Part 3:

Solver-Aided Programming for All

Luke Nelson, Emina Torlak, Xi Wang

Paul G. Allen School of Computer Science & Engineering

University of Washington

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

Efficient verification and synthesis tools

Solver-Aided Programming for All

domains and programmers!

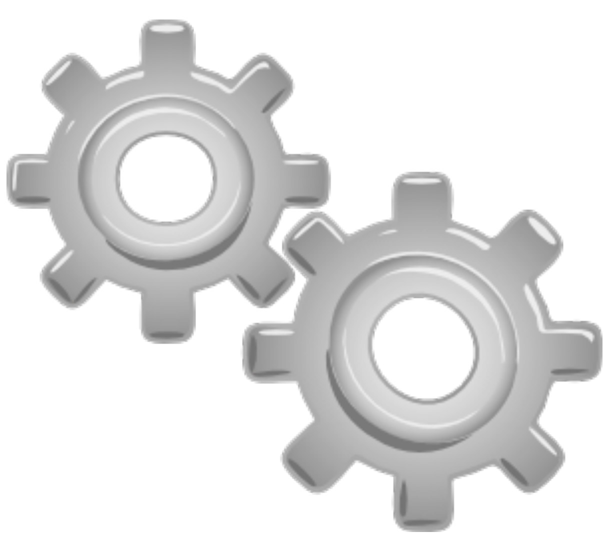
ROSETTE

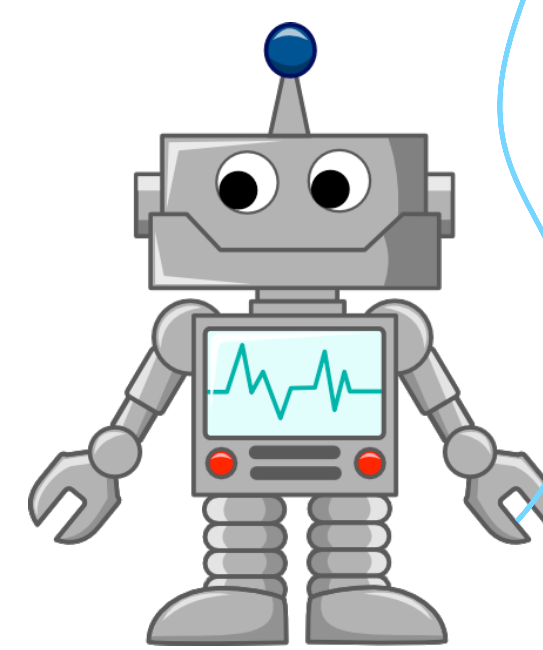
outline

solver-aided tools, languages, and applications

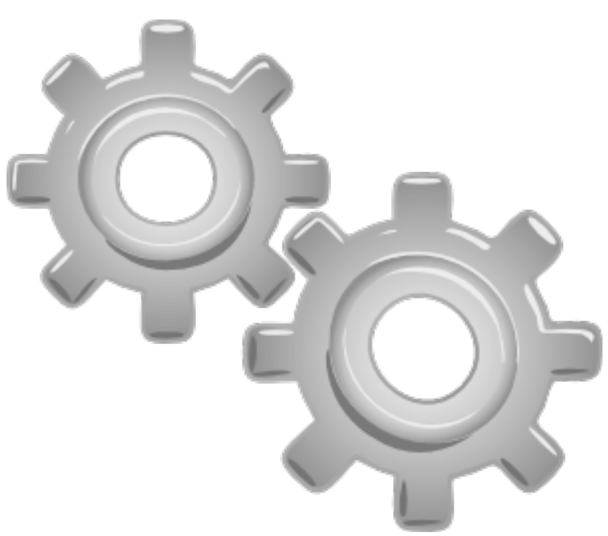
30+ tools

programming languages,
software engineering,
systems, architecture,
networks, security,
formal methods,
databases,
education,
games,
…

