Translation to constraints by example



vs

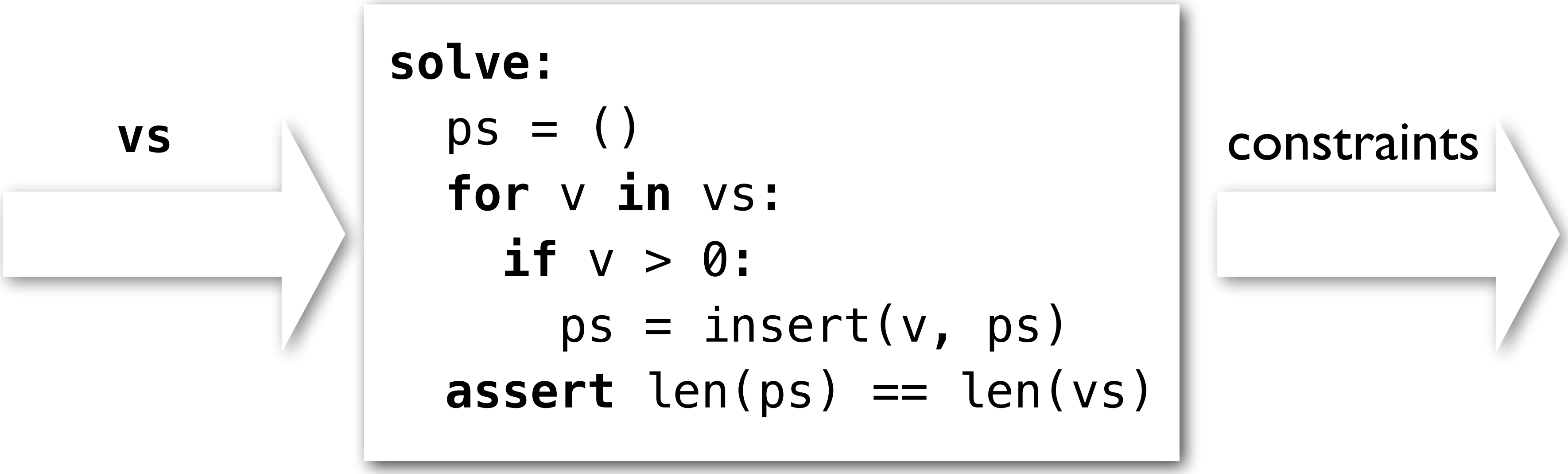
(3, 1, −2)

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

ps

(1, 3)

Translation to constraints by example

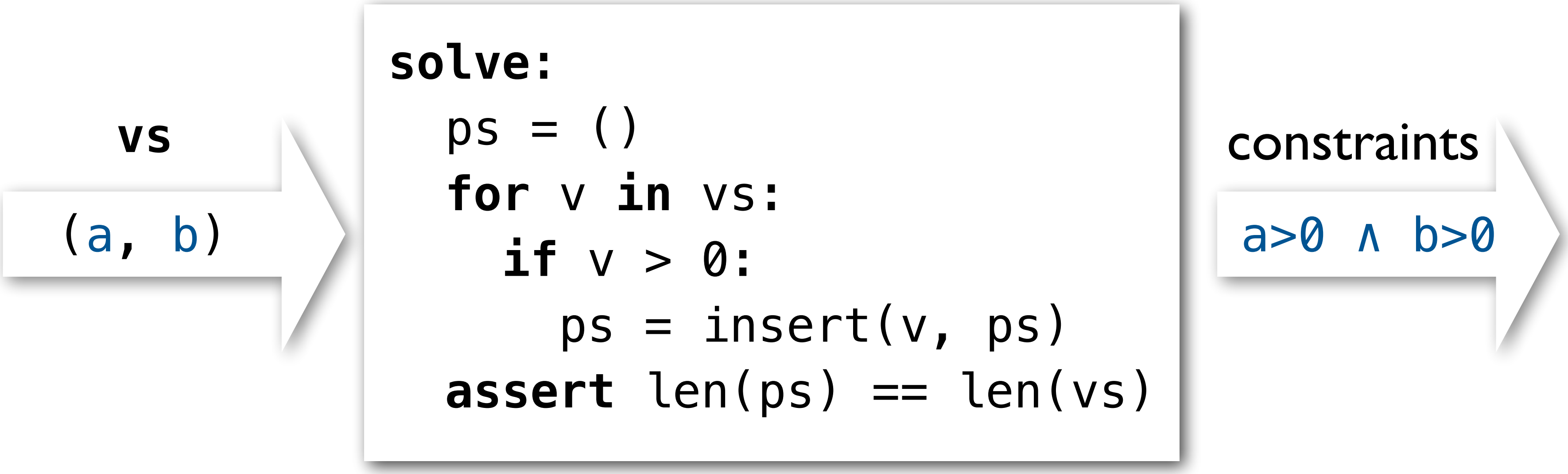


vs

```
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)

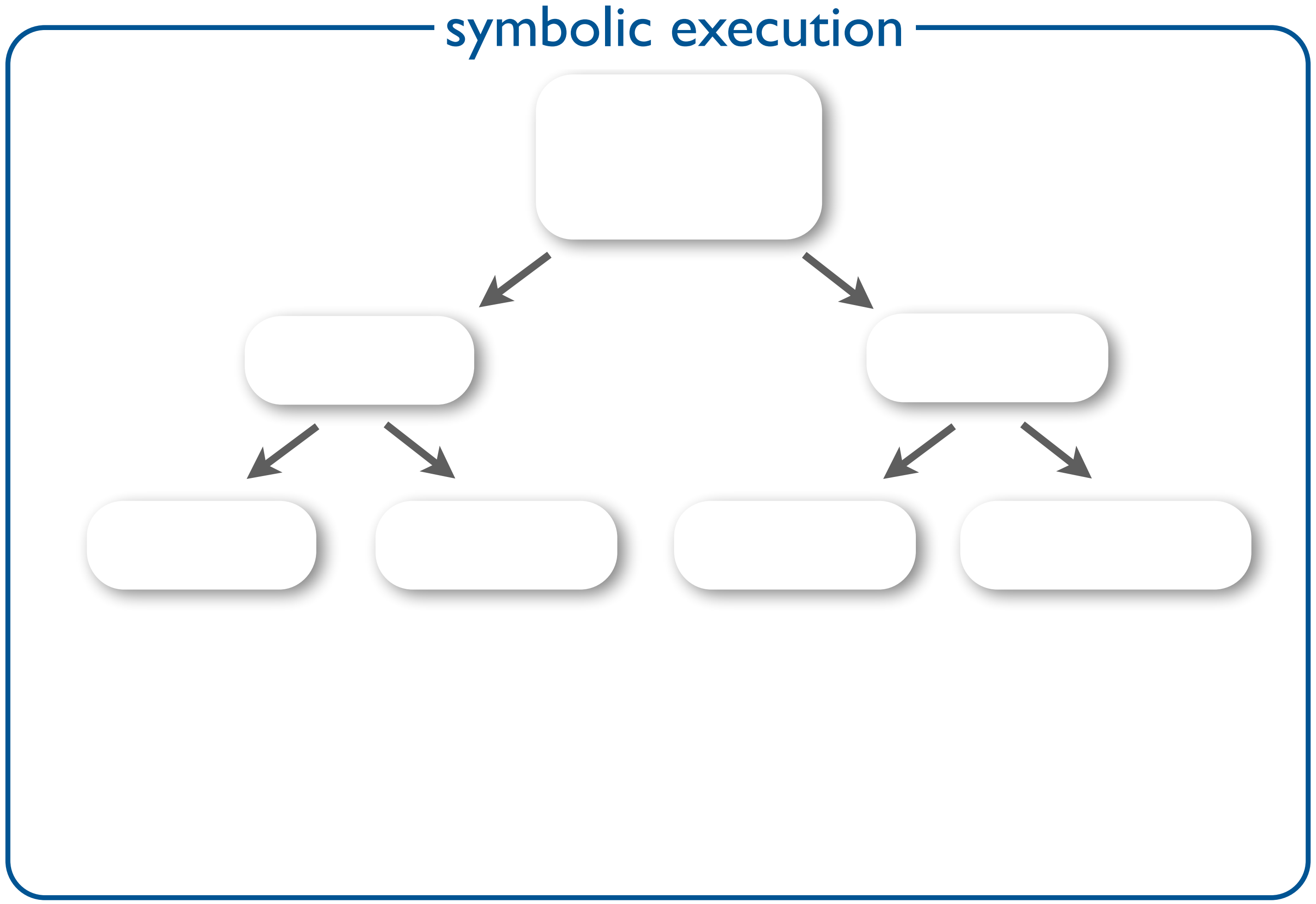
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

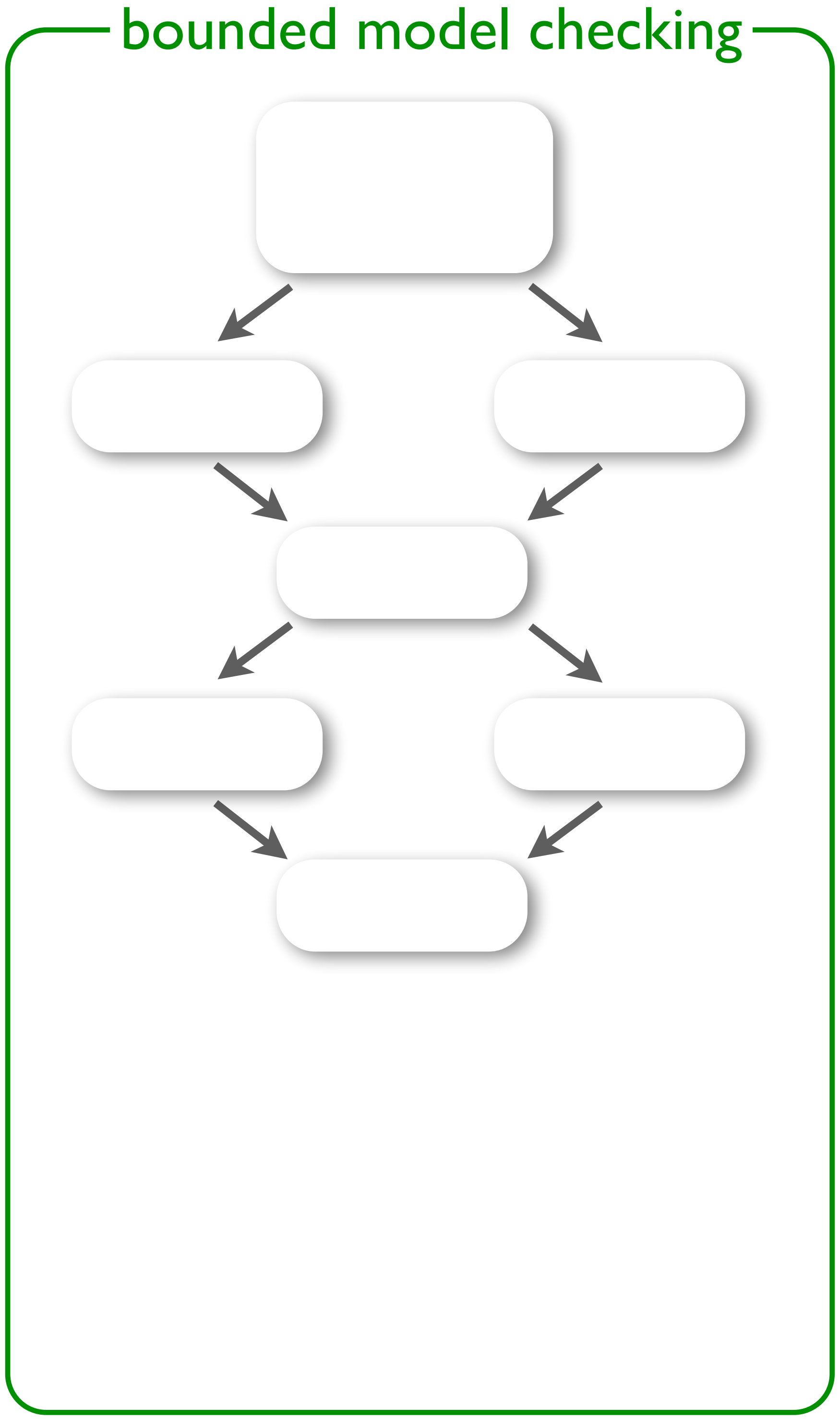
constraints

a>0 ∧ b>0

Design space of precise symbolic encodings

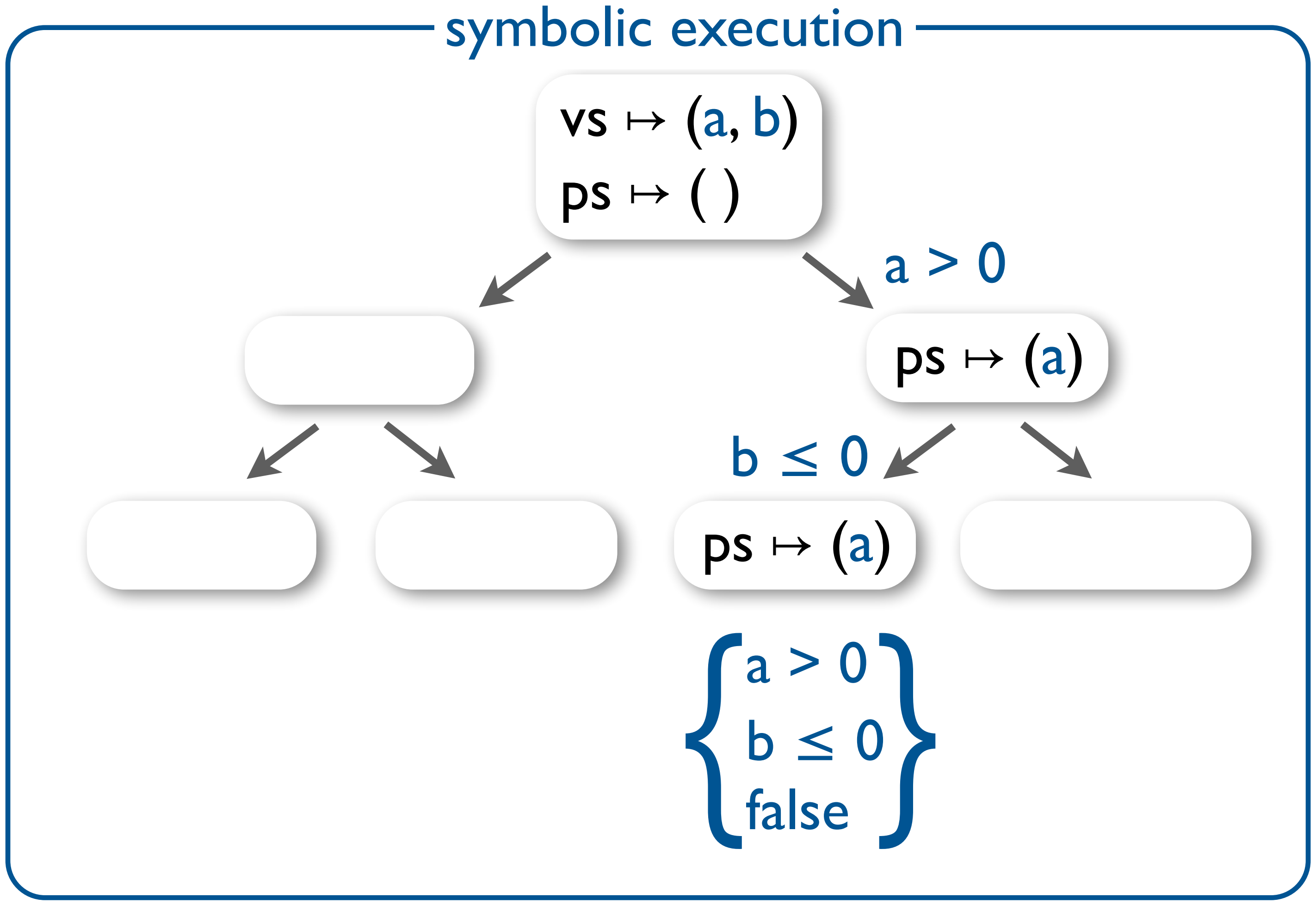
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```





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

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

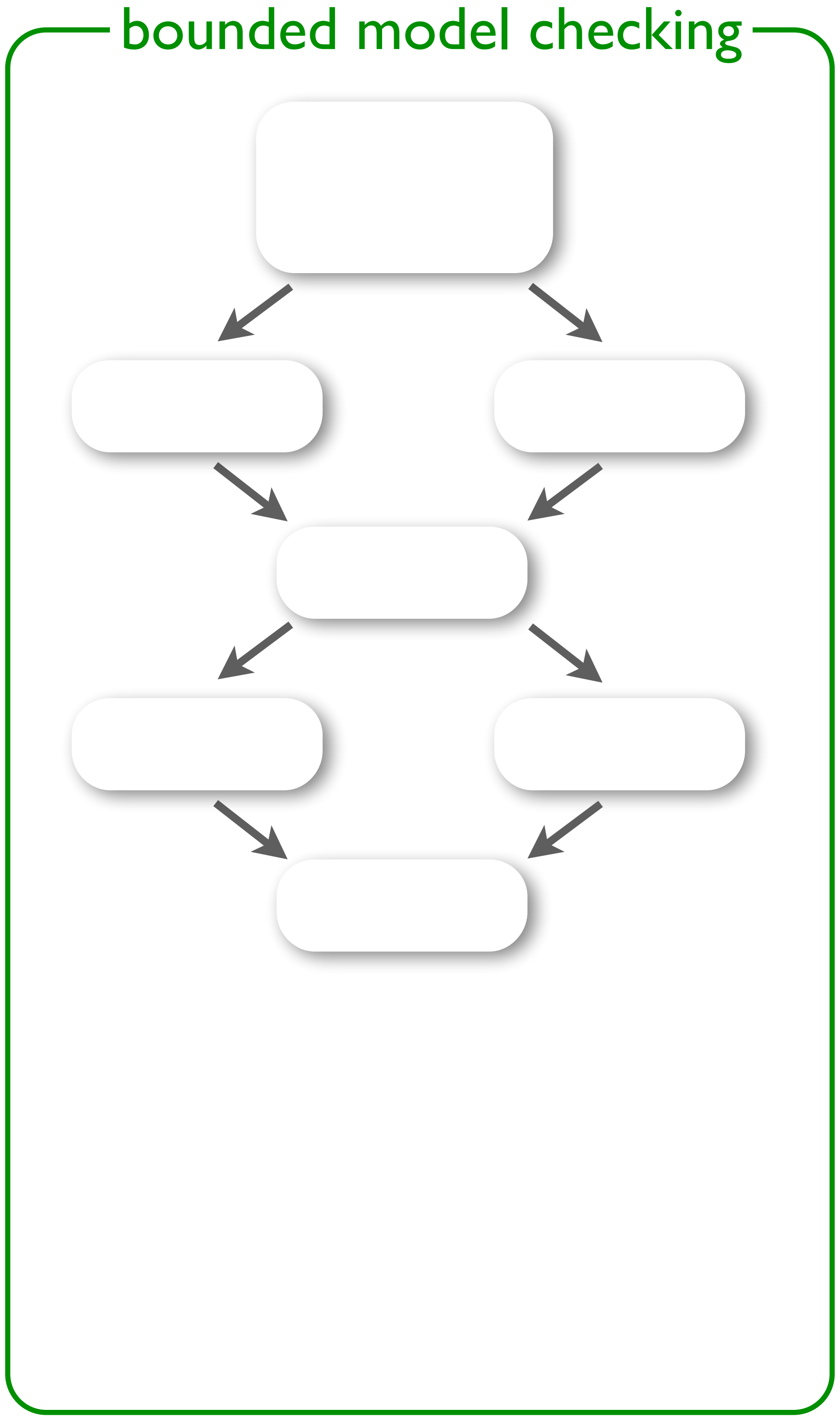
$a > 0$

$ps \mapsto (a)$

$b \leq 0$

$ps \mapsto (a)$

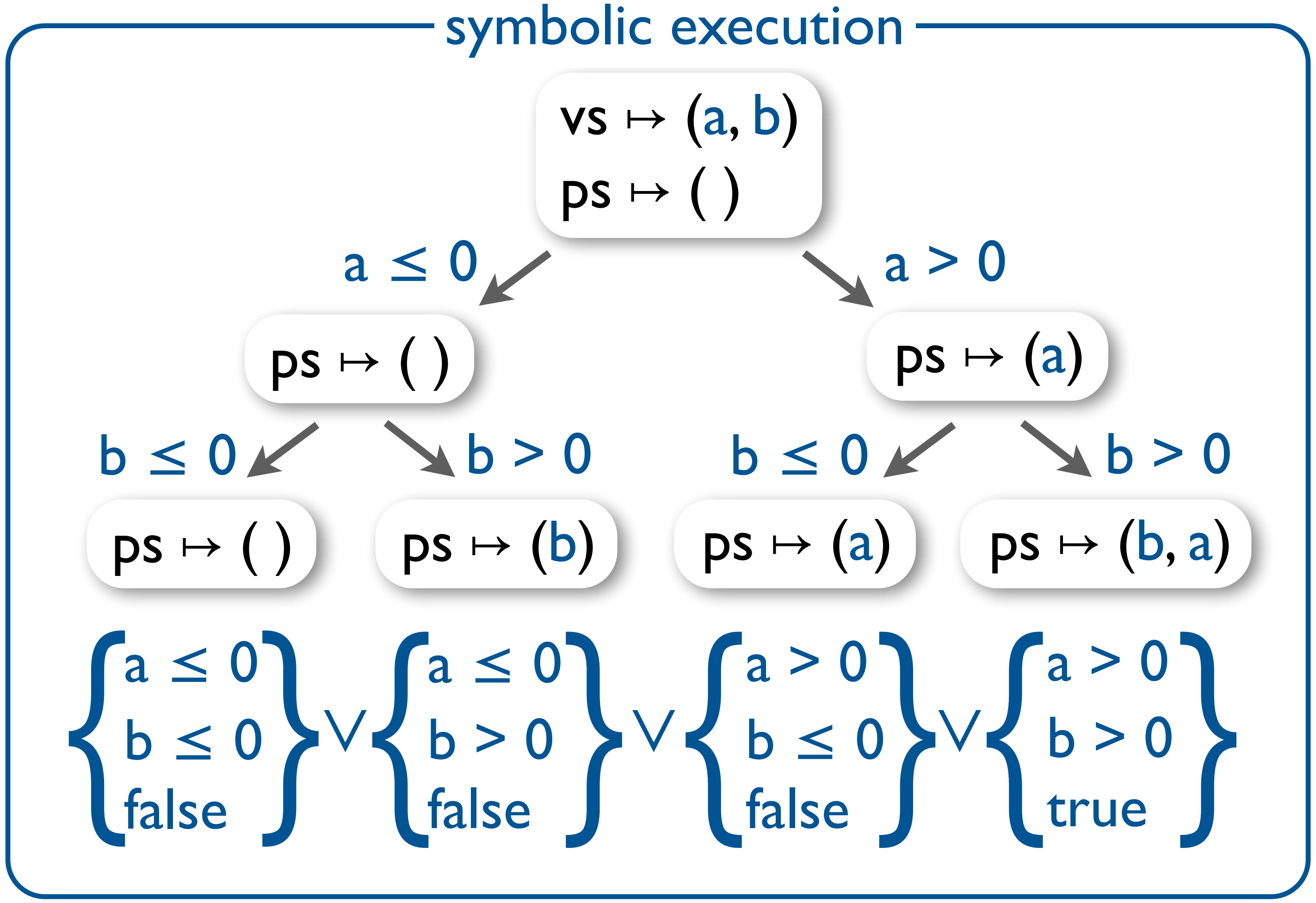
$$\left\{\begin{matrix} a > 0 \\ b \leq 0 \\ \text{false} \end{matrix}\right\}$$



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)
```