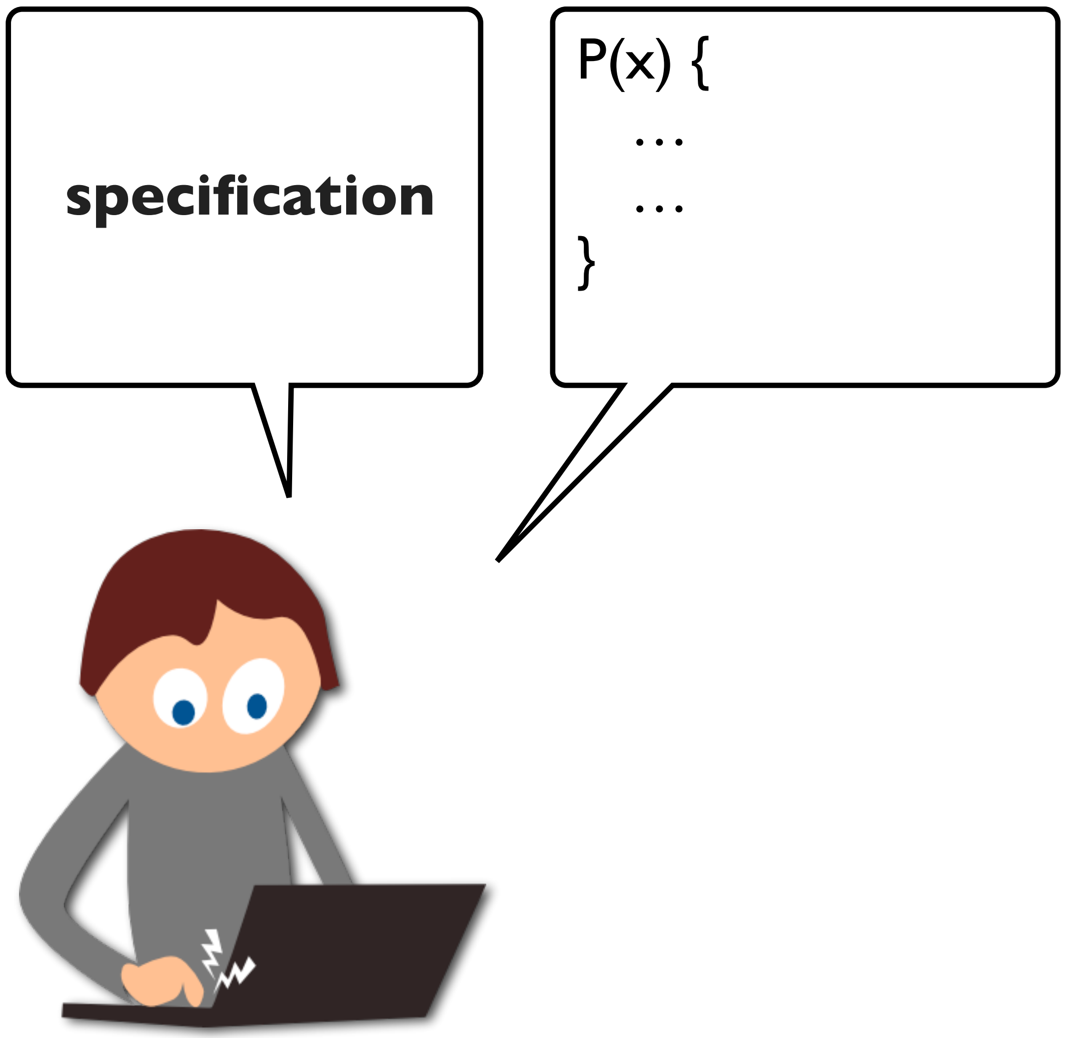




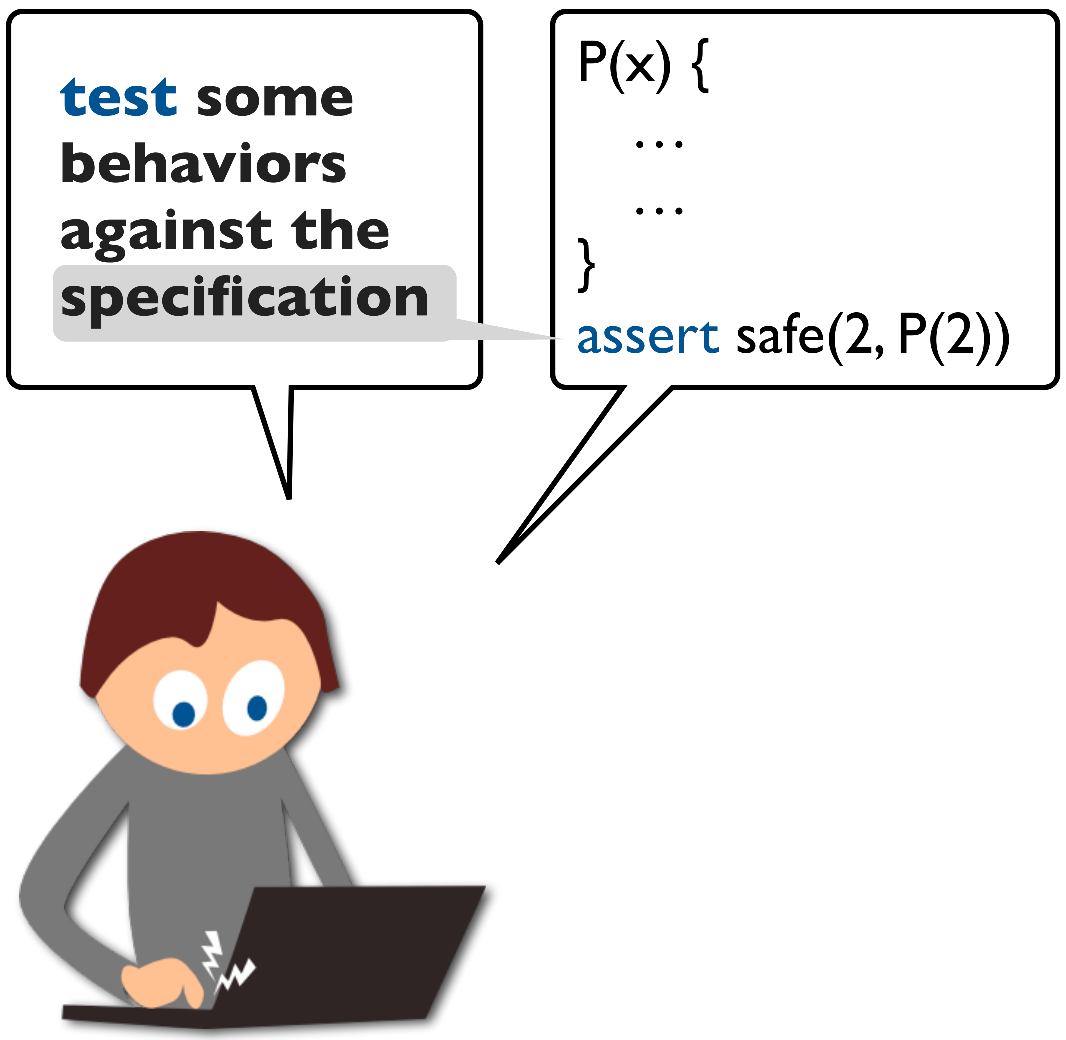
solver-aided tools



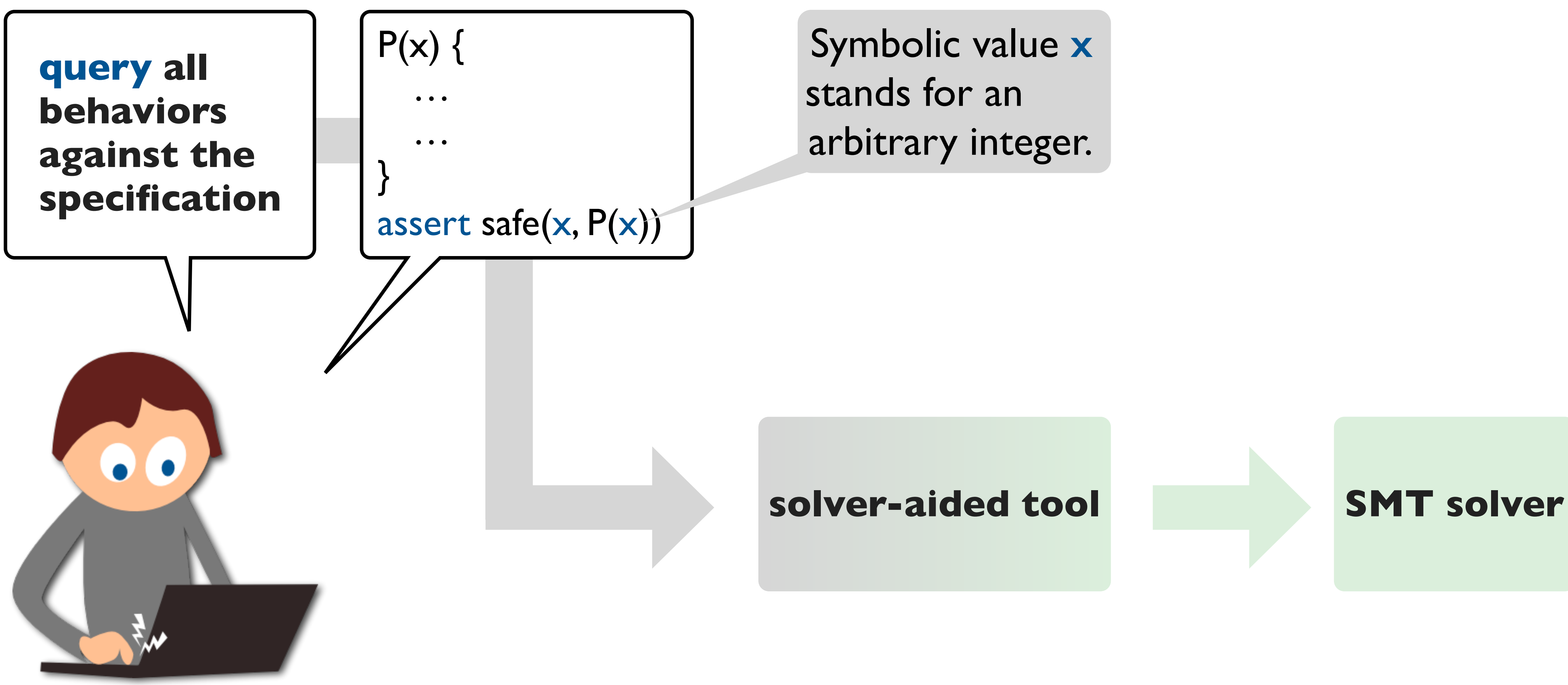
Title Text



Title Text

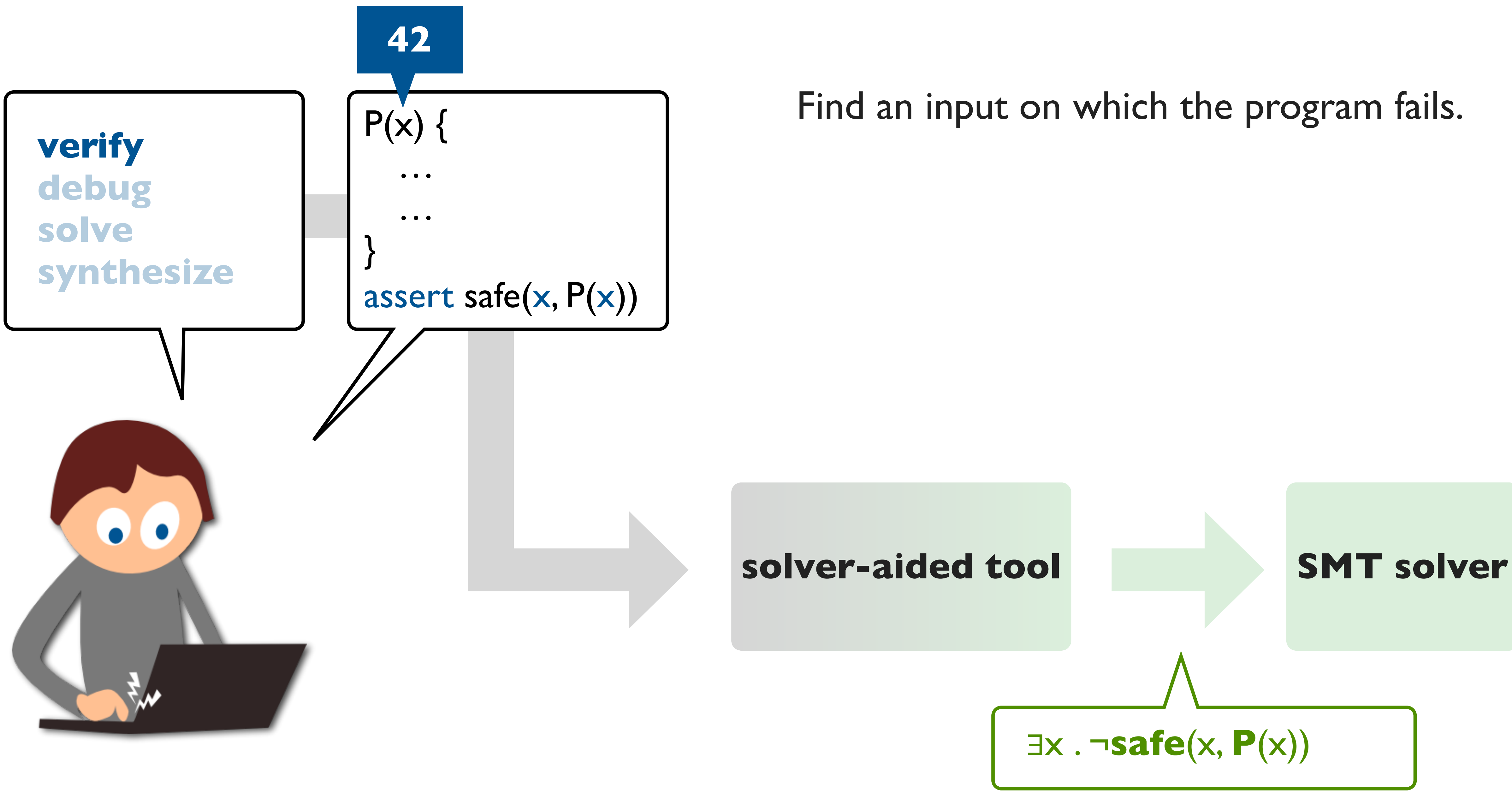

test some behaviors against the specification
P(x) {
…
…
}
assert safe(2, P(2))

Title Text



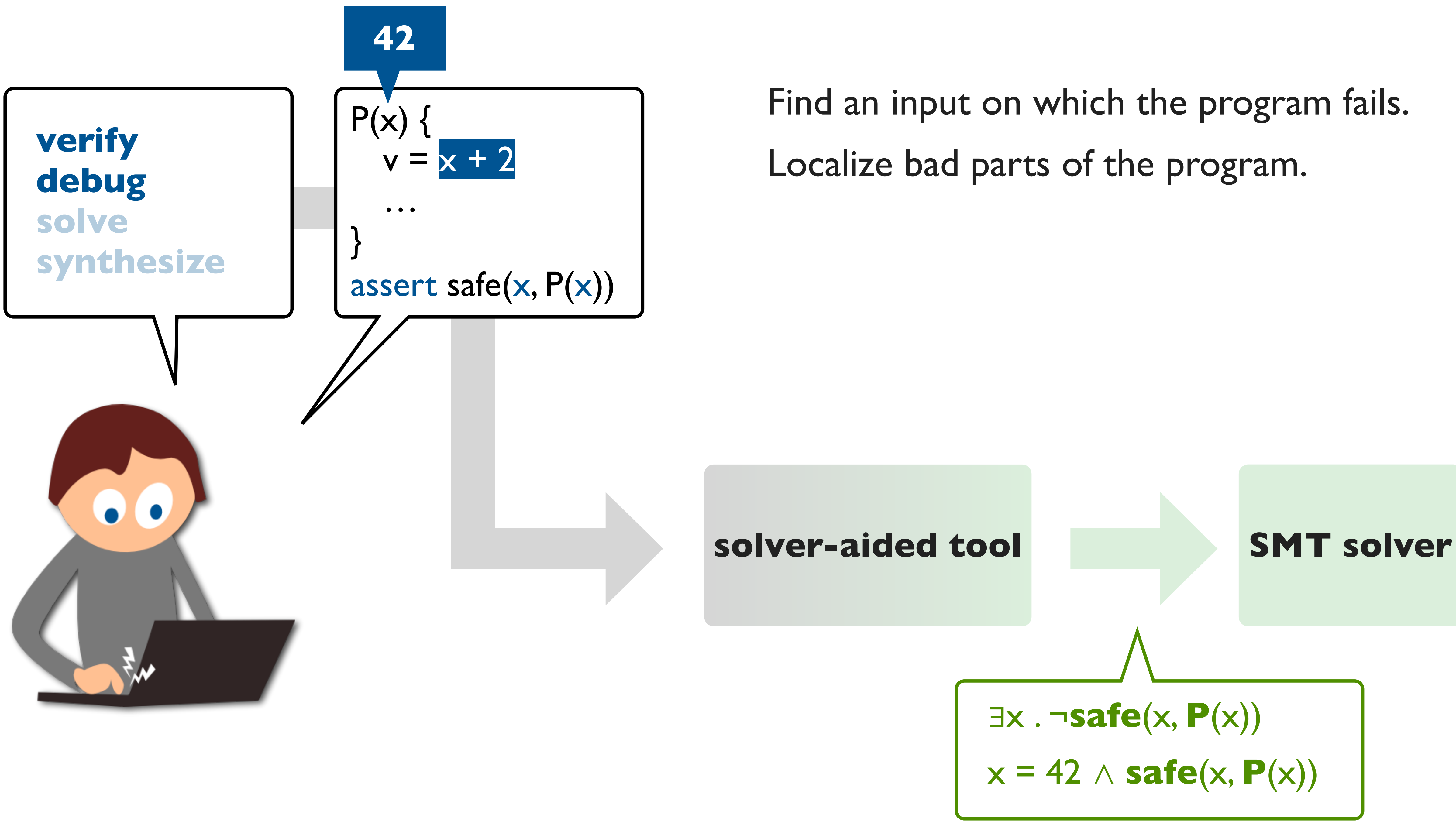
query all behaviors against the specification
P(x) {
…
…
}
assert safe(x, P(x))
Symbolic value x stands for an arbitrary integer.
solver-aided tool
SMT solver

Title Text



42
verify
debug
solve
synthesize
P(x) {
...
...
}
assert safe(x, P(x))
Find an input on which the program fails.
solver-aided tool
SMT solver
∃x . ¬safe(x, P(x))

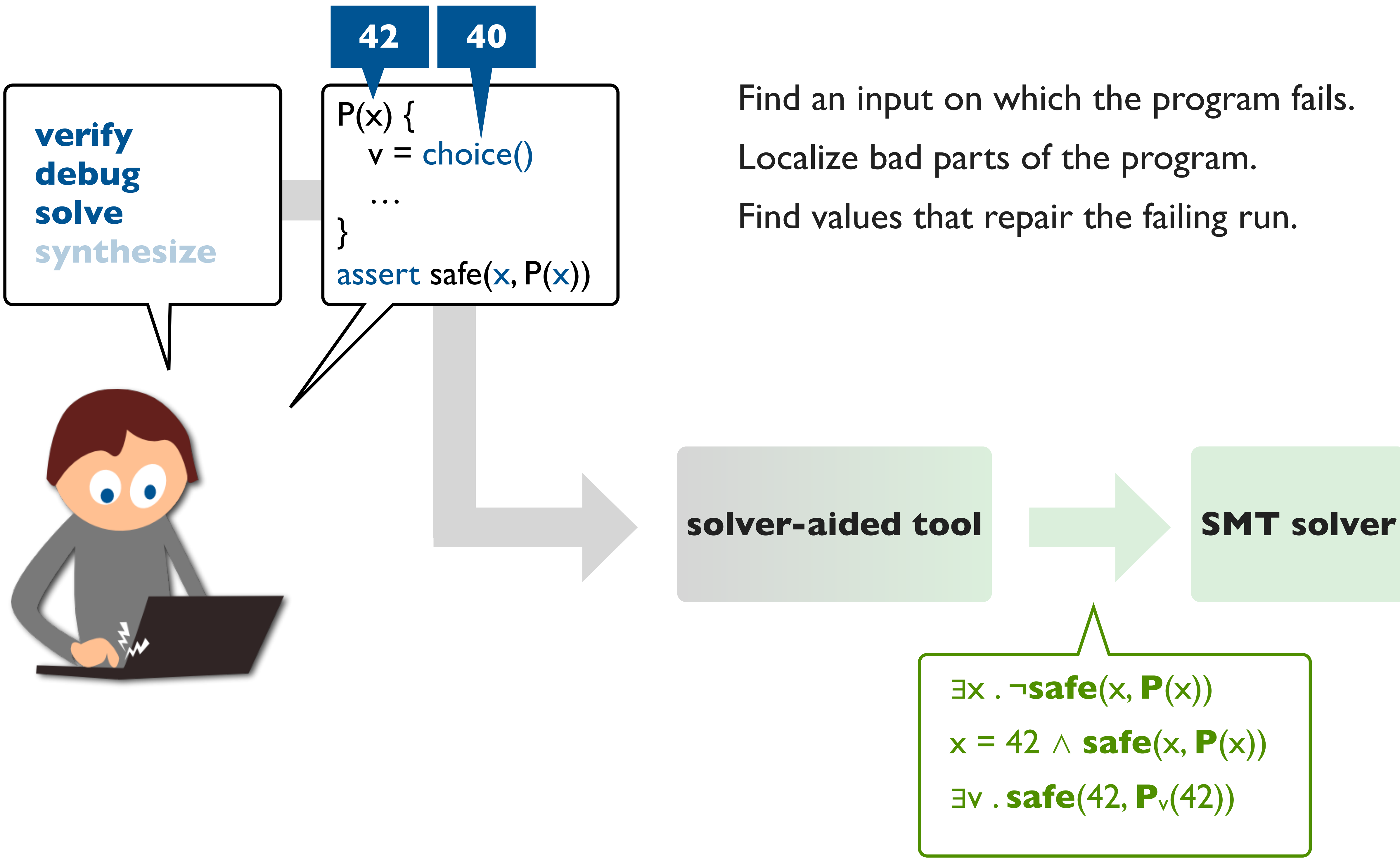
Title Text



Find an input on which the program fails.

Localize bad parts of the program.

Title Text



Find an input on which the program fails.

Localize bad parts of the program.

Find values that repair the failing run.

Title Text

verify
debug
solve
synthesize

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

x-2

Find an input on which the program fails.

Localize bad parts of the program.

Find values that repair the failing run.

Find code that repairs the program.

solver-aided tool → **SMT solver**

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

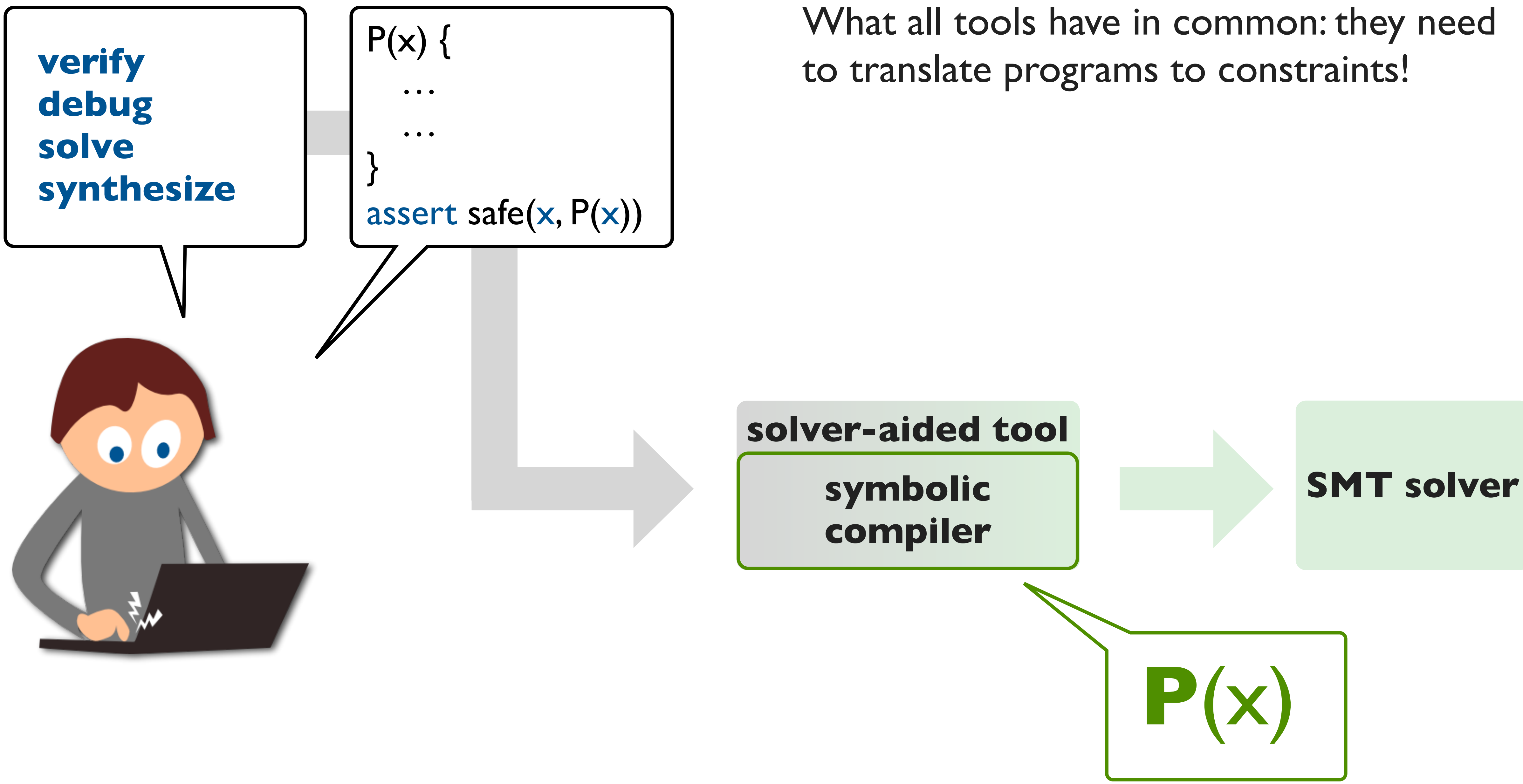
$x = 42 \land \mathbf{safe}(x, \mathbf{P}(x))$