symbolic execution

vs ↦ (a, b)
ps ↦ ( )

$a \le 0$ → ps ↦ ( )

$a > 0$ → ps ↦ (a)

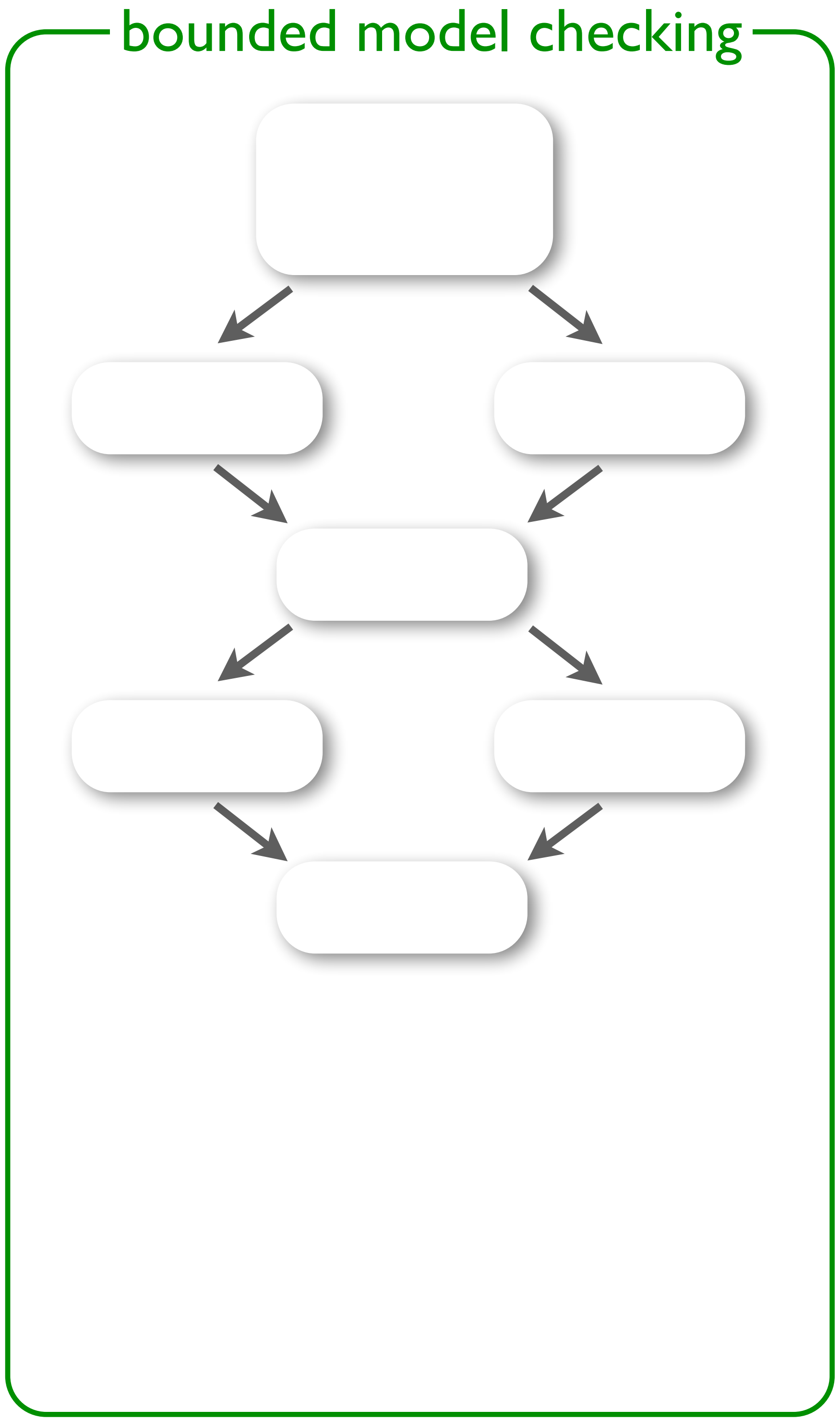
$b \le 0$ → ps ↦ ( )

$b > 0$ → ps ↦ (b)

$b \le 0$ → ps ↦ (a)

$b > 0$ → ps ↦ (b, a)

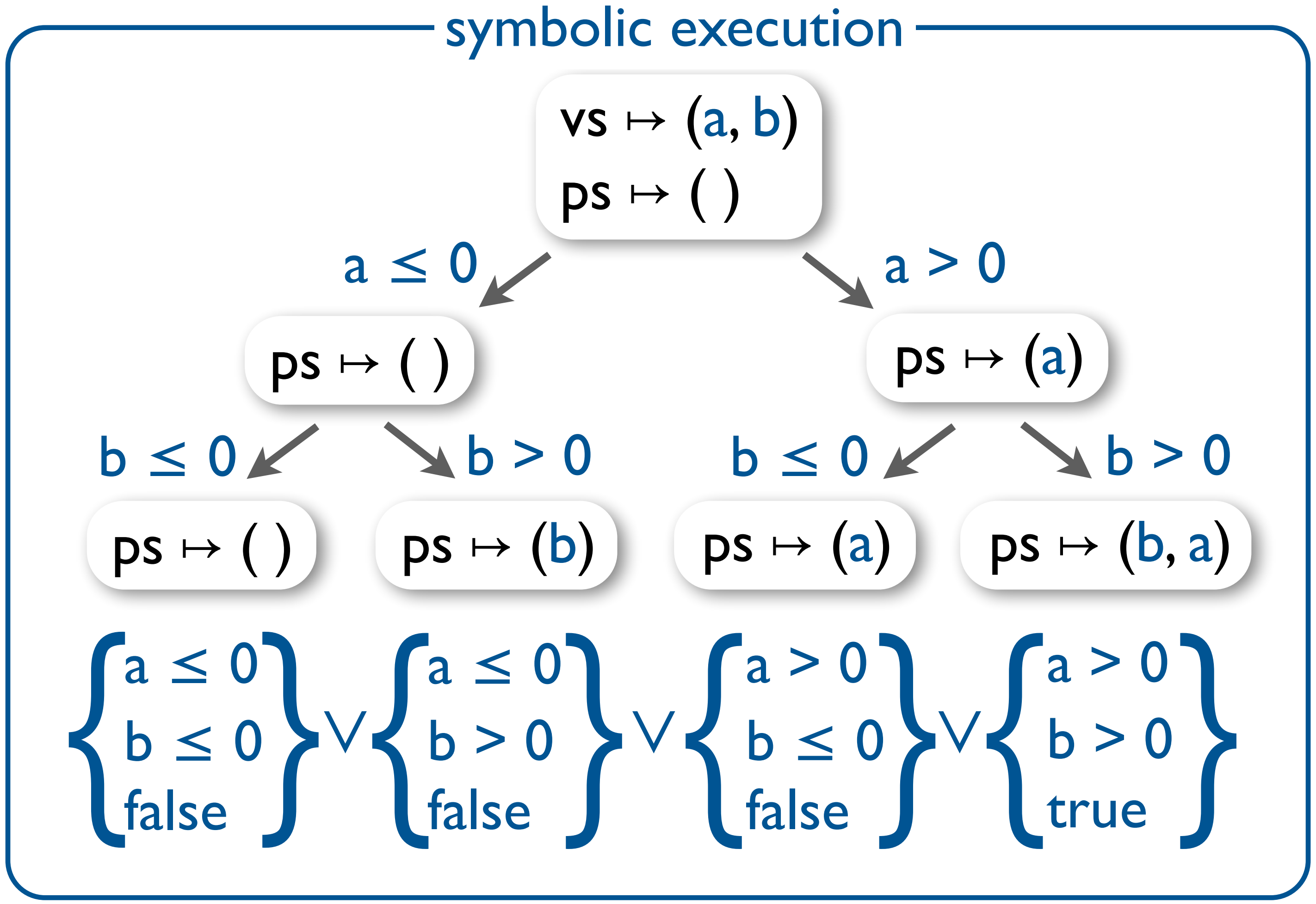
$$\begin{Bmatrix} a \le 0 \\ b \le 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a \le 0 \\ b > 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b \le 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b > 0 \\ \text{true} \end{Bmatrix}$$

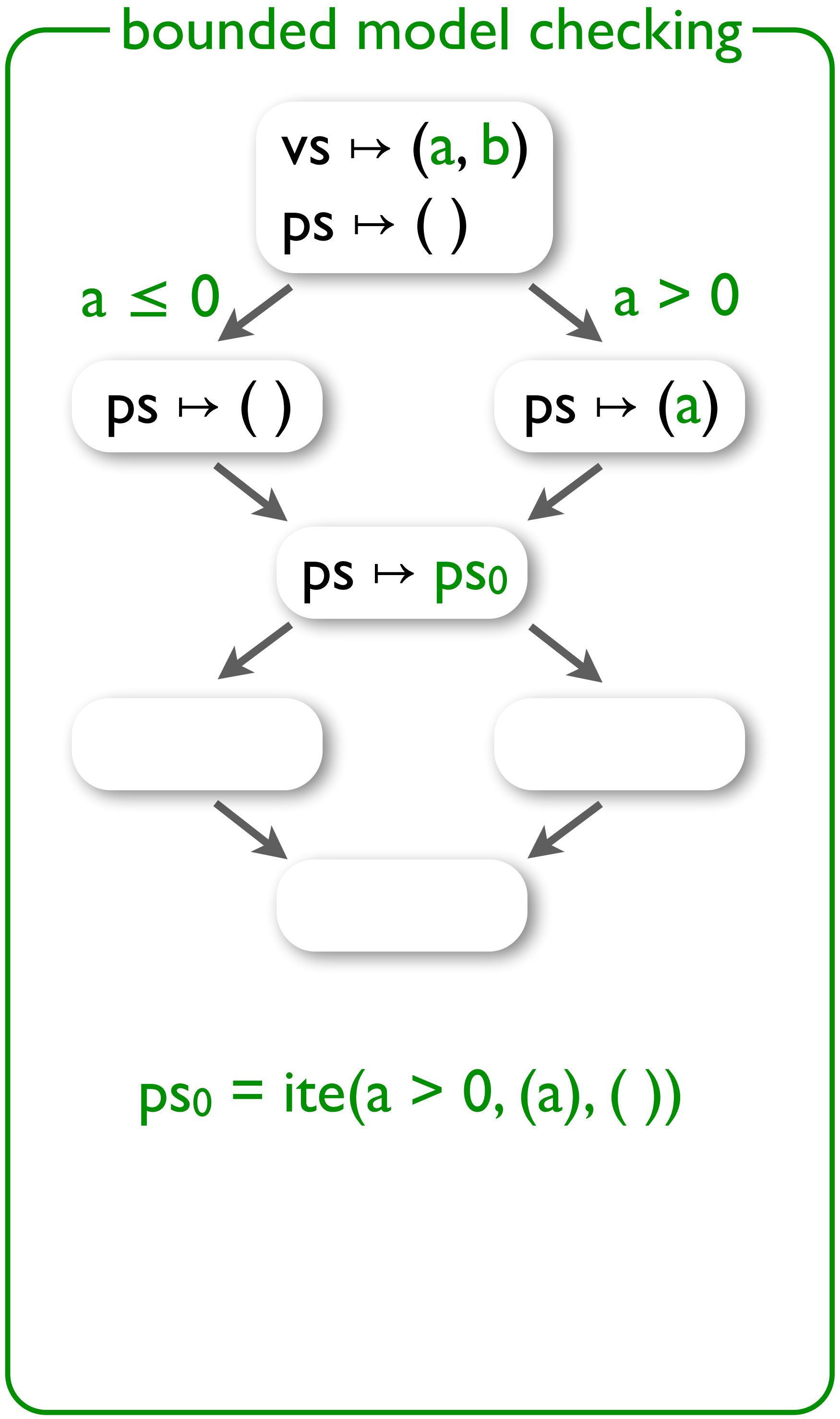


bounded model checking

Design space of precise symbolic encodings

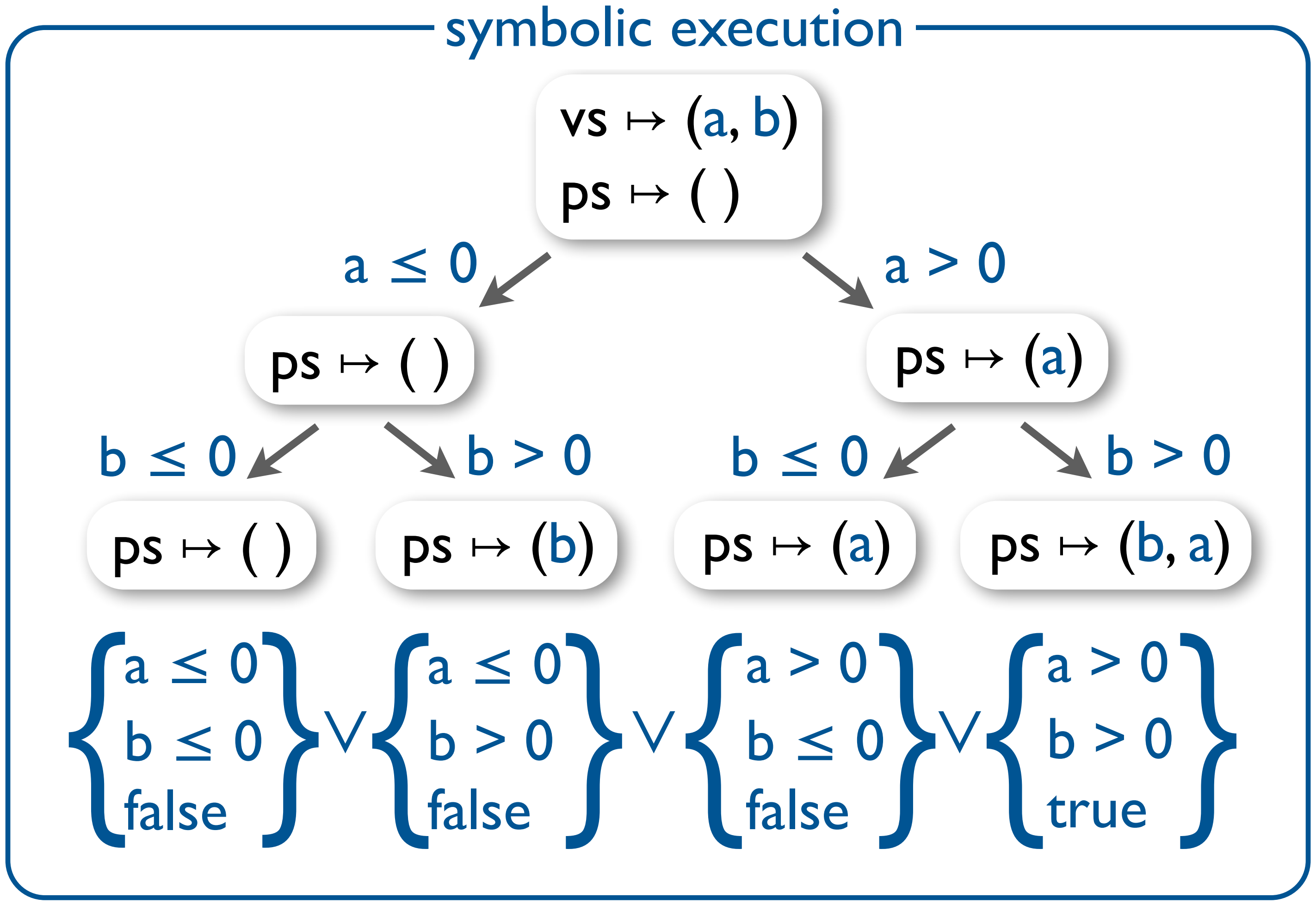
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```