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

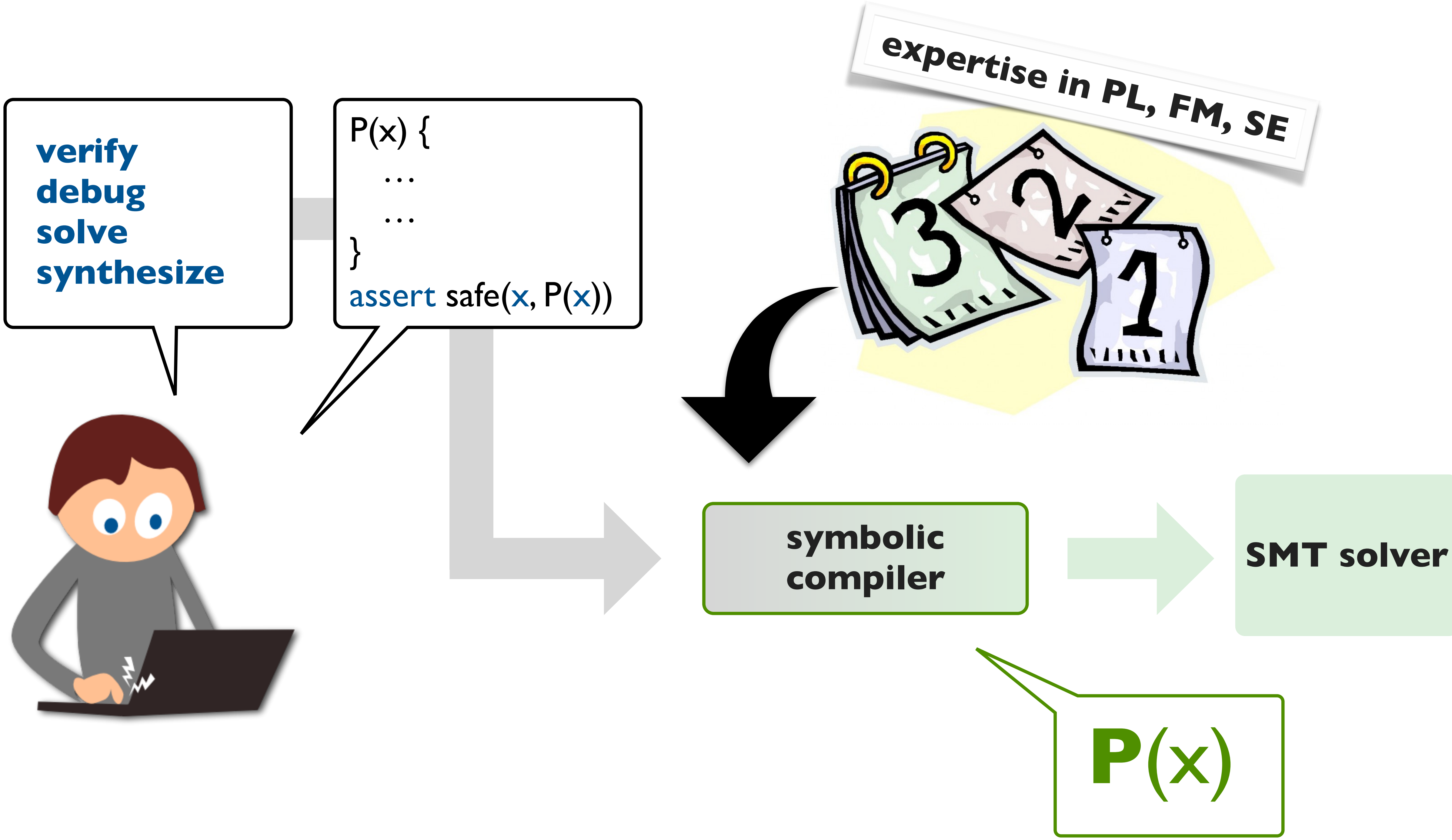
$\exists e . \forall x . \, \mathbf{safe}(x, \mathbf{P}_e(x))$

The classic (hard) way to build solver-aided tools

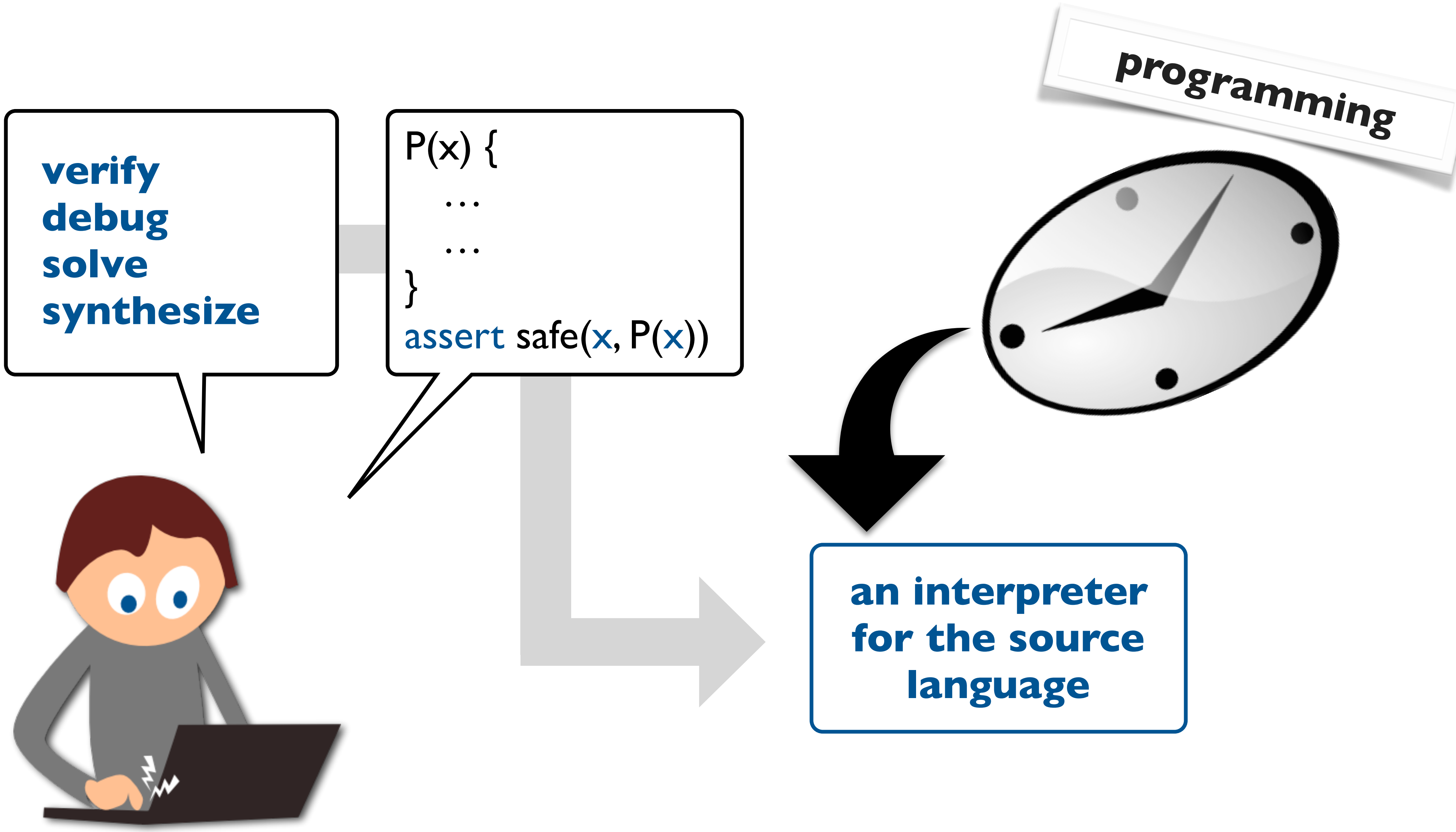
What all tools have in common: they need to translate programs to constraints!



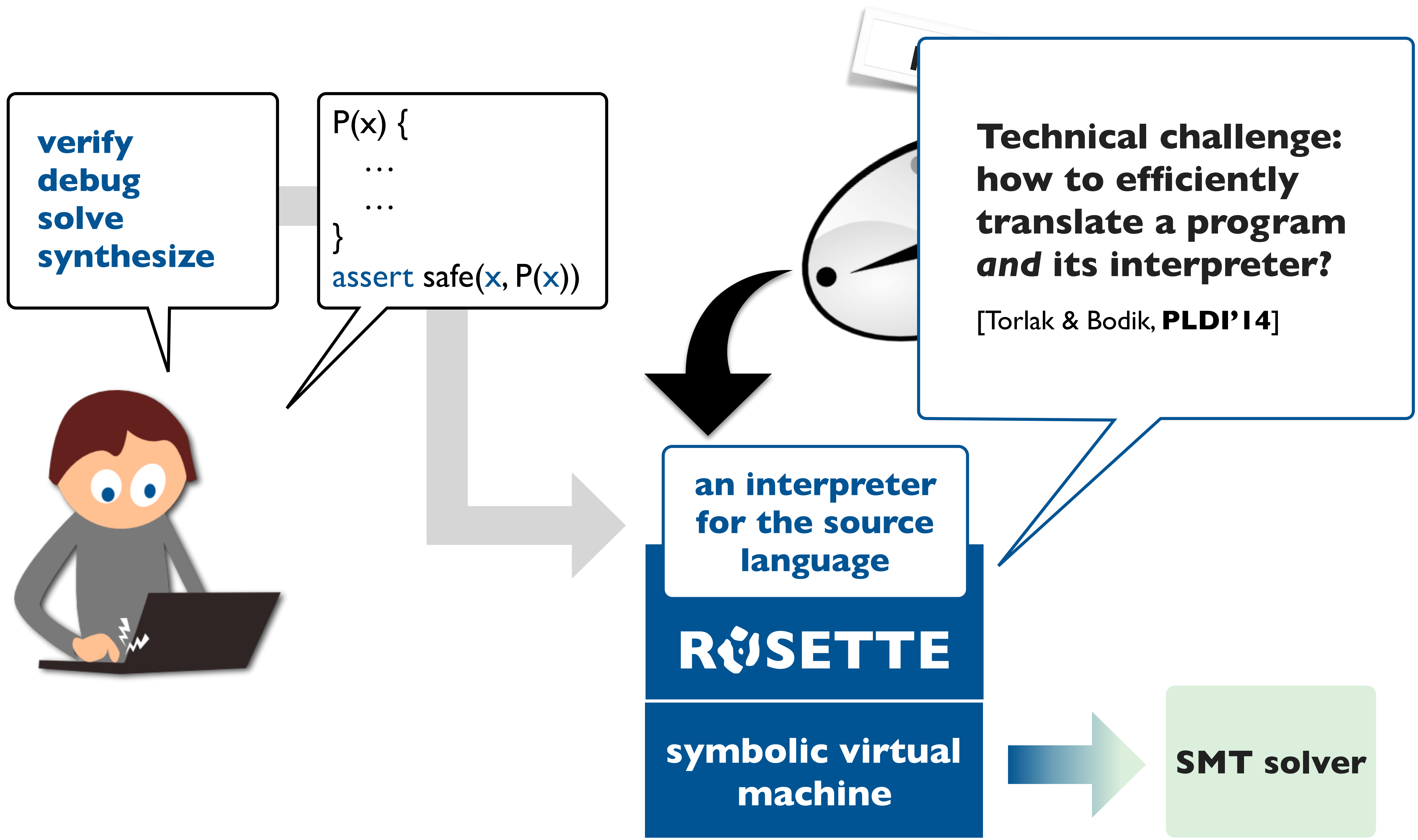
The classic (hard) way to build solver-aided tools



Is there an easier way to build tools?



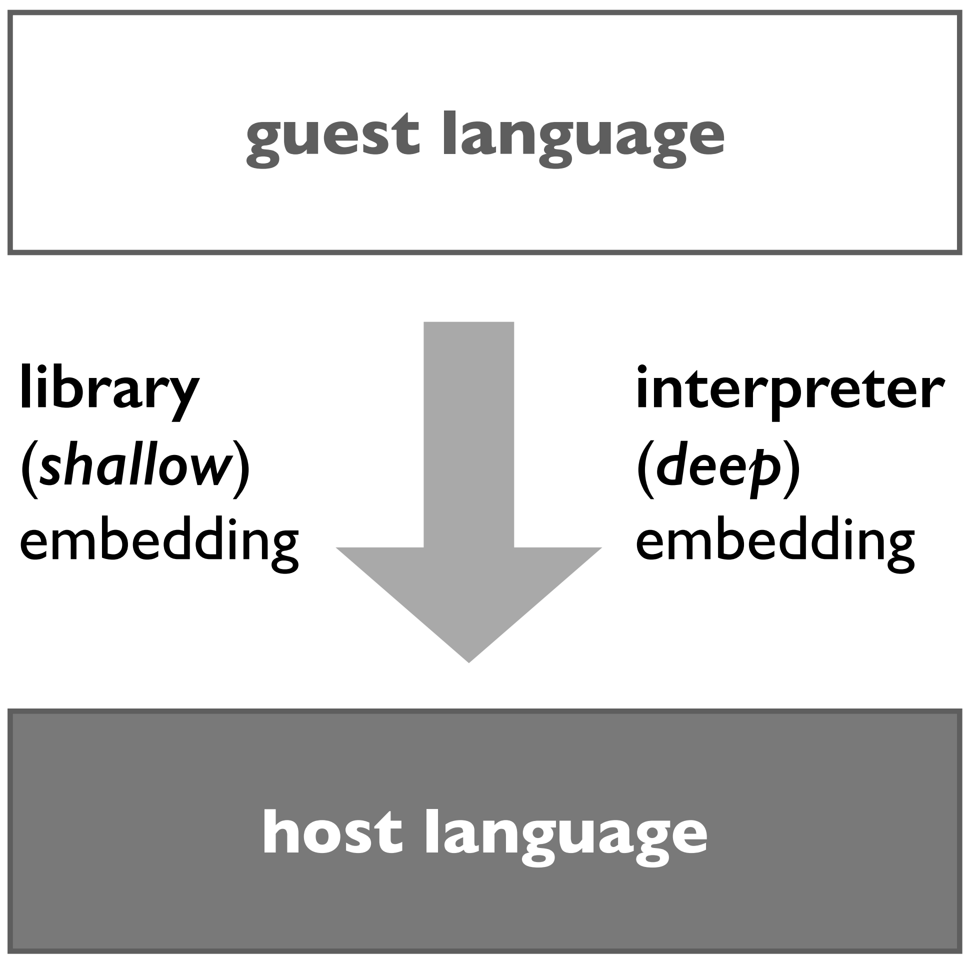
Yes, with solver-aided languages!



design

solver-aided languages

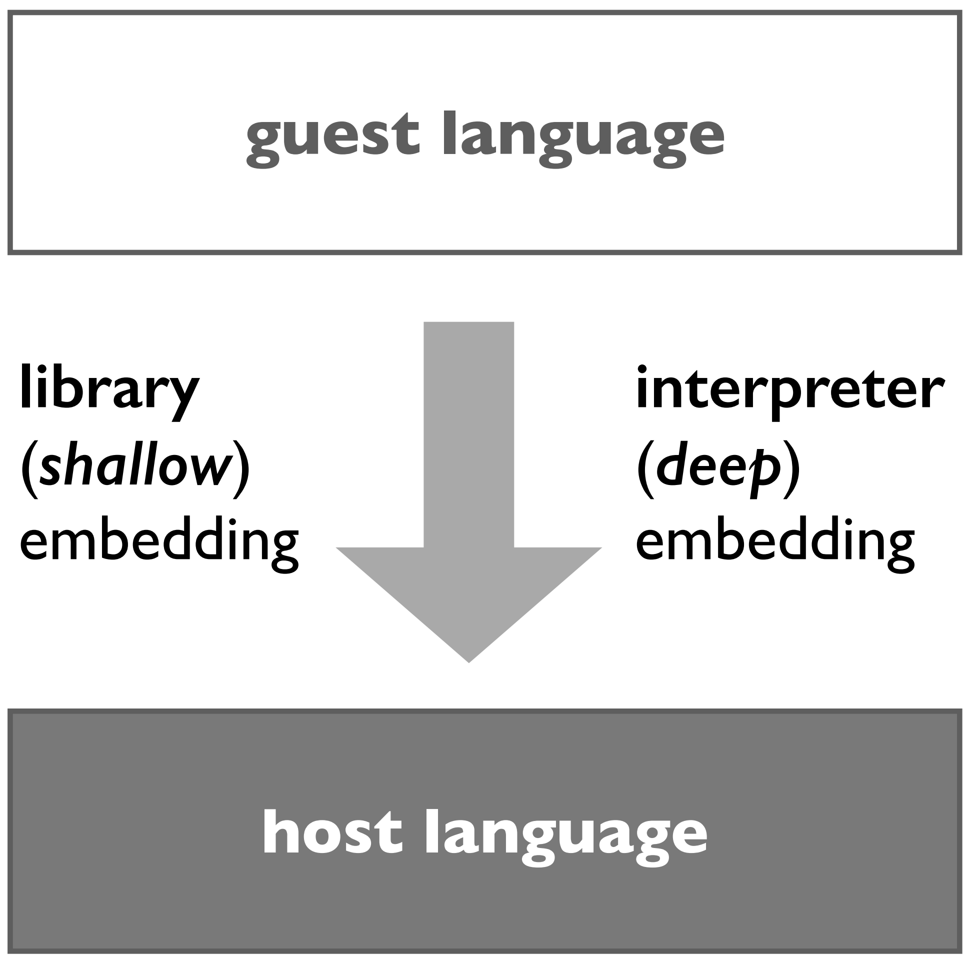
Layers of classic languages: guests and hosts



Usually a domain-specific language (DSL) for expressing and solving a particular class of problems.