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

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

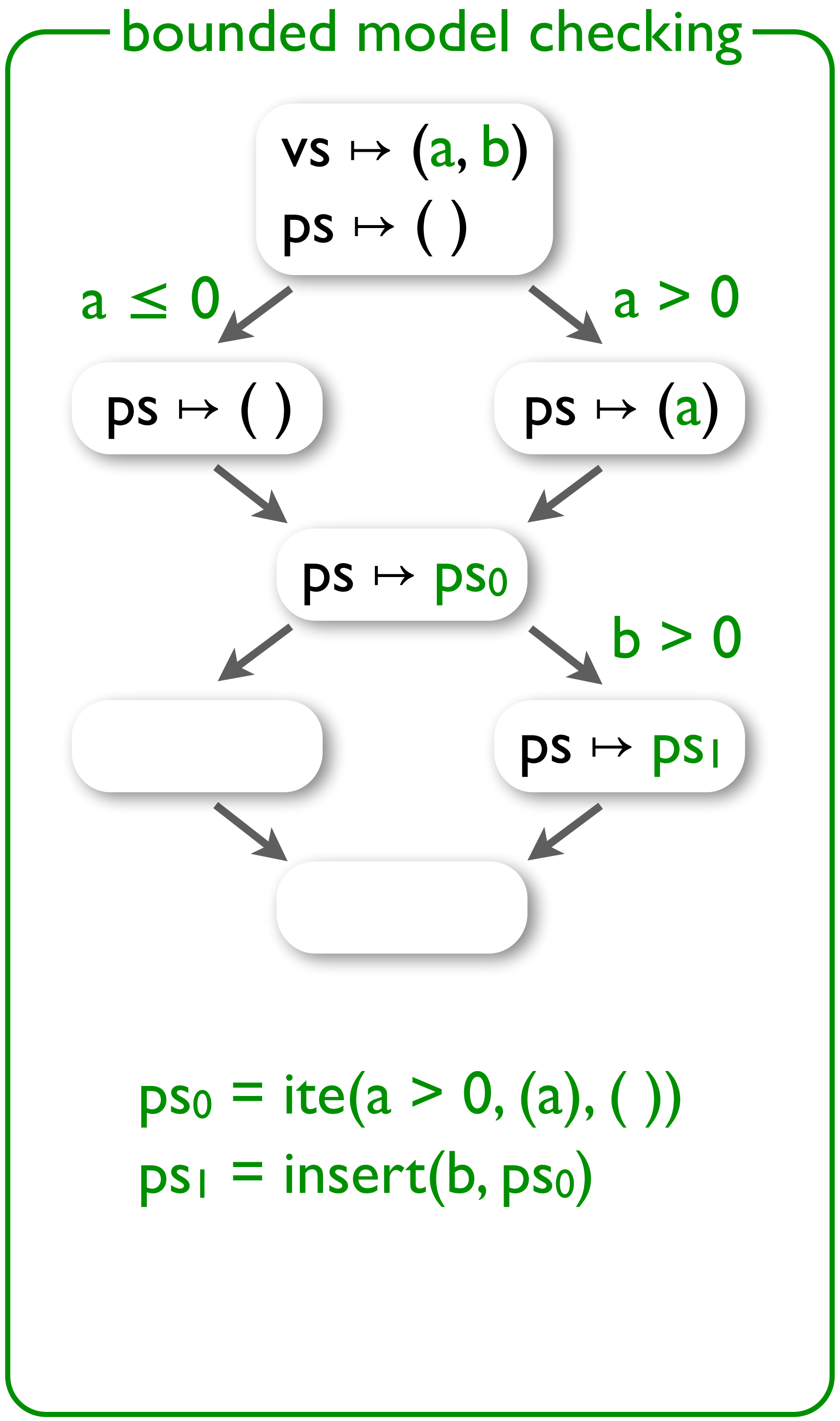
$a \le 0$ | $a > 0$

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

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

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

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



bounded model checking

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

$a \le 0$ | $a > 0$

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

$ps \mapsto ps_0$

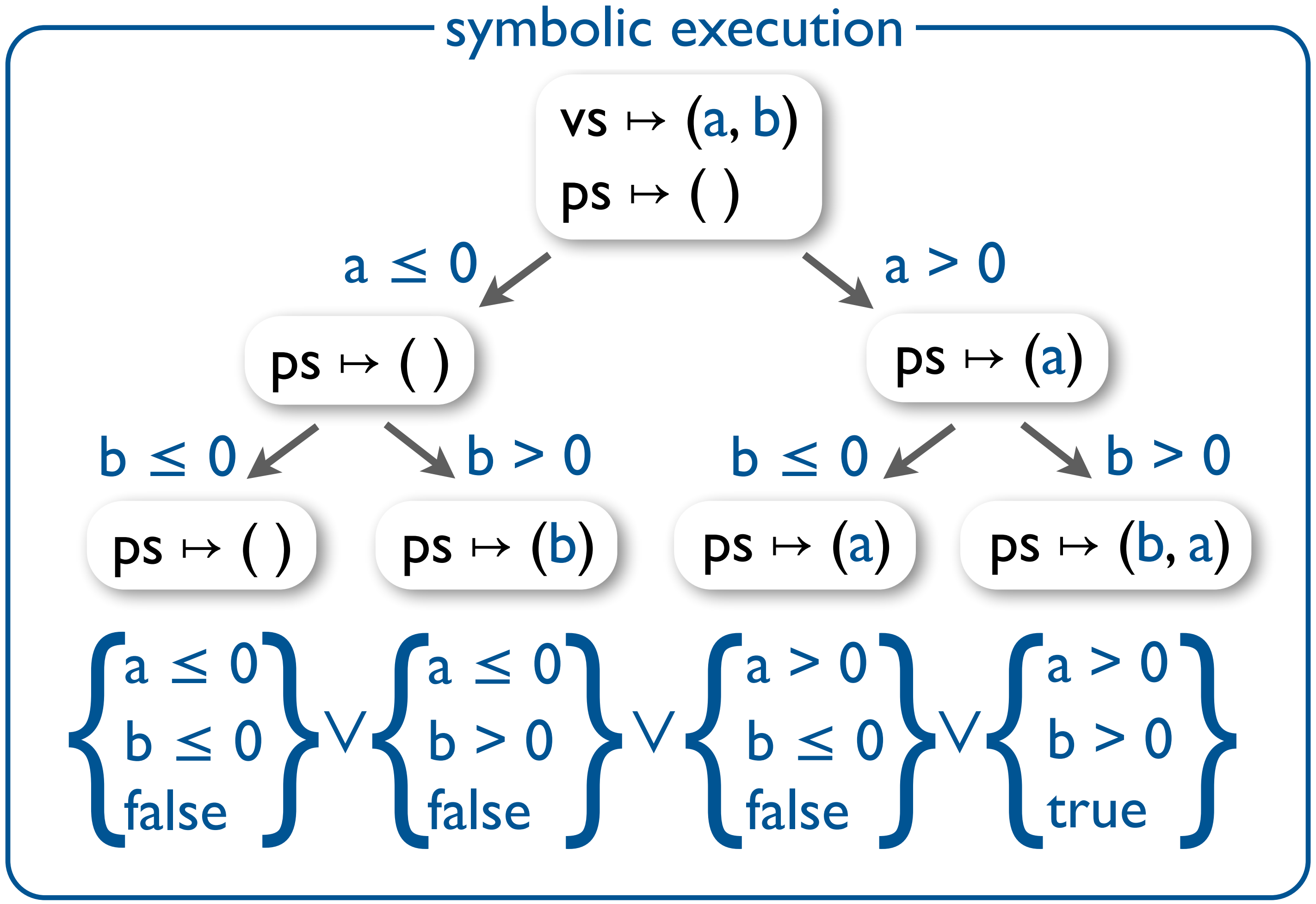
$b > 0$

$ps \mapsto ps_1$

$ps_0 = ite(a > 0, (a), (\,))$
$ps_1 = insert(b, ps_0)$

Design space of precise symbolic encodings

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



symbolic execution

vs ↦ (a, b)
ps ↦ ( )

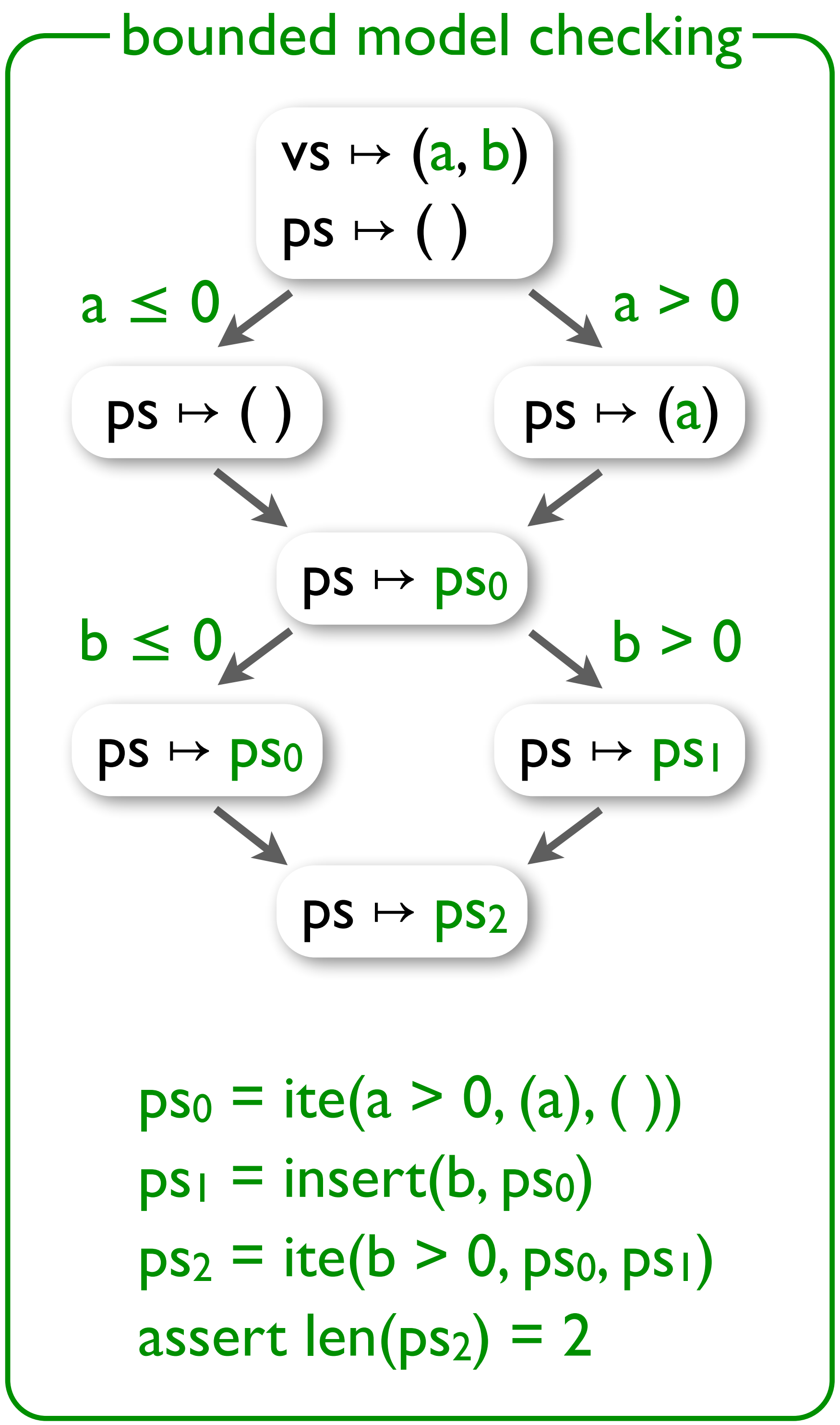
$a \le 0$ | $a > 0$

ps ↦ ( ) | ps ↦ (a)

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

ps ↦ ( ) | ps ↦ (b) | ps ↦ (a) | ps ↦ (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 ↦ (a, b)
ps ↦ ( )

$a \le 0$ | $a > 0$

ps ↦ ( ) | ps ↦ (a)

ps ↦ ps_0

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

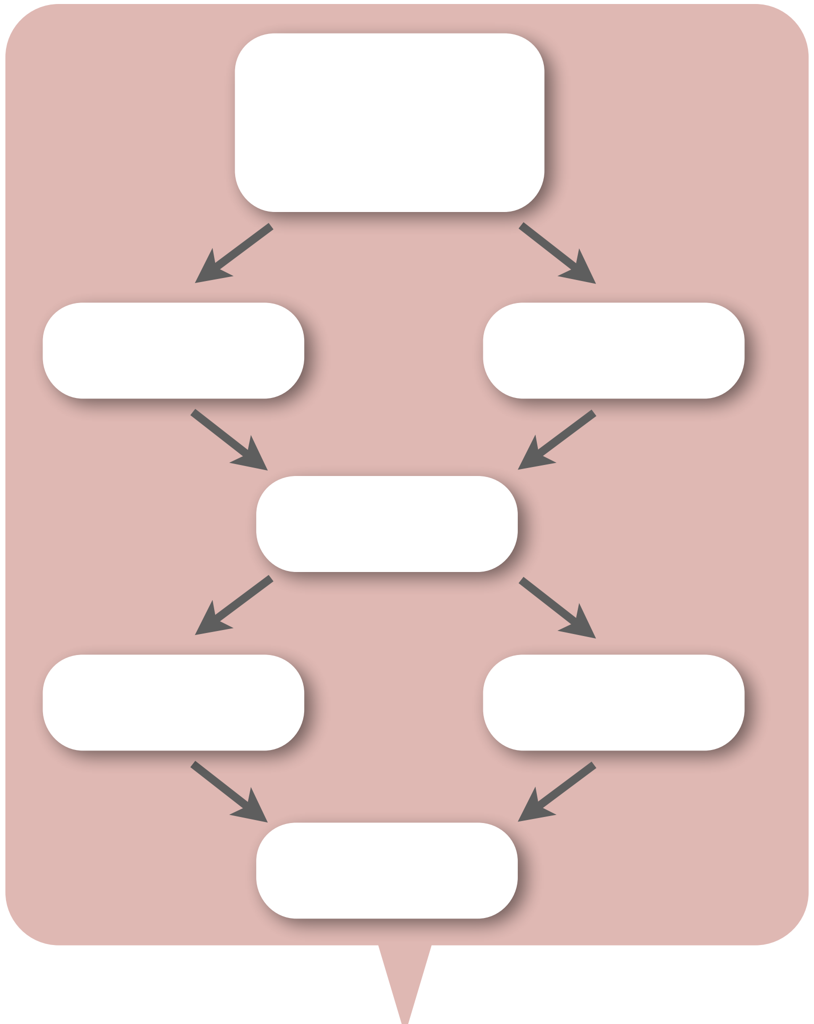
ps ↦ ps_0 | ps ↦ ps_1

ps ↦ ps_2

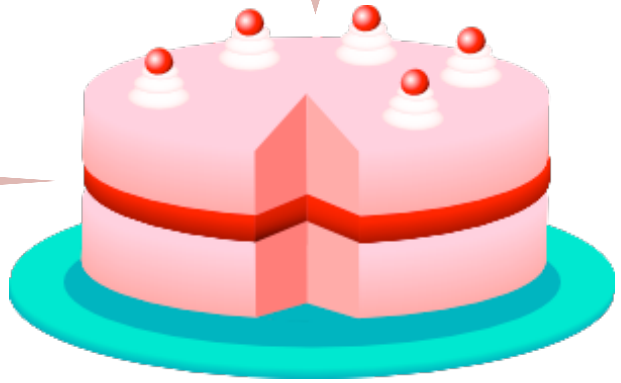
ps_0 = ite(a > 0, (a), ( ))
ps_1 = insert(b, ps_0)
ps_2 = ite(b > 0, ps_0, ps_1)
assert len(ps_2) = 2