A general-purpose high level language, usually with meta-programming features.

Layers of classic languages: many guests and hosts



layout and visualization
LaTex, dot, dygraphs, D3

databases
SQL, Datalog

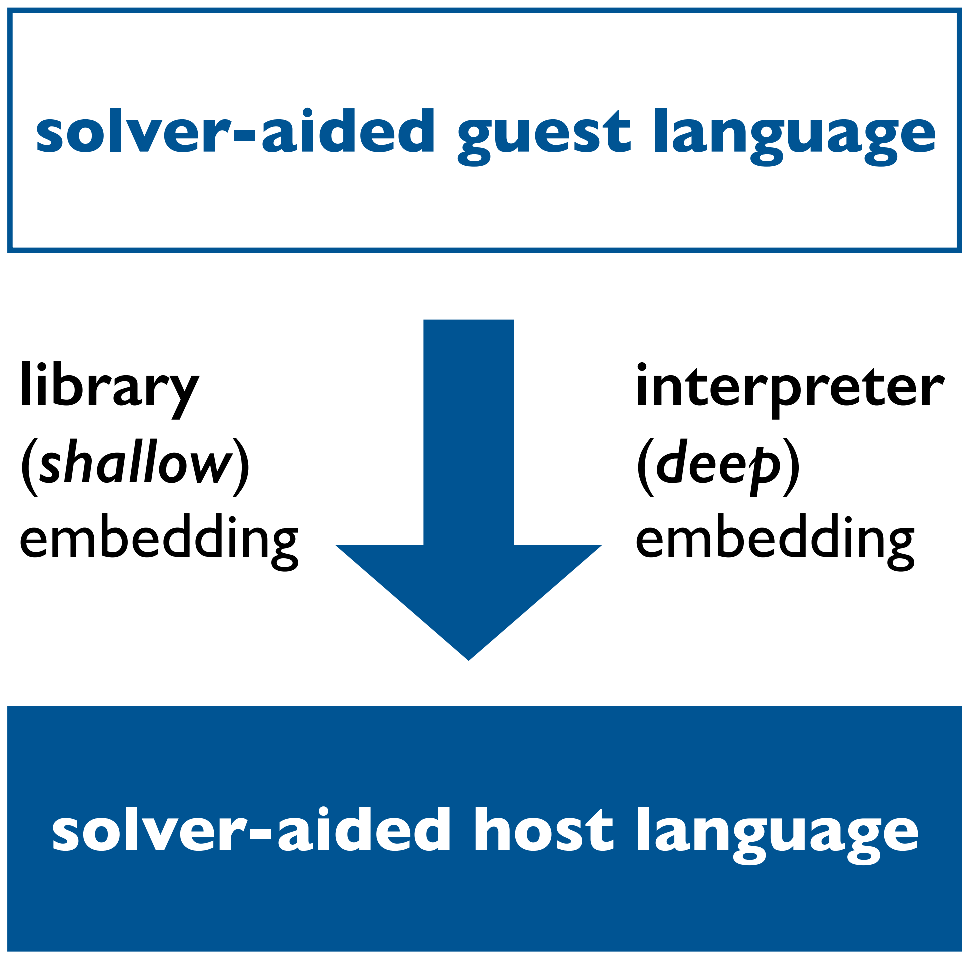
math and statistics
Eigen, Matlab, R

hardware design
Verilog, VHDL, Chisel, Bluespec

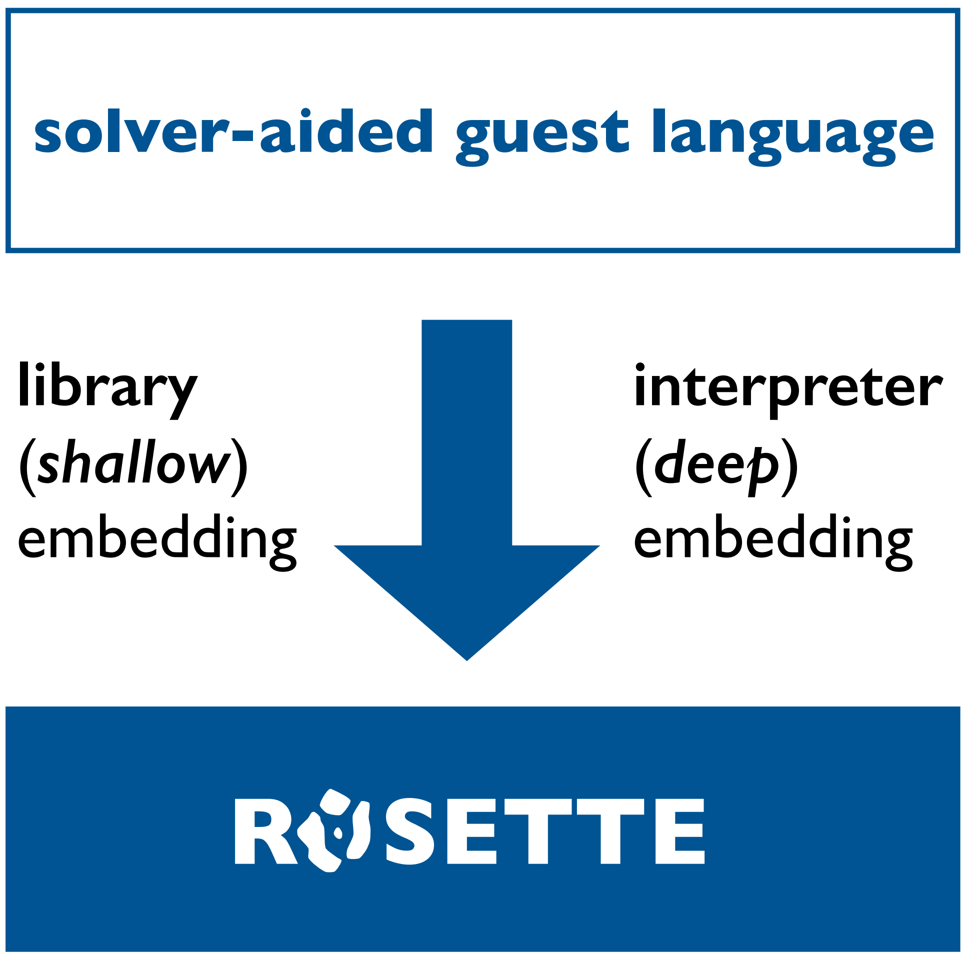
artificial intelligence
Church, BLOG

Racket, Scala, JavaScript, …

Layers of solver-aided languages



Layers of solver-aided languages: tools as languages



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)

computer architecture
Swizzle Inventor (ASPLOS'19)

radiation therapy control
Neutrons (CAV'16)

… and more

The anatomy of a solver-aided host language

An existing host language

Solver-aided constructs

= +

symbolic values
assertions
queries

The anatomy of a solver-aided host language

Racket: "a programming language for creating new programming languages"

= +

A modern descendent of Scheme and Lisp with powerful macro-based meta programming.

symbolic values
assertions
queries

The anatomy of a solver-aided host language

= +

```
(define-symbolic id type)
(define-symbolic* id type)
```

symbolic values

```
(assert expr)
```

assertions

```
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
```

queries

A tiny example solver-aided guest language: BV

```
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 BV.

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]

Interpreting BV

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 ...)
```

Interpreting BV

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
```

```
`(-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)))
```

Interpreting BV

```
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)))
```

Verifying BV

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))))
```

Verifying BV

```
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

ROSETTE

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))))
```

Verifying BV

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.