A new design: type-driven state merging

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



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

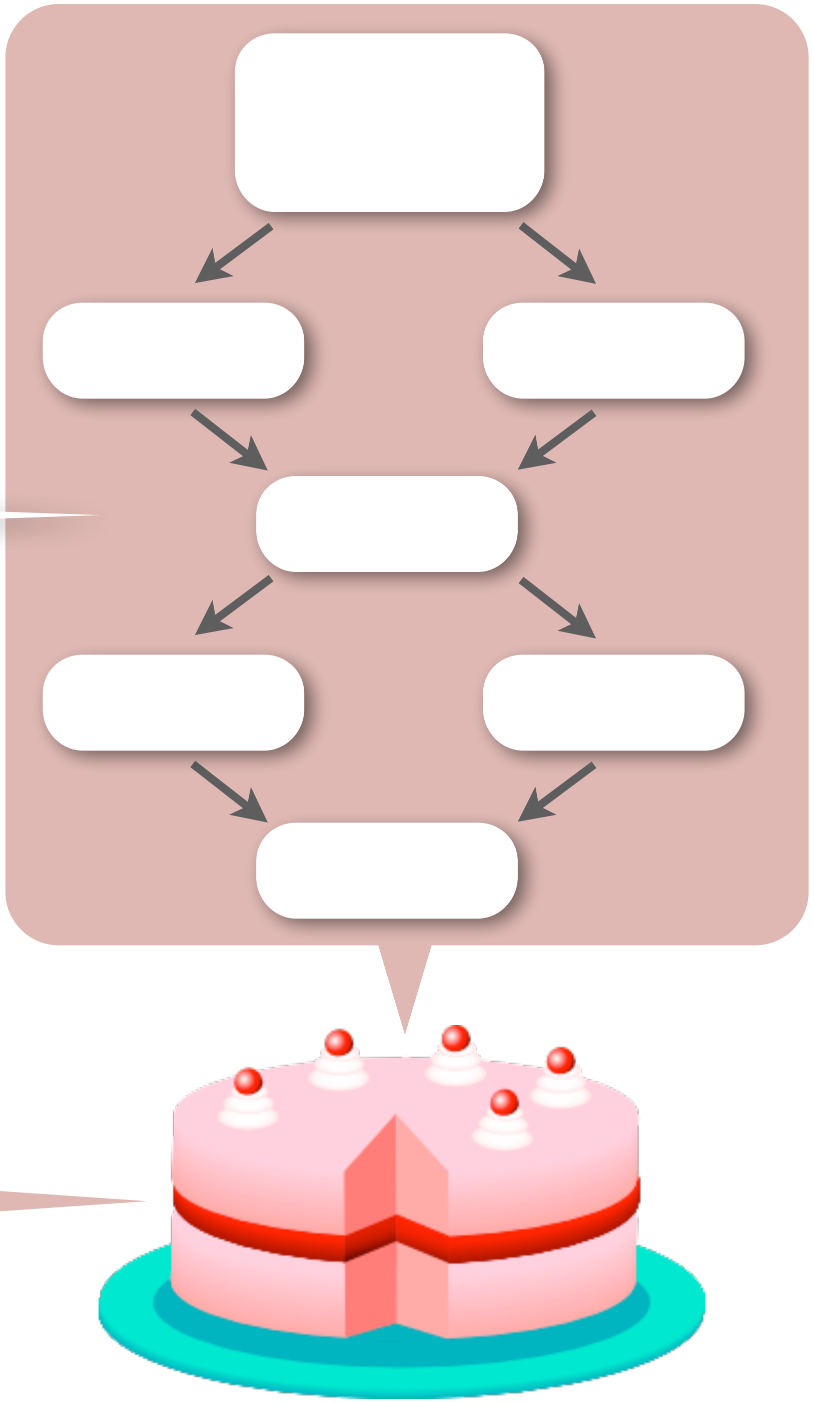


A new design: type-driven state merging

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

Merge values of

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



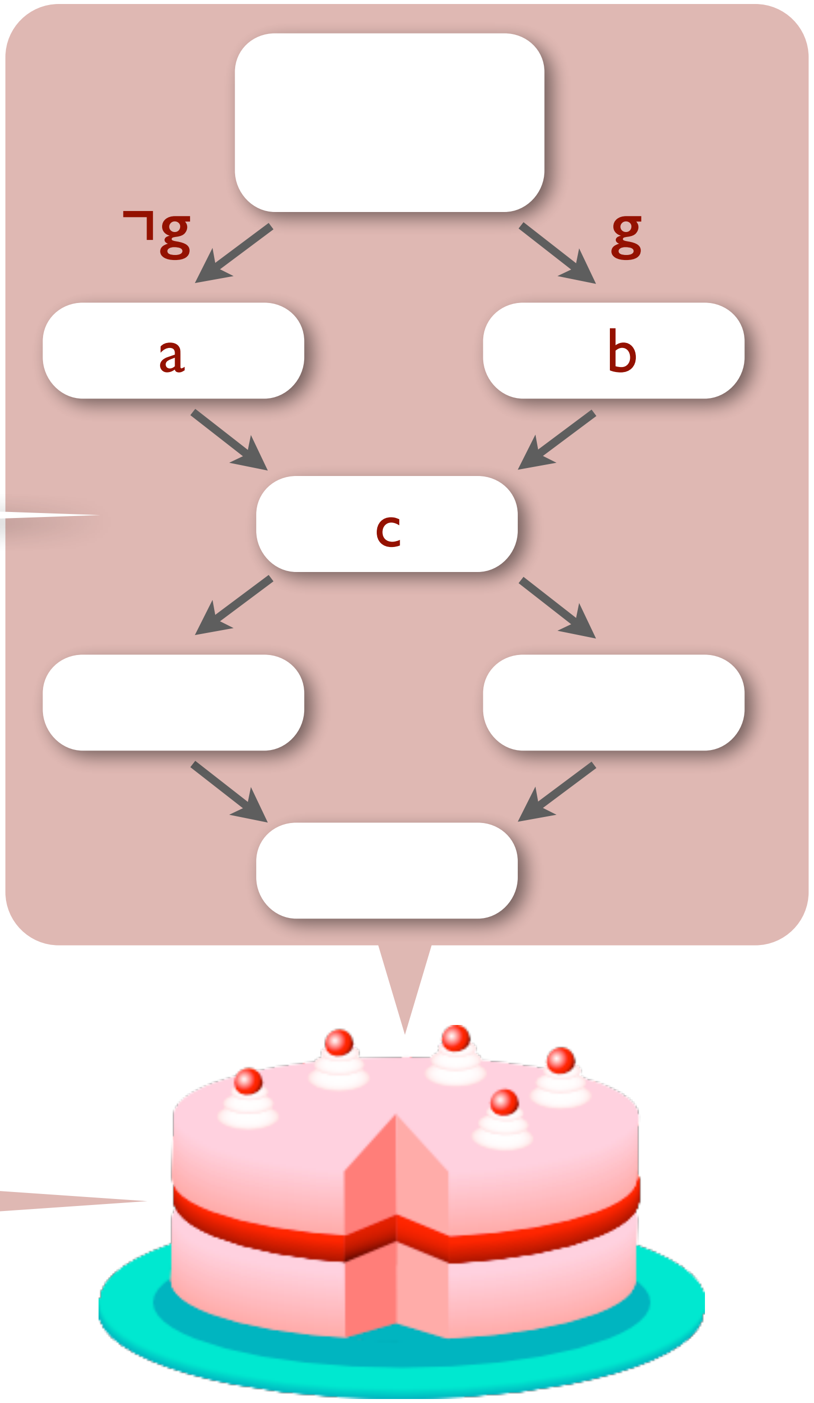
$$\left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$

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 values of

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



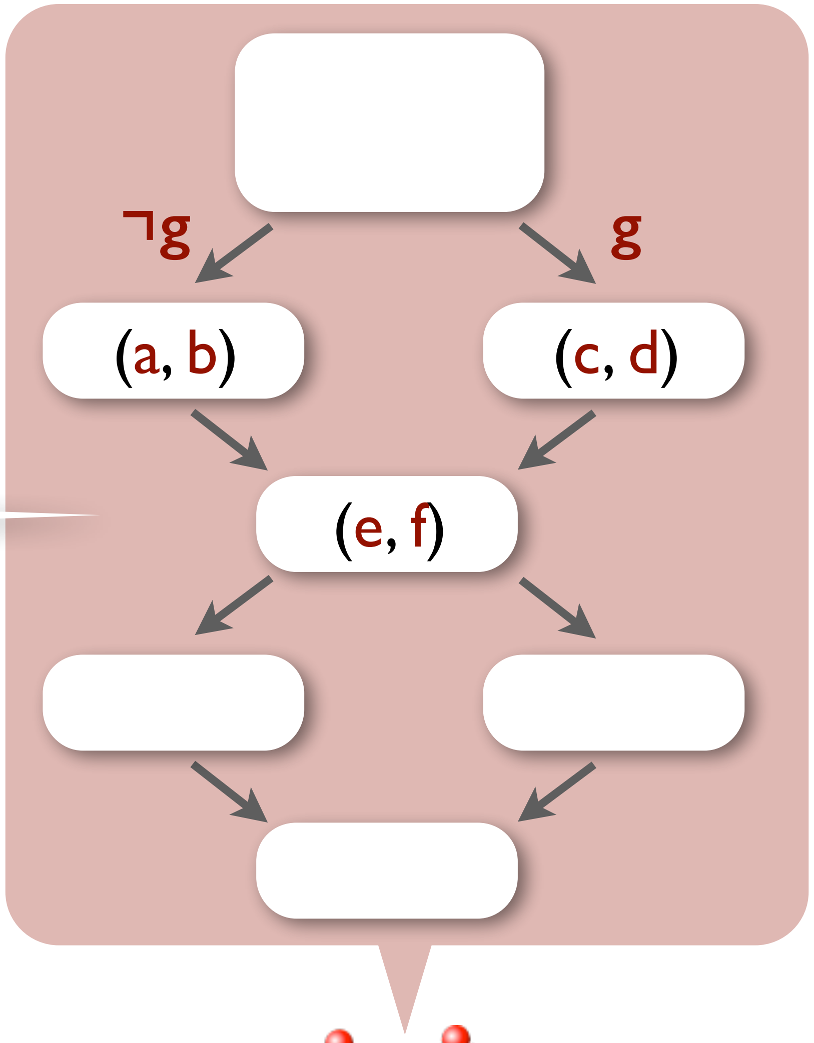
$$\left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$

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 values of

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



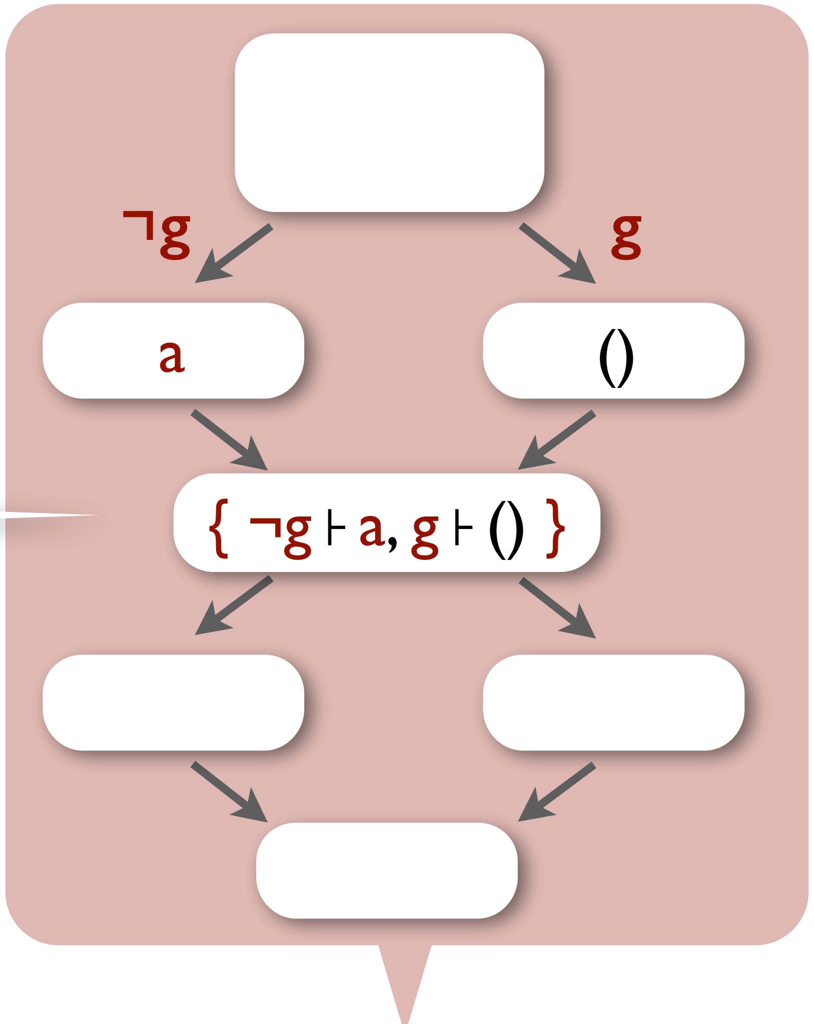
$$\left\{ \begin{array}{l} a > 0 \\ b > 0 \\ true \end{array} \right\}$$

A new design: type-driven state merging

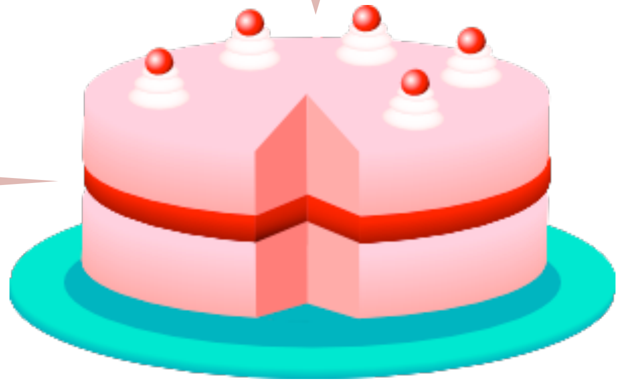
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

Merge values of

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

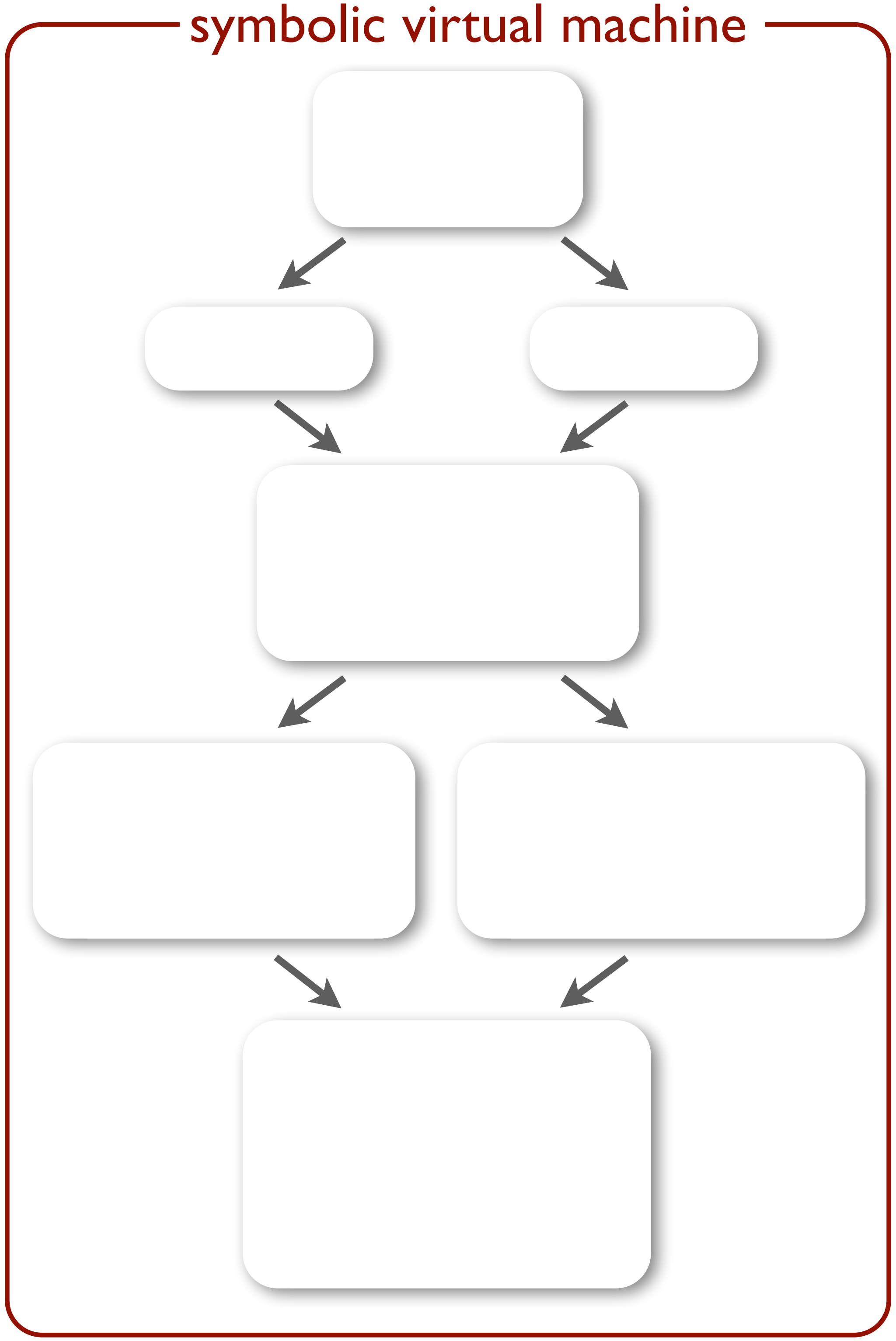


$$\left\{ \begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array} \right\}$$



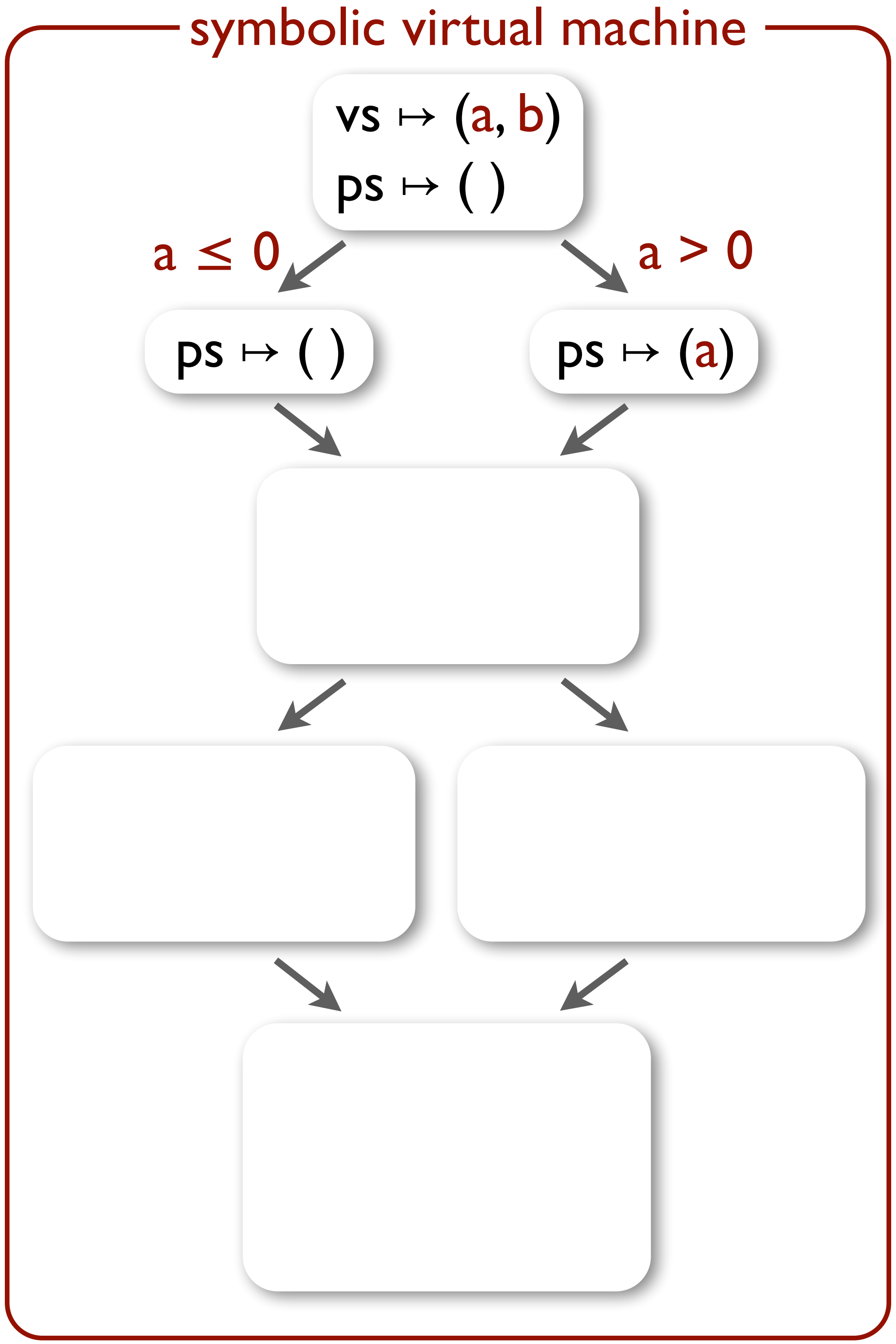
A new design: type-driven state merging

```
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)
```

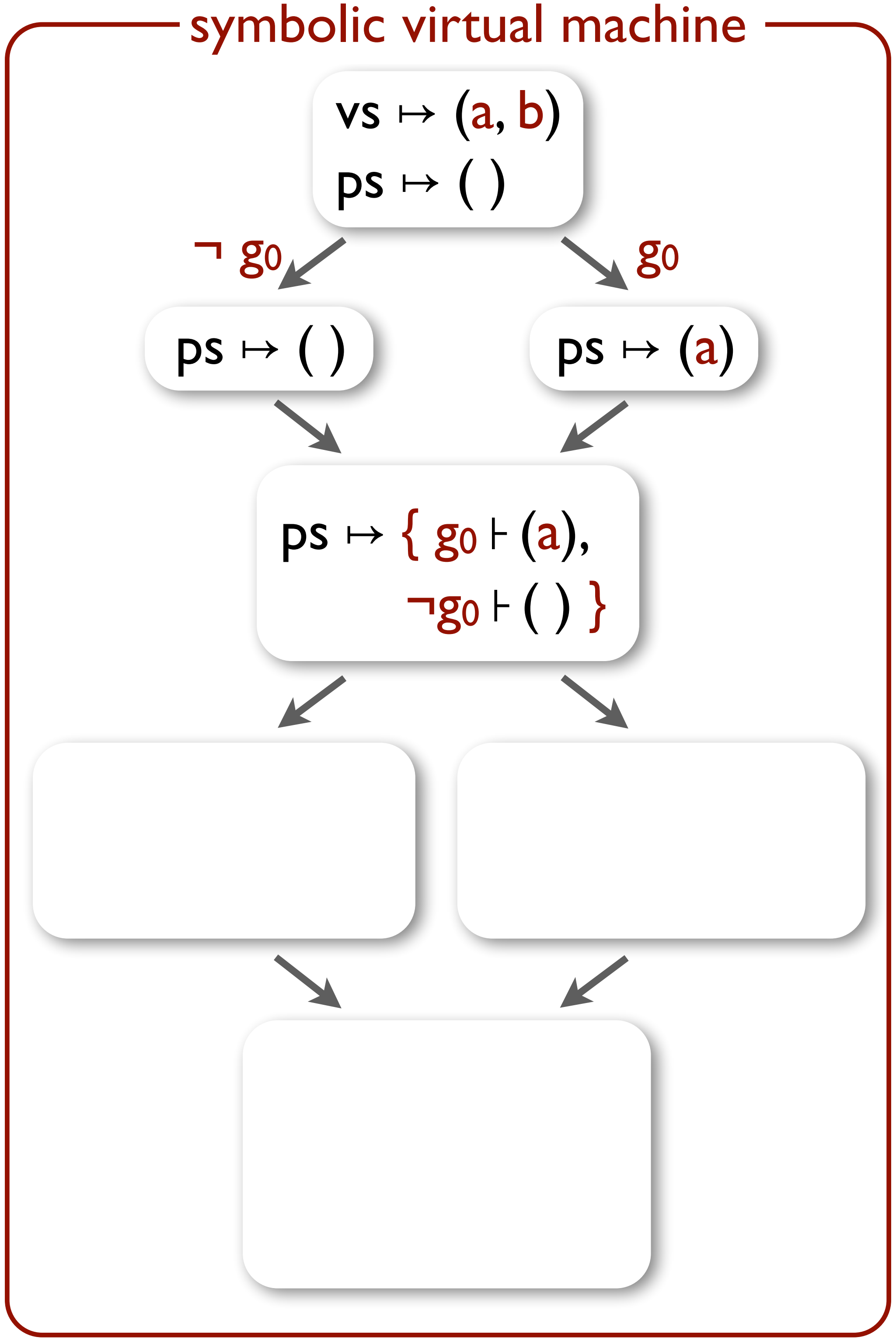


A new design: type-driven state merging

```
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.

$g_0 = a > 0$



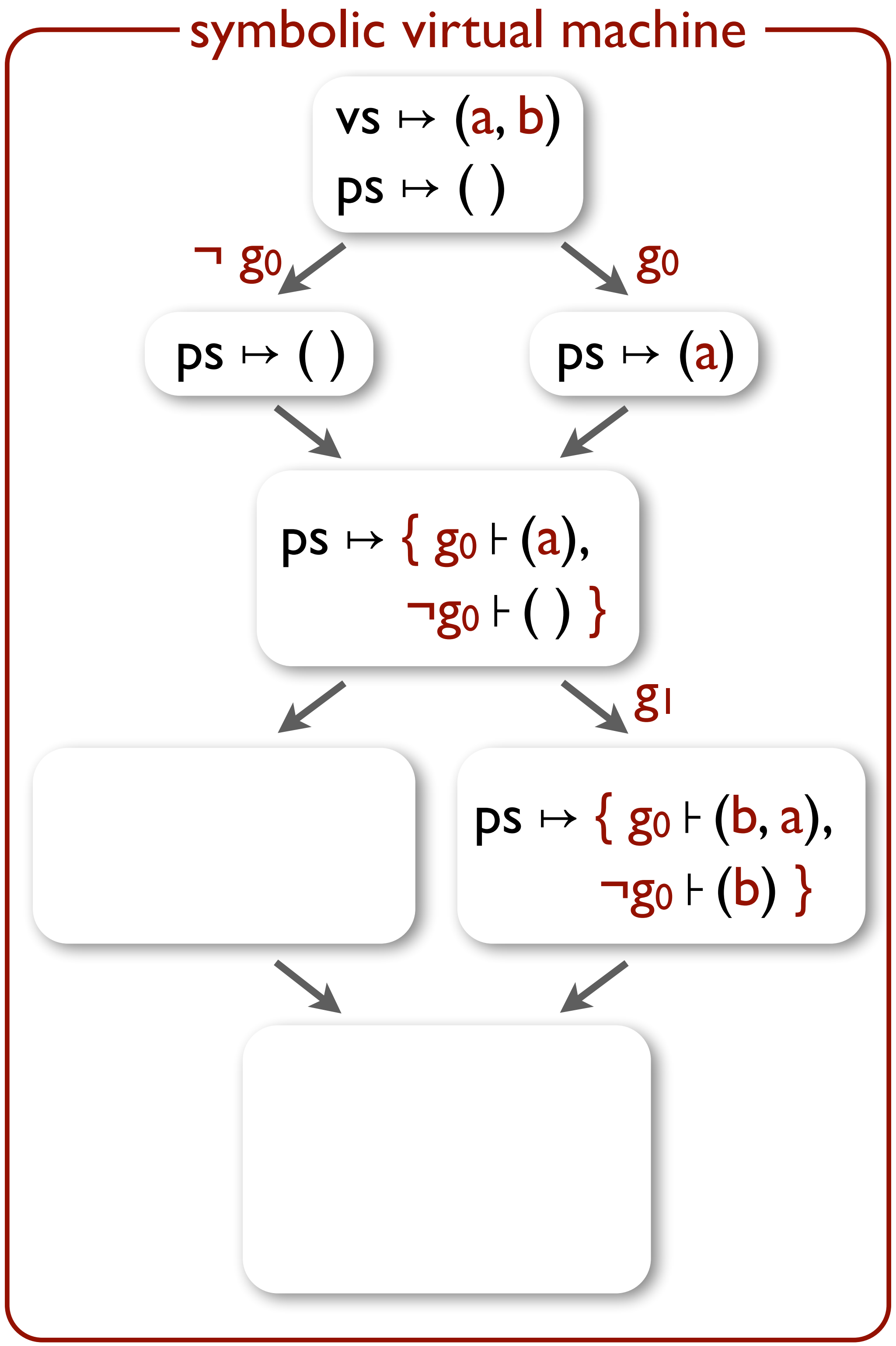
A new design: type-driven state merging

```
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.

$g_0 = a > 0$

$g_1 = b > 0$

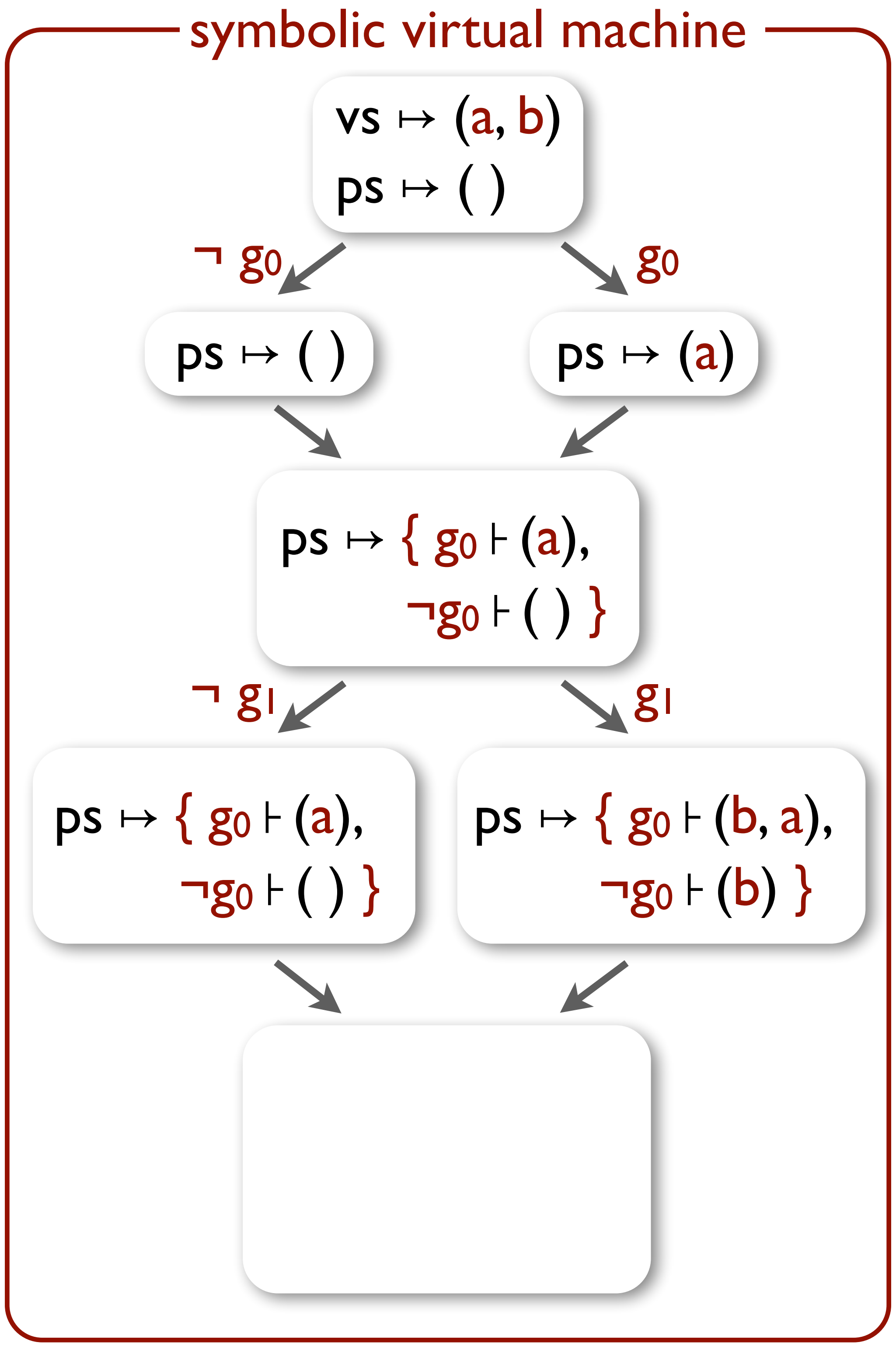


A new design: type-driven state merging

```
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)
```

$g_0 = a > 0$

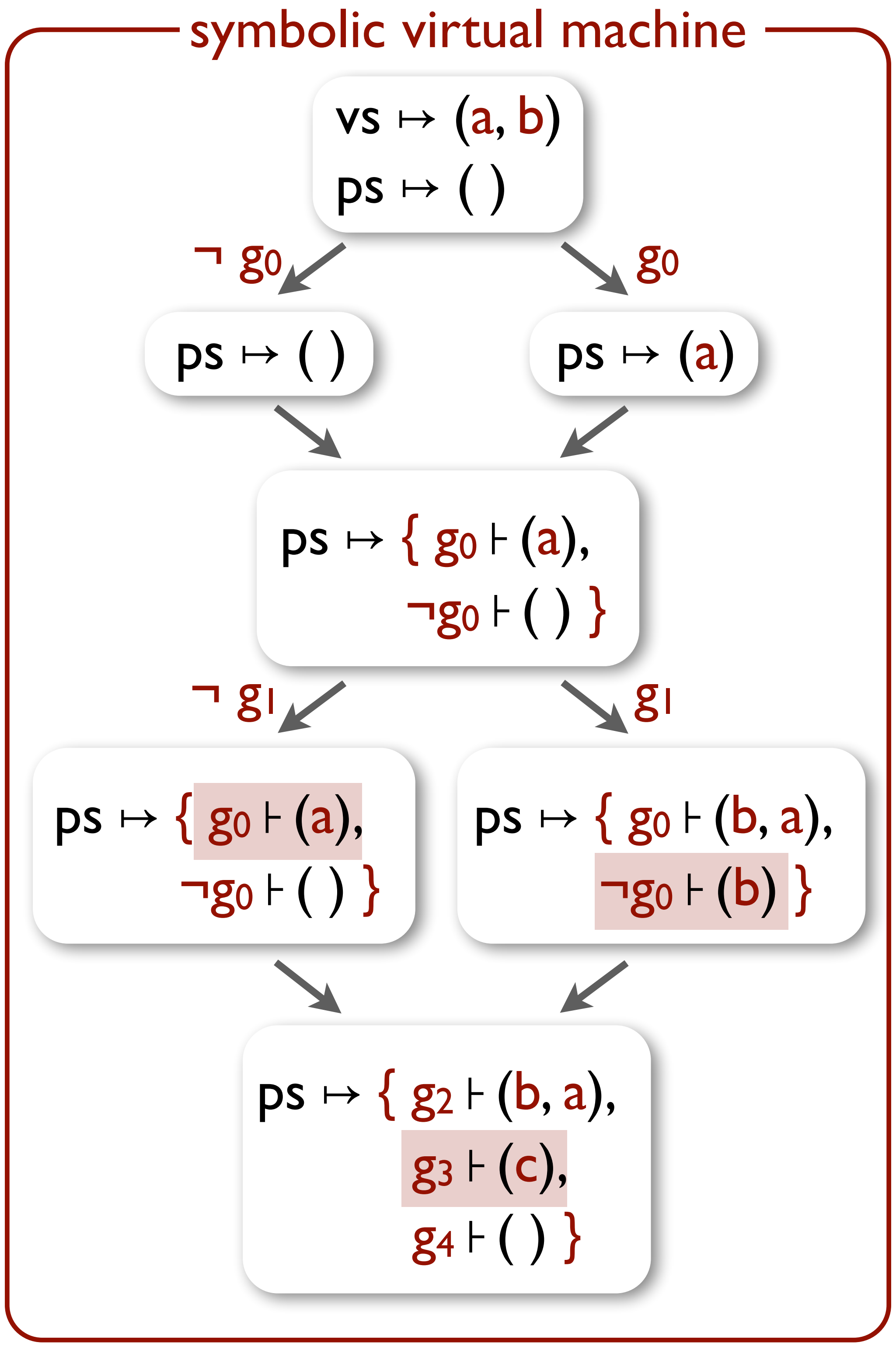
$g_1 = b > 0$

$g_2 = g_0 \wedge g_1$

$g_3 = \neg(g_0 \Leftrightarrow g_1)$

$g_4 = \neg g_0 \wedge \neg g_1$

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



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)
```

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

$g_0 = a > 0$

$g_1 = b > 0$

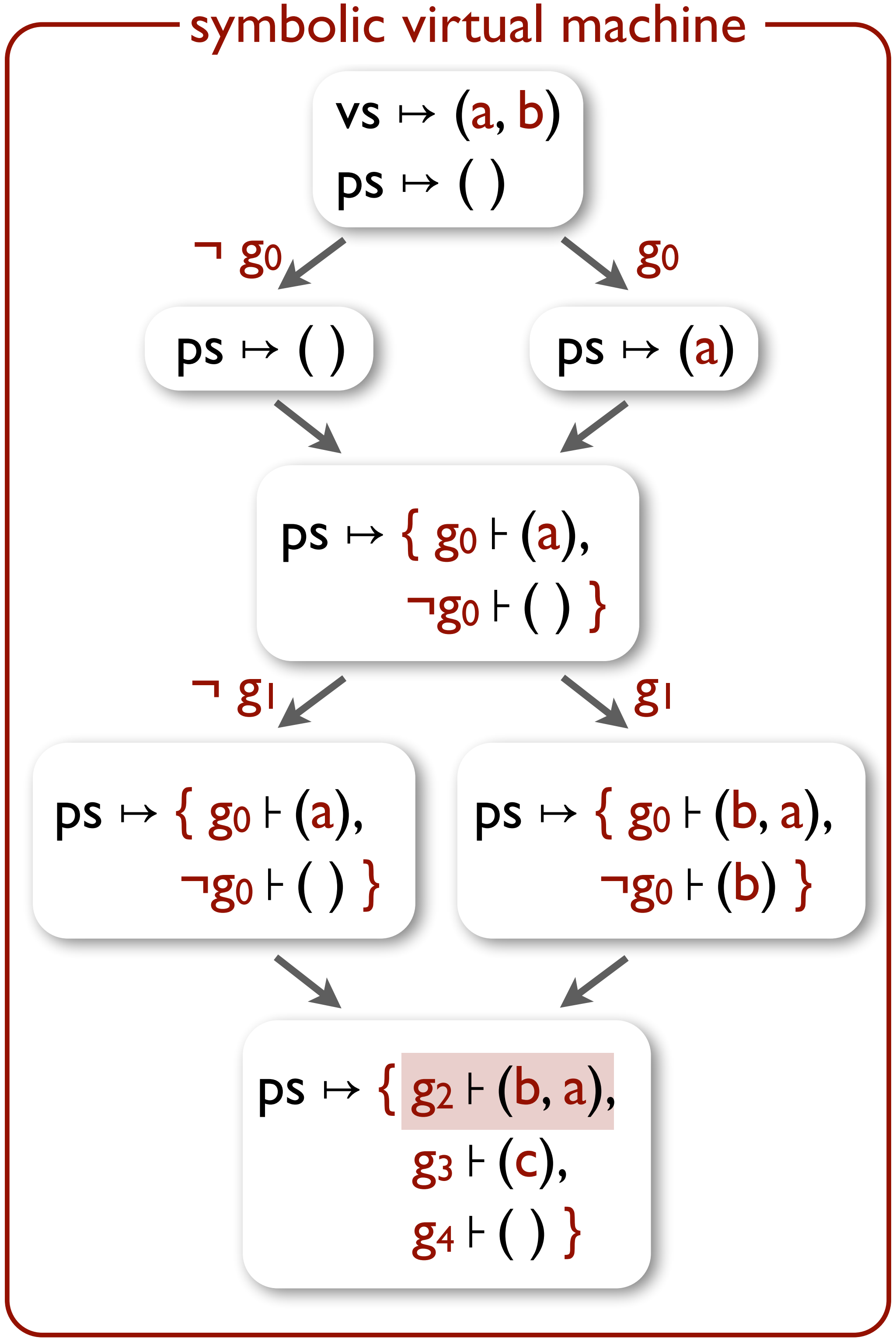
$g_2 = g_0 \wedge g_1$

$g_3 = \neg(g_0 \Leftrightarrow g_1)$

$g_4 = \neg g_0 \wedge \neg g_1$

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

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

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

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

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

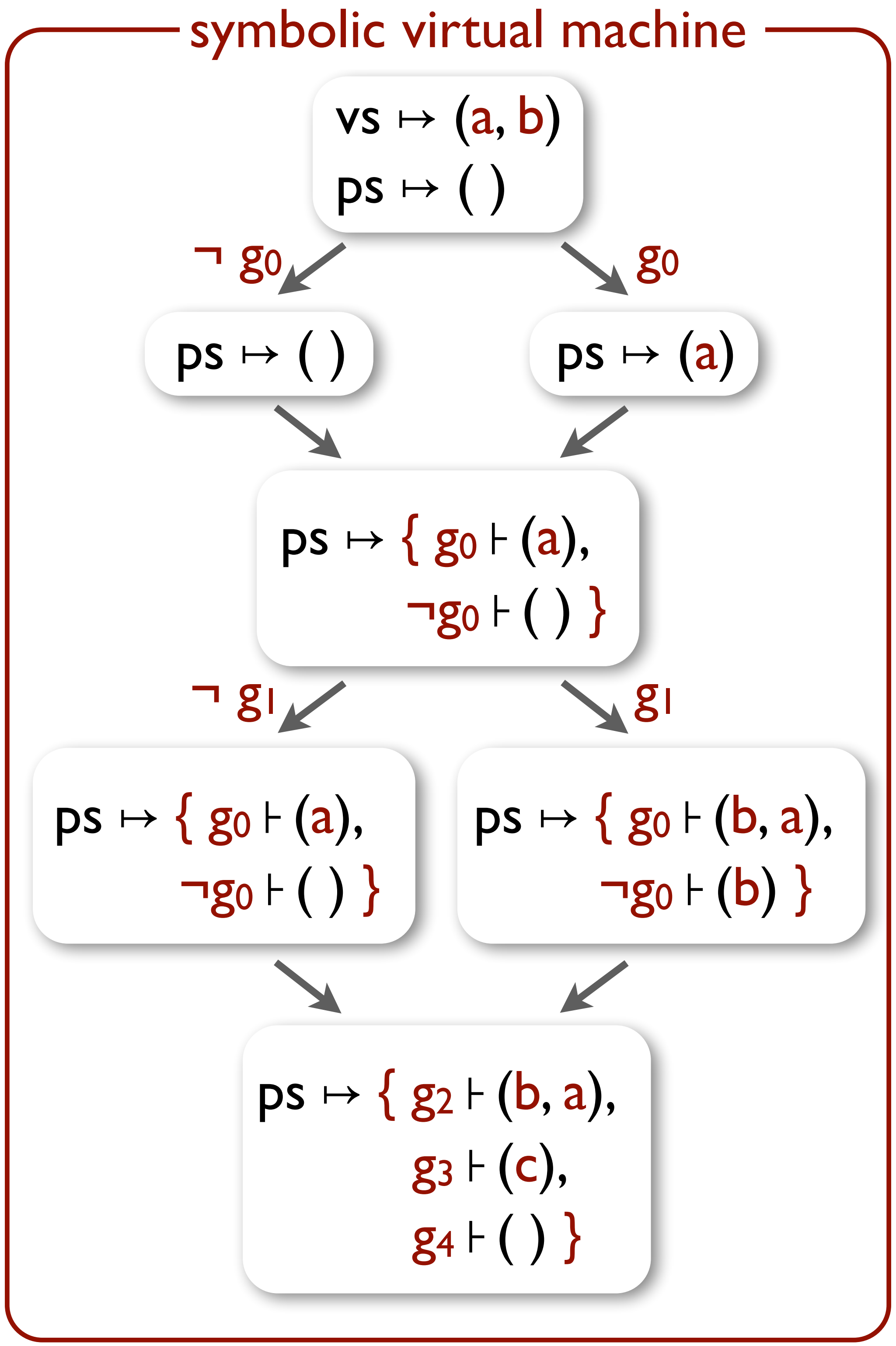
A new design: type-driven state merging

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

polynomial encoding

concrete evaluation

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



symbolic virtual machine

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

$\neg\ g_0$

g_0

$ps \mapsto ()$

$ps \mapsto (a)$

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

$\neg\ g_1$

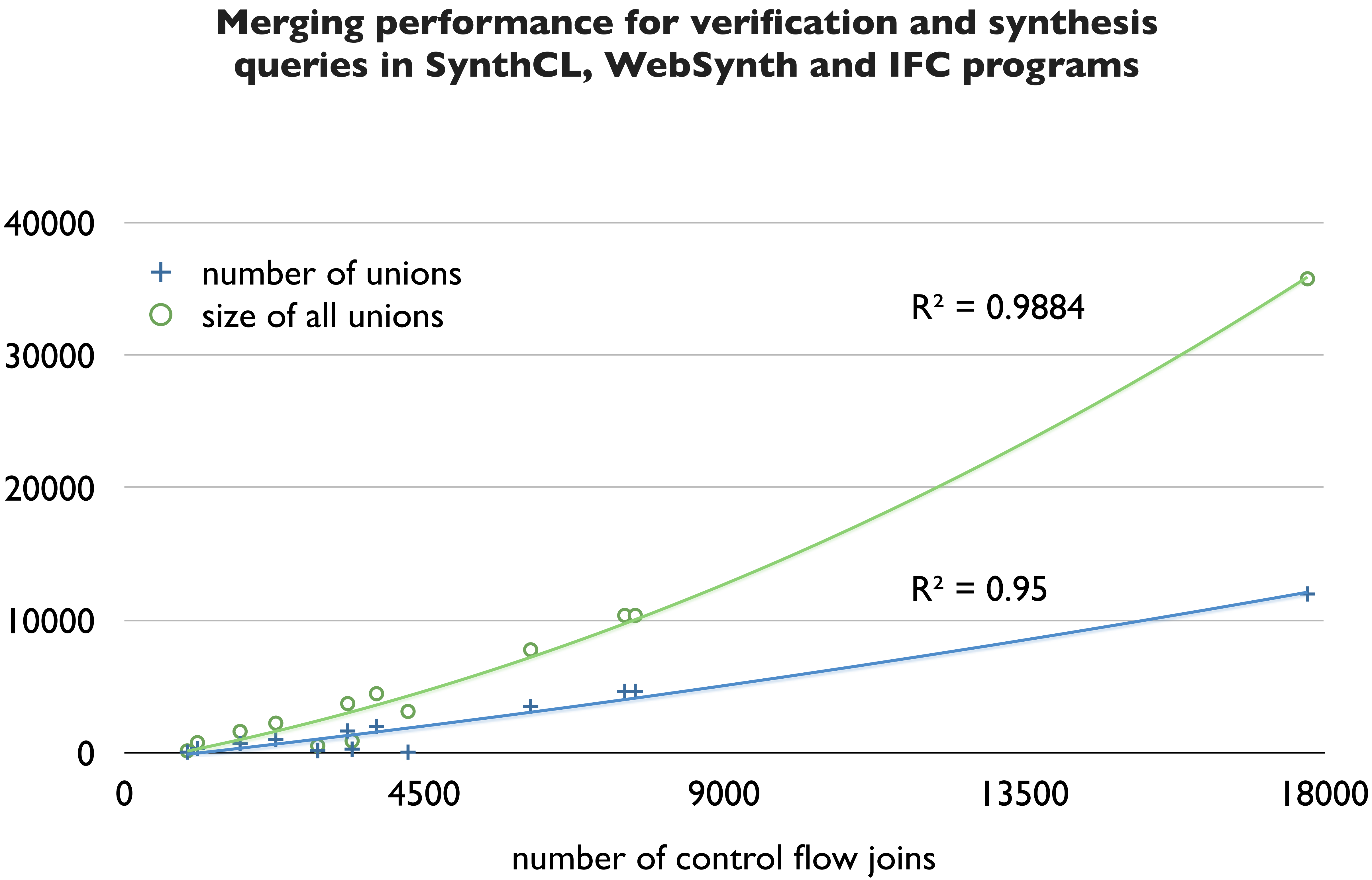
g_1

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

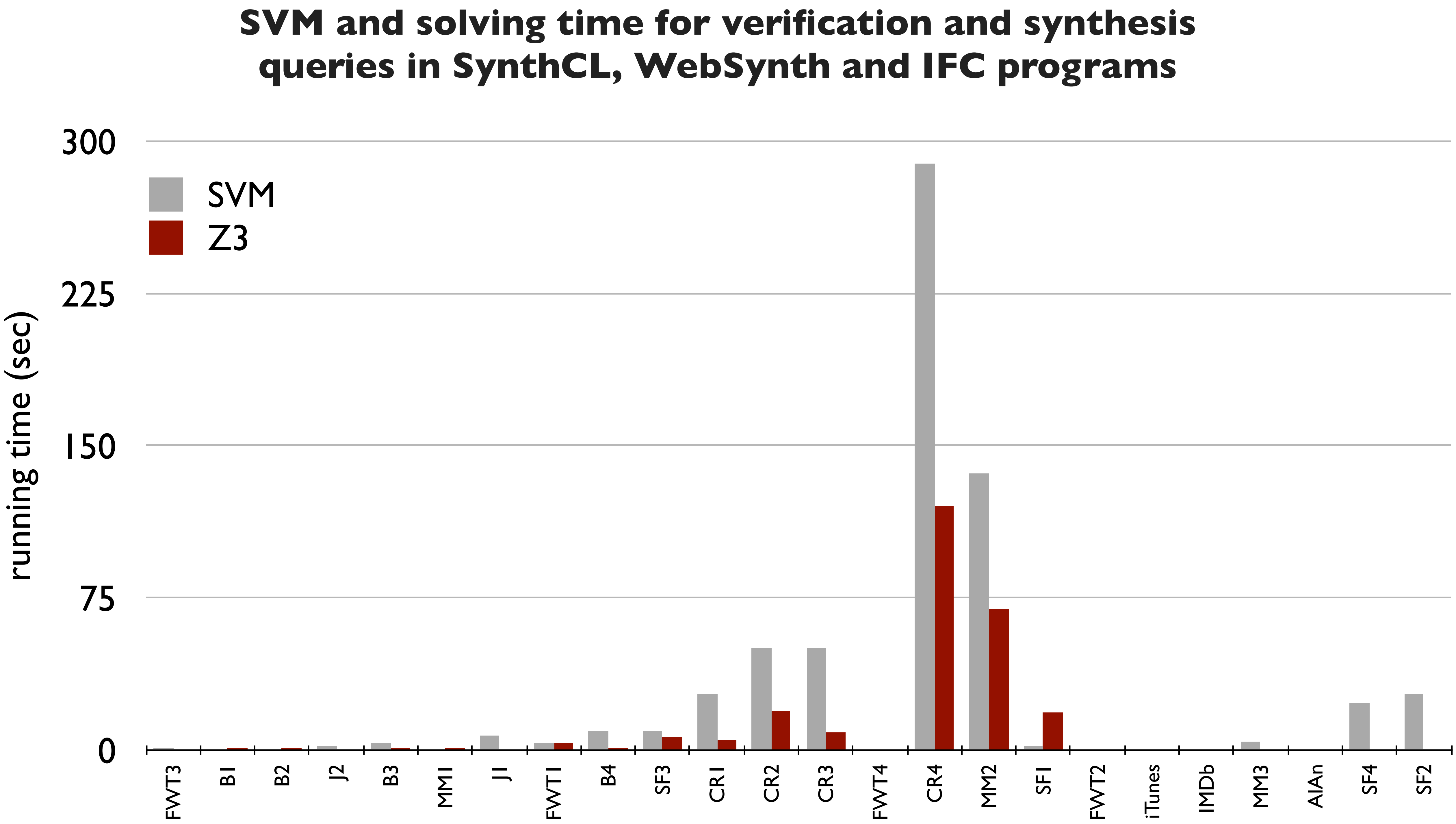
$ps \mapsto \{\ g_0 \vdash (b, a),$
$\neg g_0 \vdash (b)\ \}$

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

Effectiveness of type-driven state merging

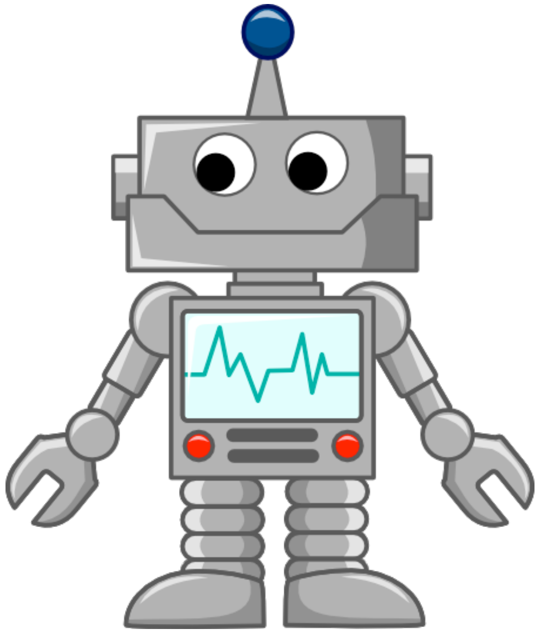


Effectiveness of type-driven state merging



future

advanced programming for everyone



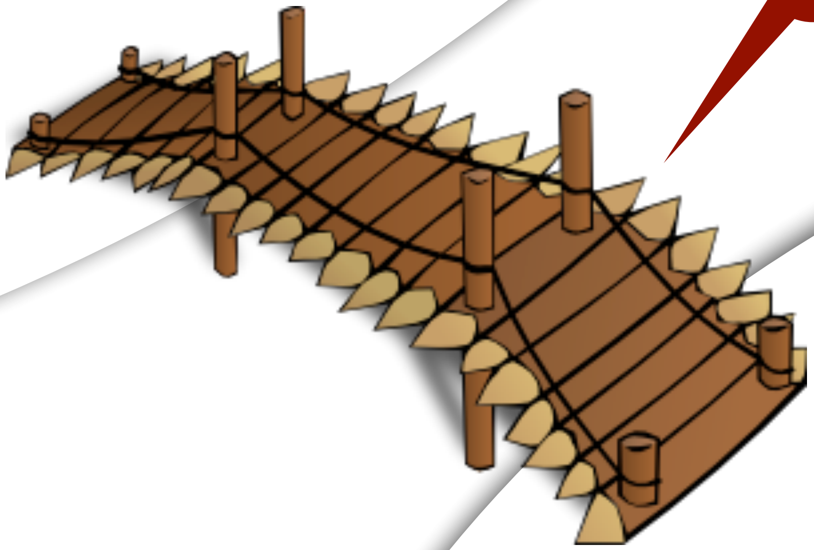
Where next?

web scraping scripts

secure stack machines

spatial programs

data-parallel programs

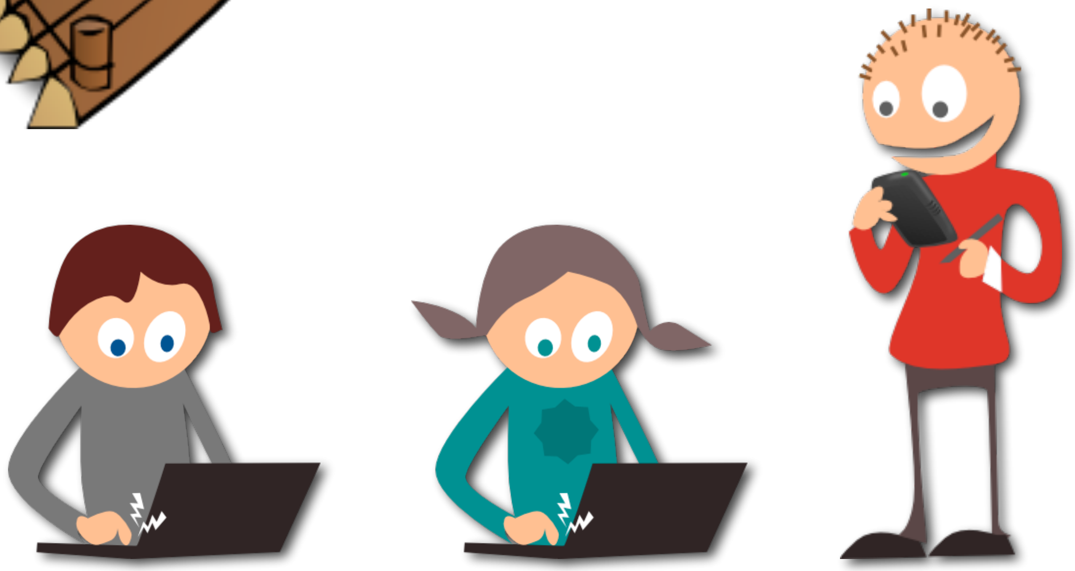


solver-aided languages

less time

less code

less effort

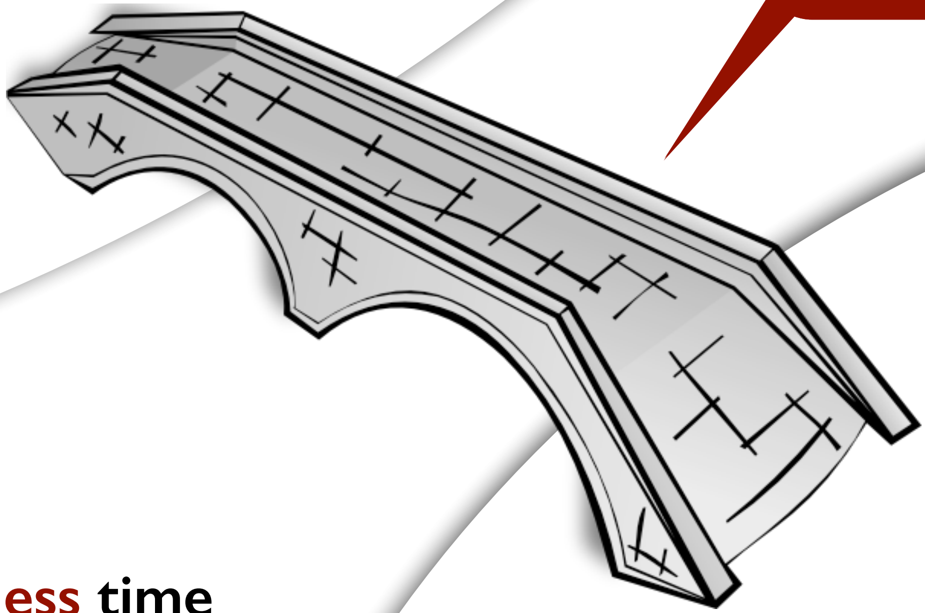


Where next?

harder programs

new kinds of programs

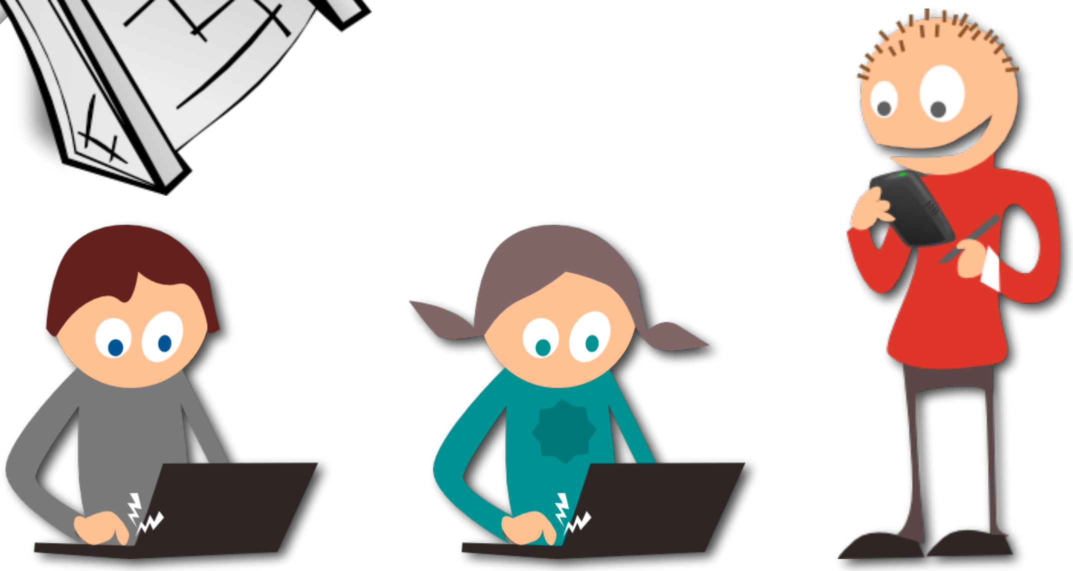
advanced solver-aided languages



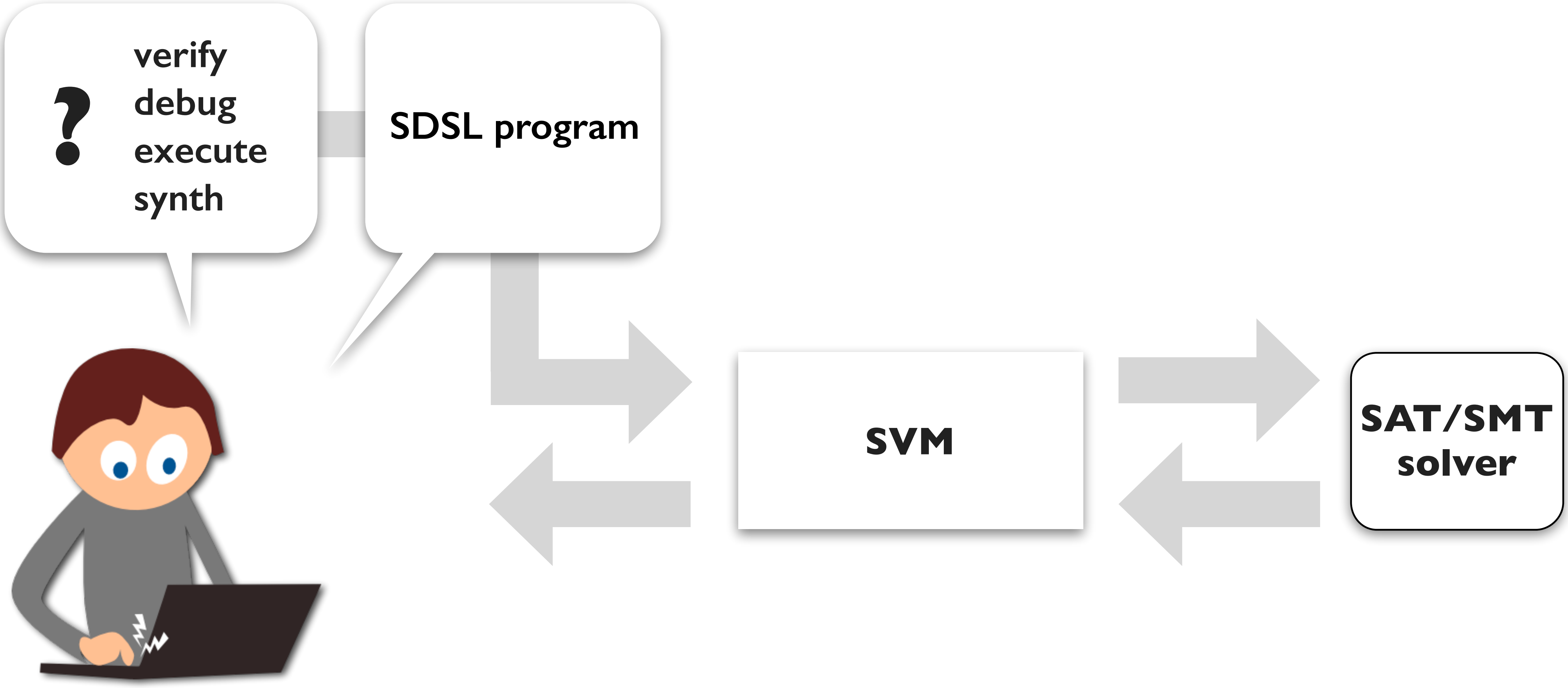
less time

less code

less effort



Keeping the programmer in the loop



?

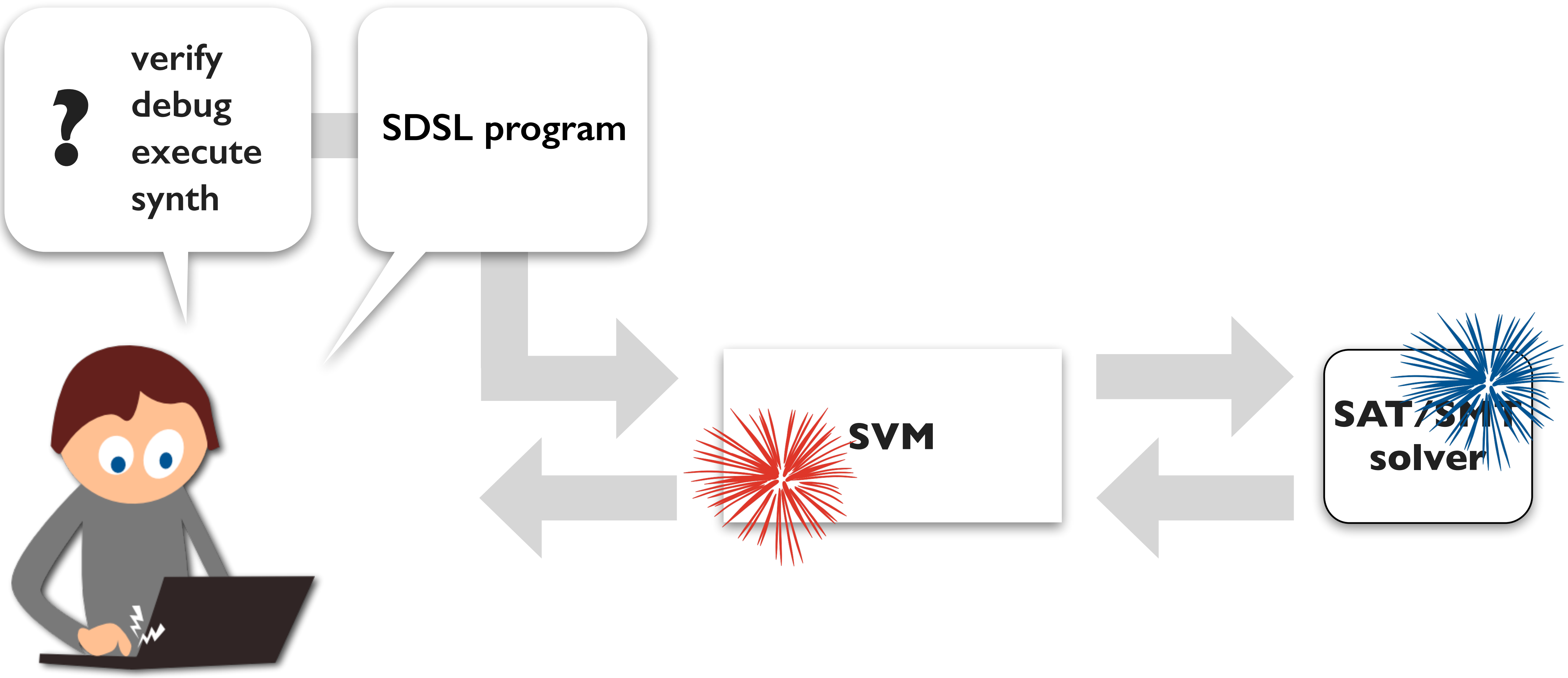
verify
debug
execute
synth

SDSL program

SVM

SAT/SMT solver

Keeping the programmer in the loop



?

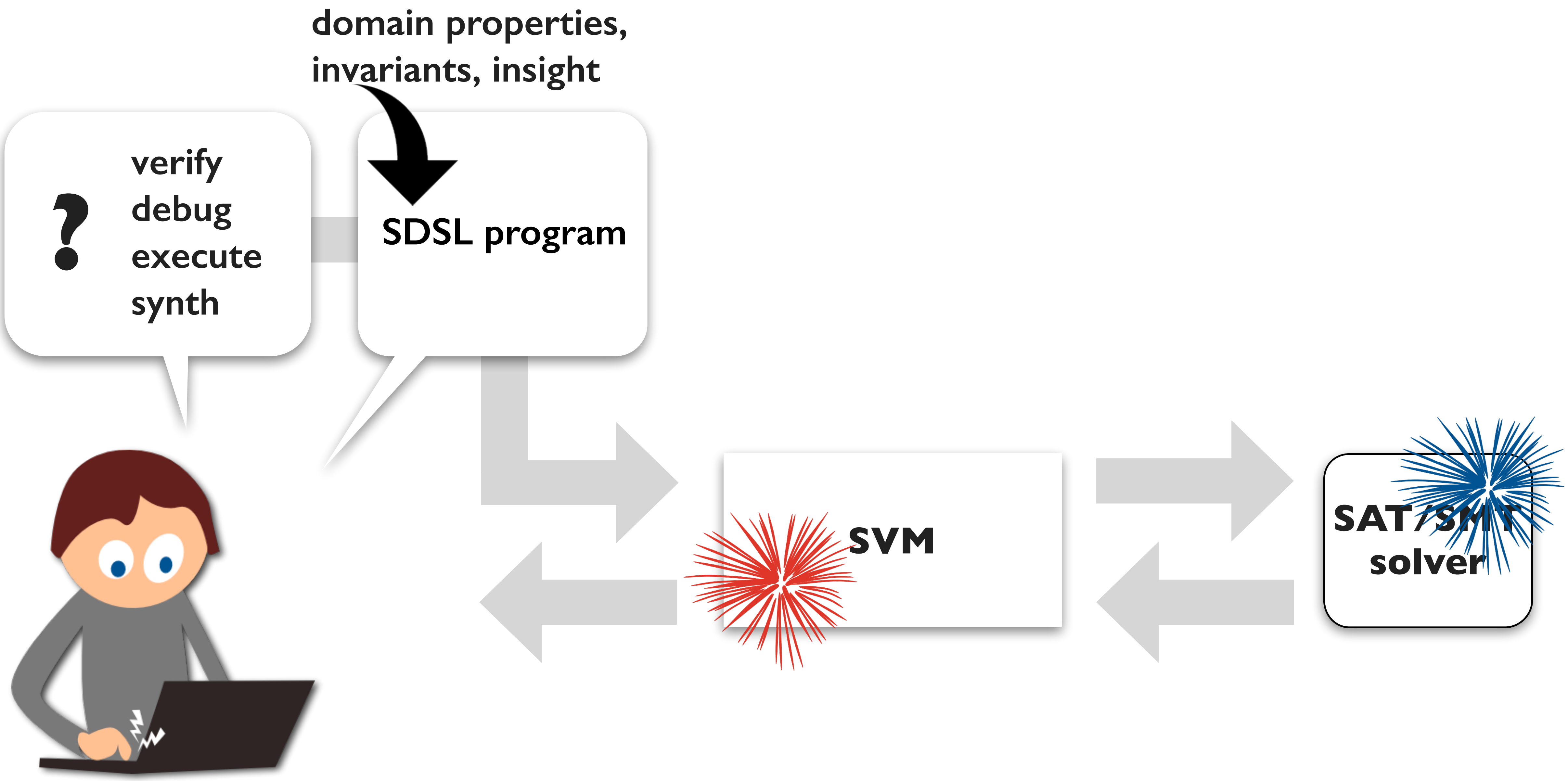
verify
debug
execute
synth

SDSL program

SVM

SAT/SMT solver

Keeping the programmer in the loop



domain properties, invariants, insight

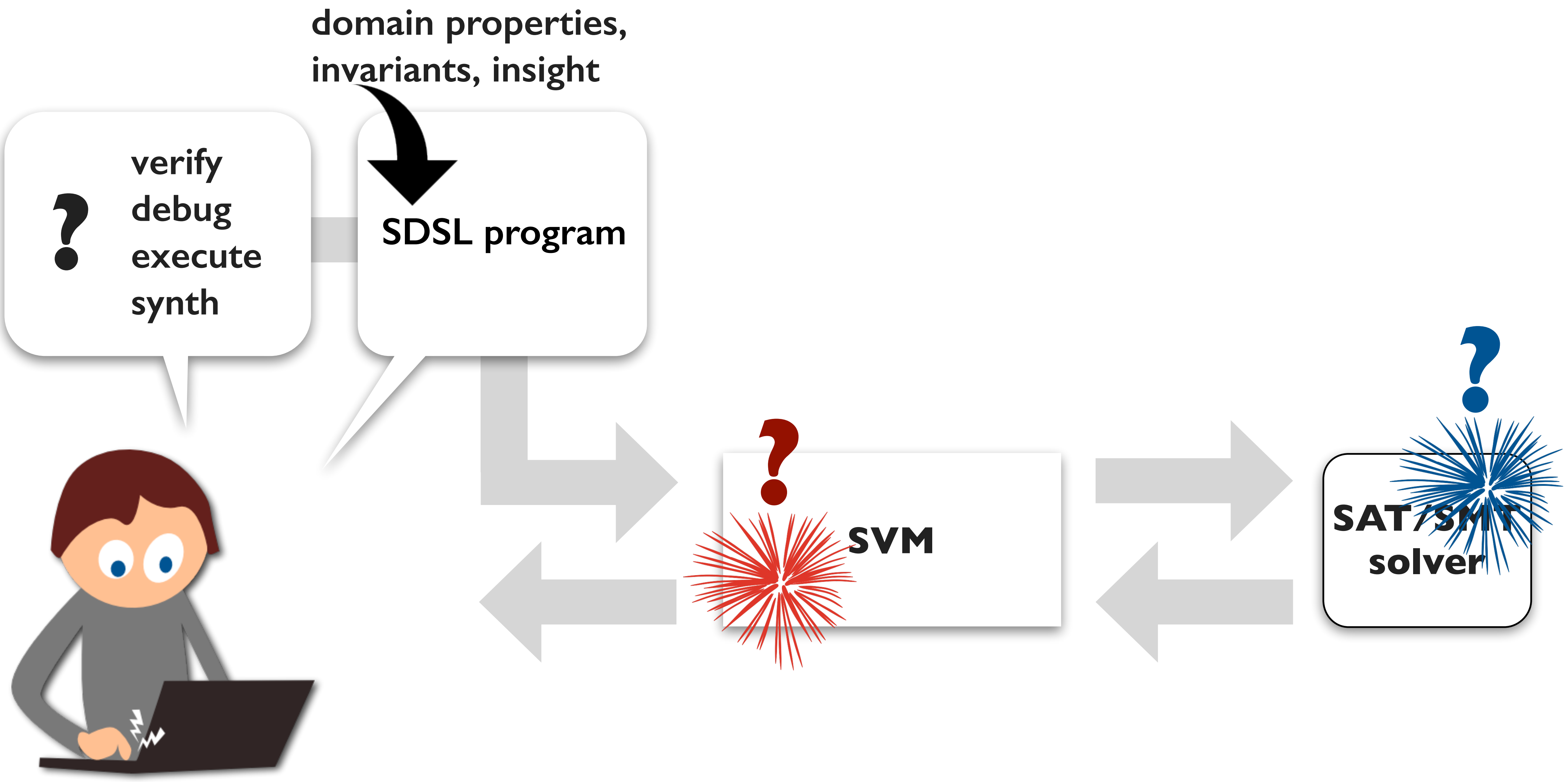
? verify debug execute synth

SDSL program

SVM

SAT/SMT solver

Keeping the programmer in the loop



domain properties, invariants, insight

?

verify
debug
execute
synth

SDSL program

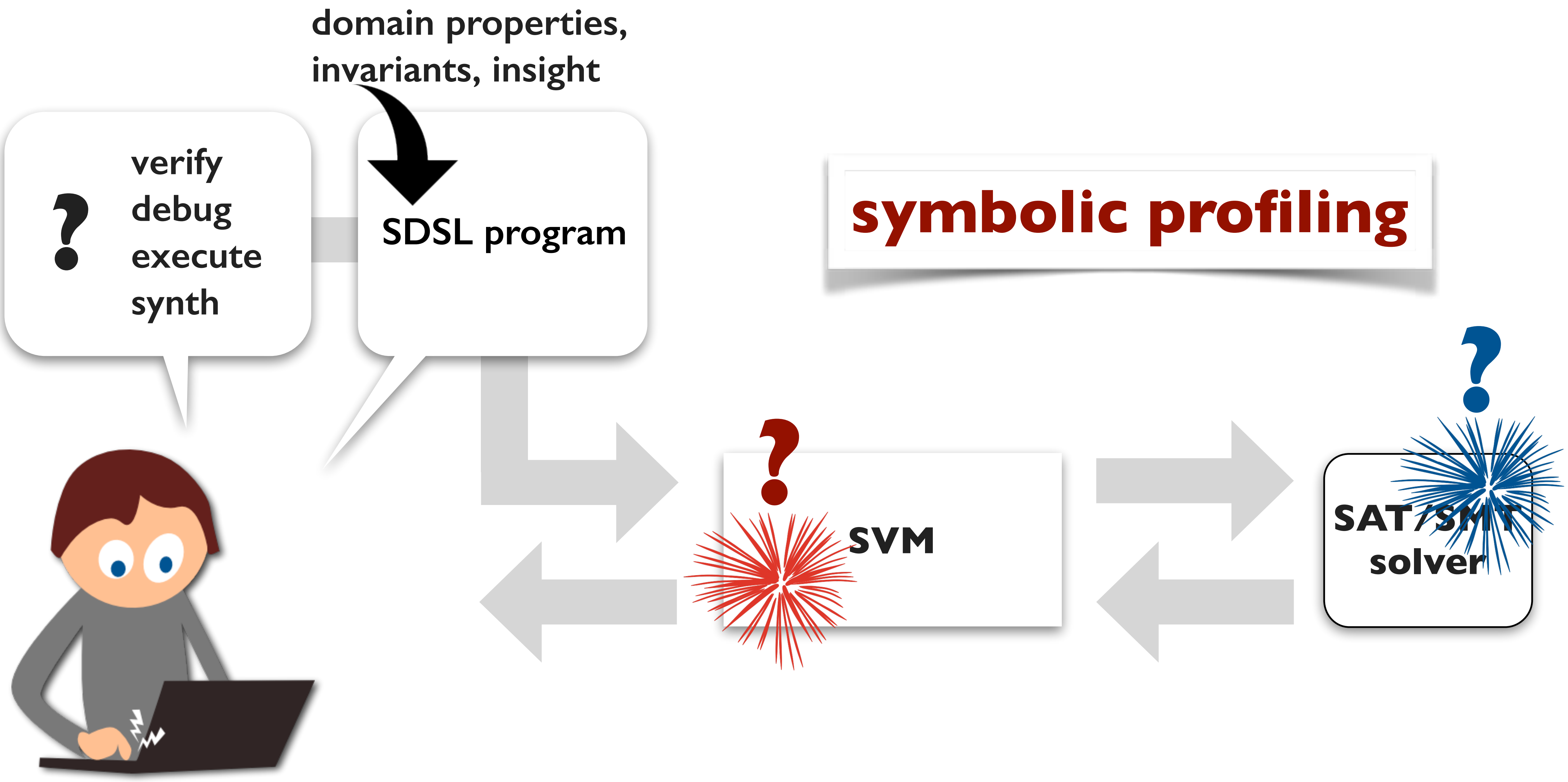
?

SVM

?

SAT/SMT solver

Keeping the programmer in the loop



domain properties,
invariants, insight

?
verify
debug
execute
synth

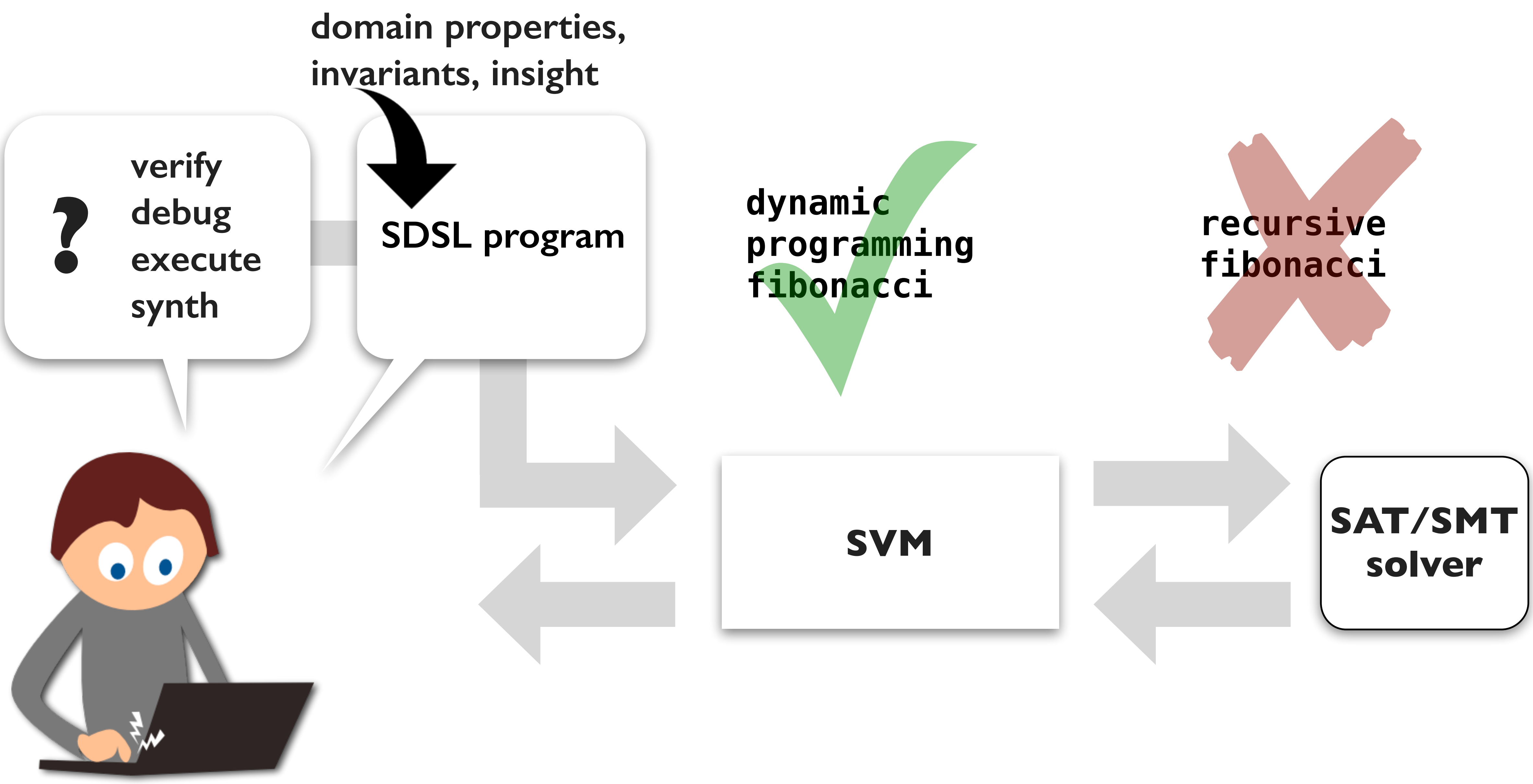
SDSL program

symbolic profiling

SVM

SAT/SMT solver

Keeping the system in the loop



domain properties, invariants, insight

?

verify

debug

execute

synth

SDSL program

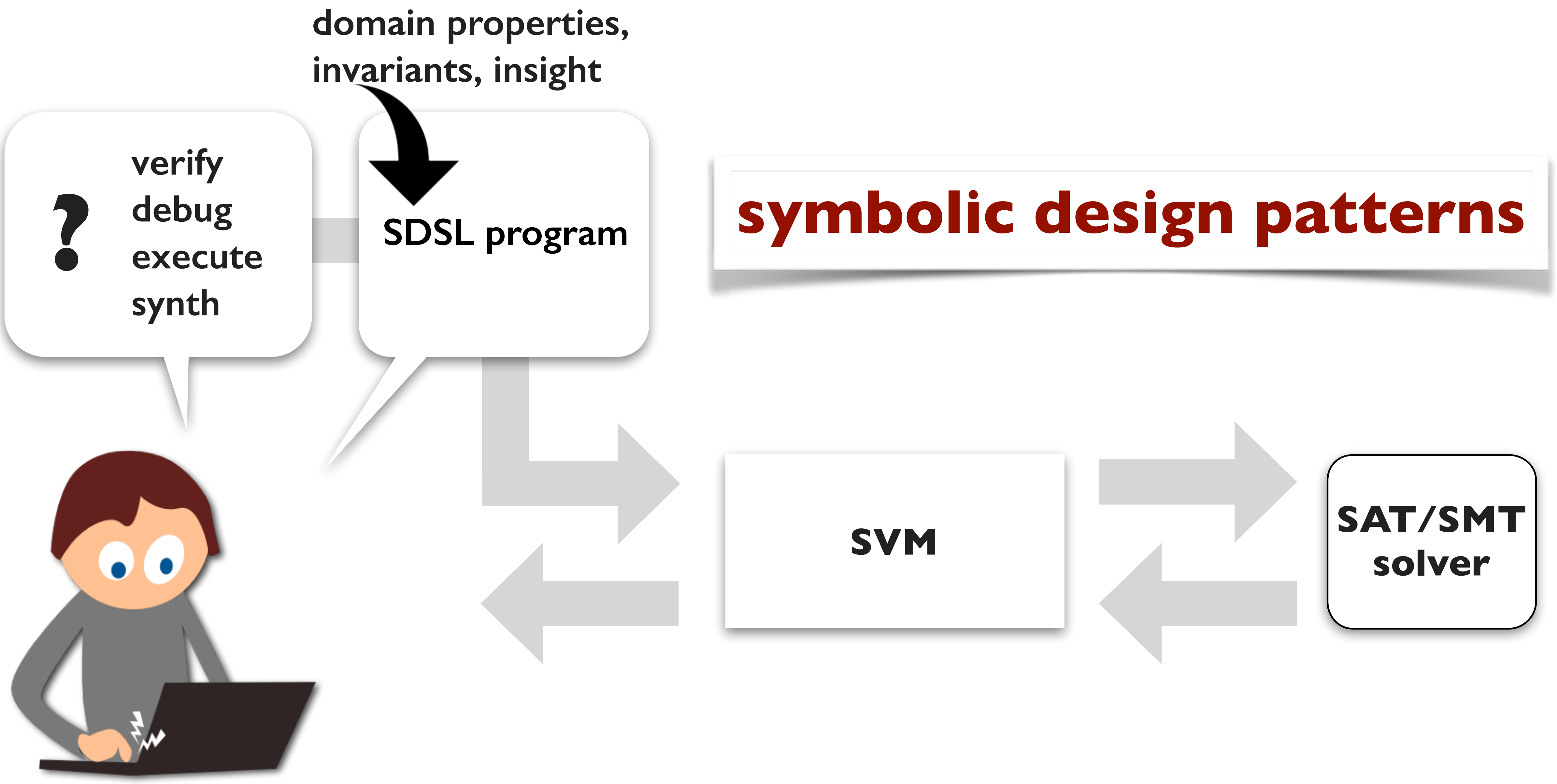
dynamic programming fibonacci

recursive fibonacci

SVM

SAT/SMT solver

Keeping the system in the loop



domain properties,
invariants, insight

?
verify
debug
execute
synth

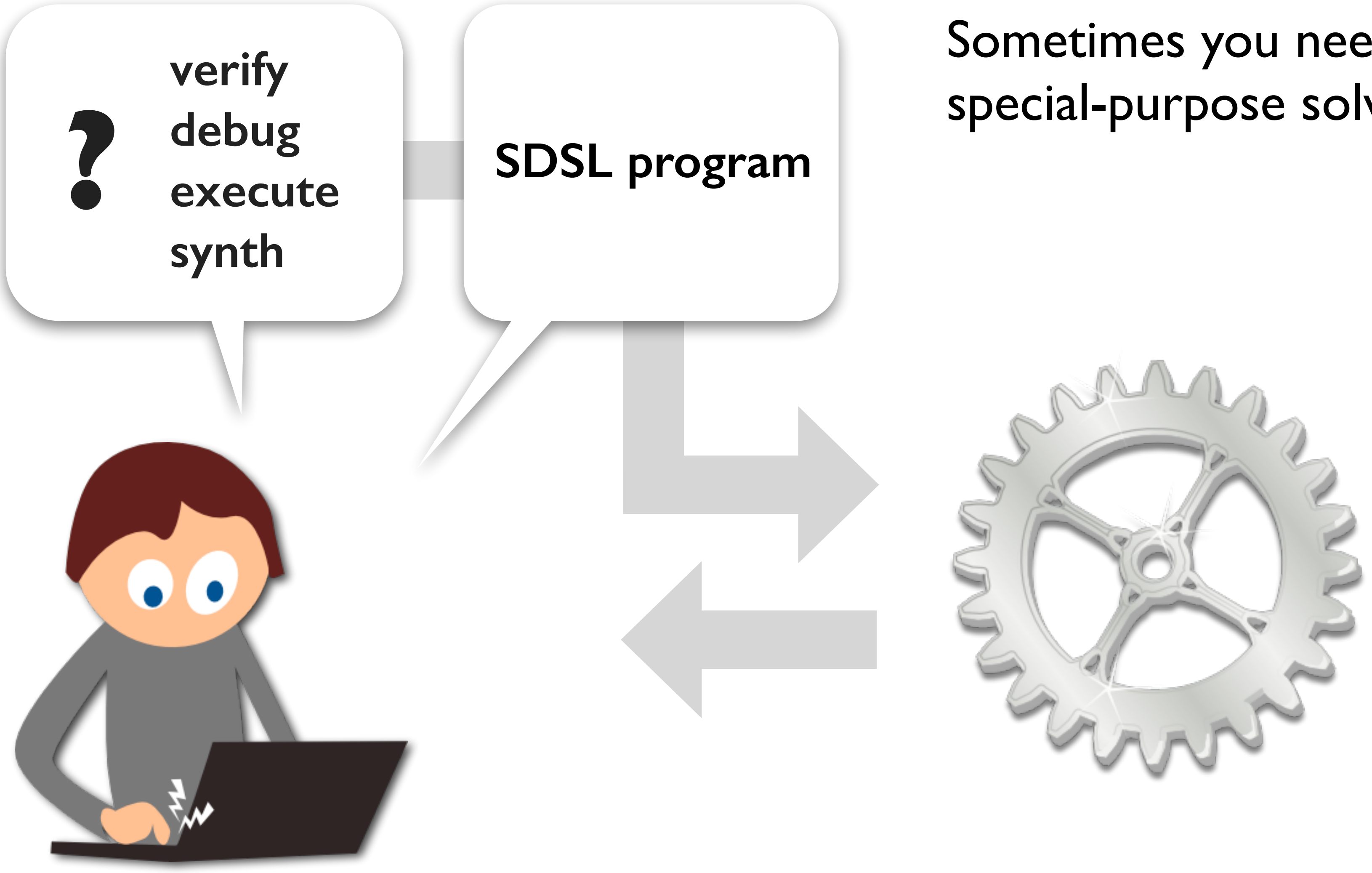
SDSL program

symbolic design patterns

SVM

SAT/SMT
solver

Domain-specific solvers



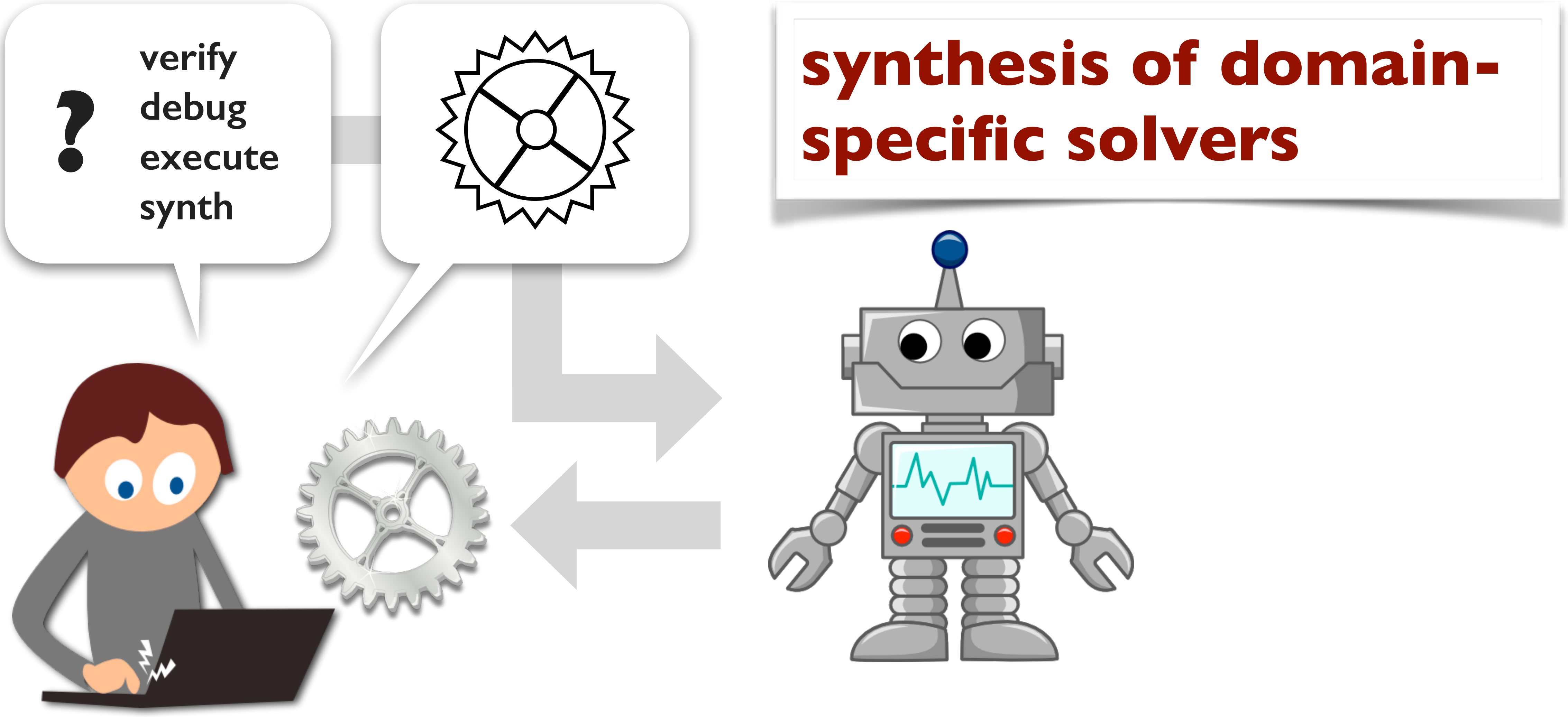
?

verify
debug
execute
synth

SDSL program

Sometimes you need a special-purpose solver …

Domain-specific solvers for everyone



synthesis of domain-specific solvers

So long, and thanks for all the fish!