Verifying BV

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.

Verifying BV

ROSETTE

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

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

query

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

Debugging BV

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))))
```

Synthesizing BV

ROSETTE

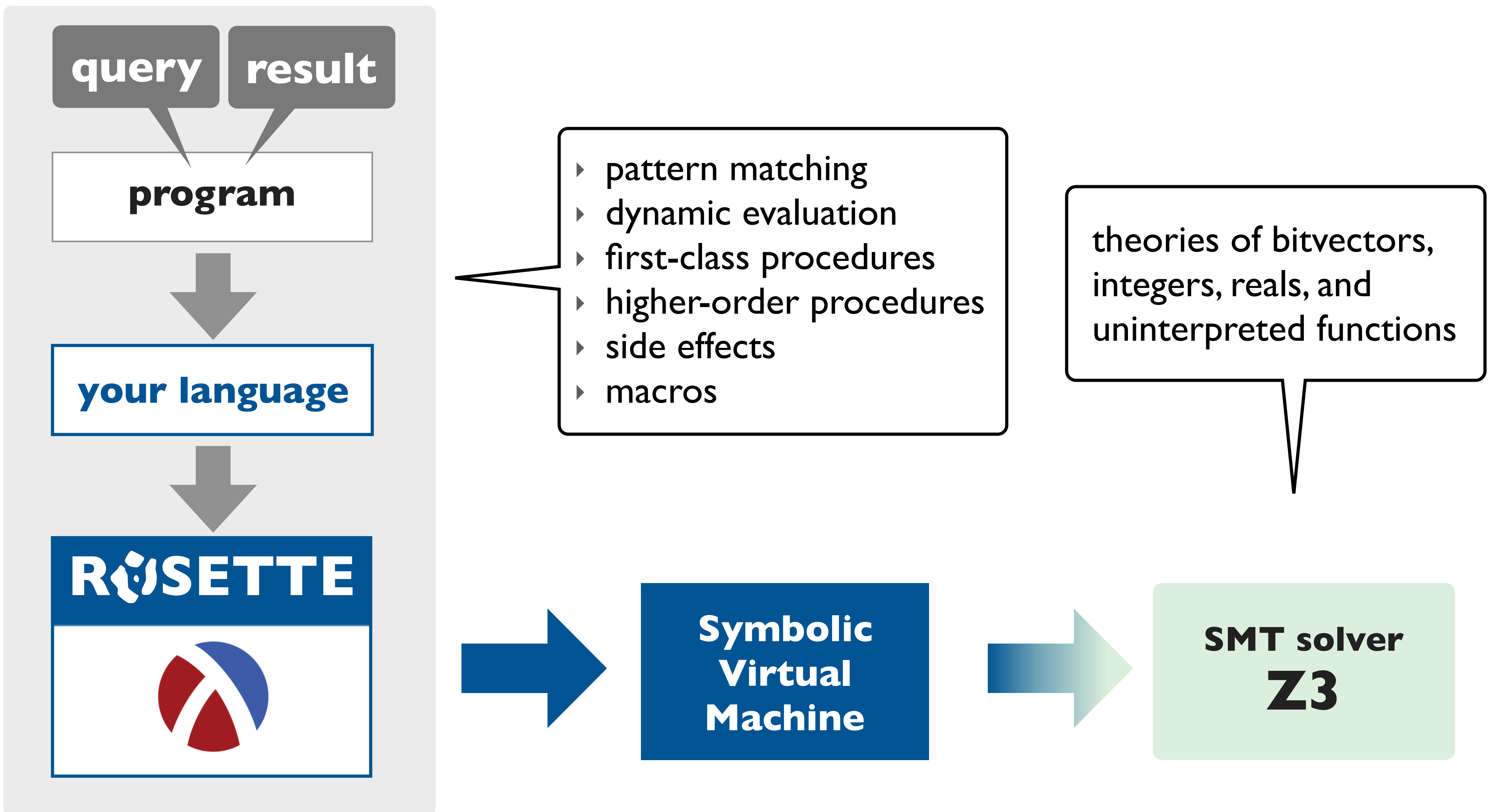
```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r1)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> synthesize(bvmax, max)
```

query

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

How it all works: a big picture view



Translation to constraints by example

vs

`(a, b)`

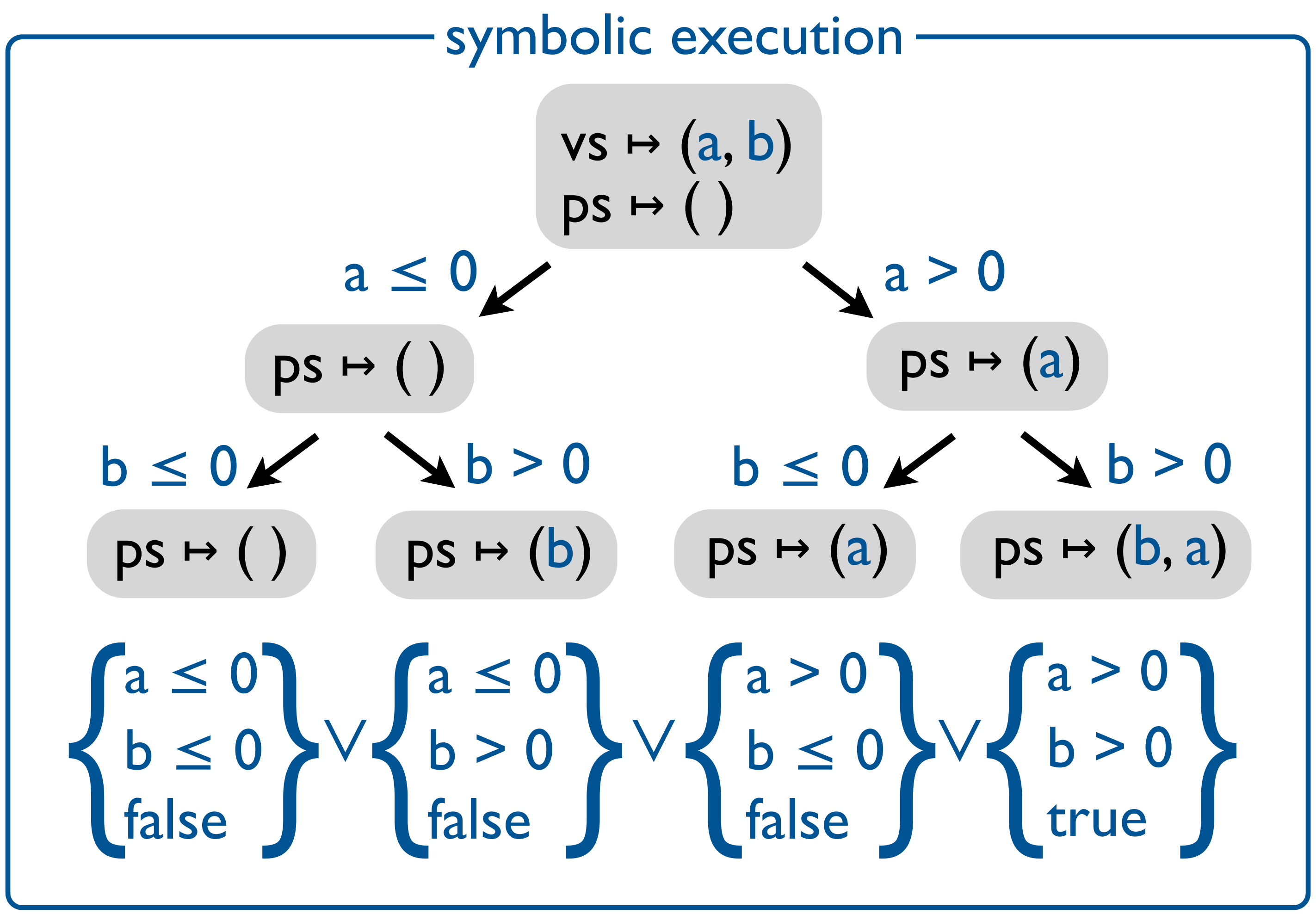
reverse and filter, keeping only positive numbers

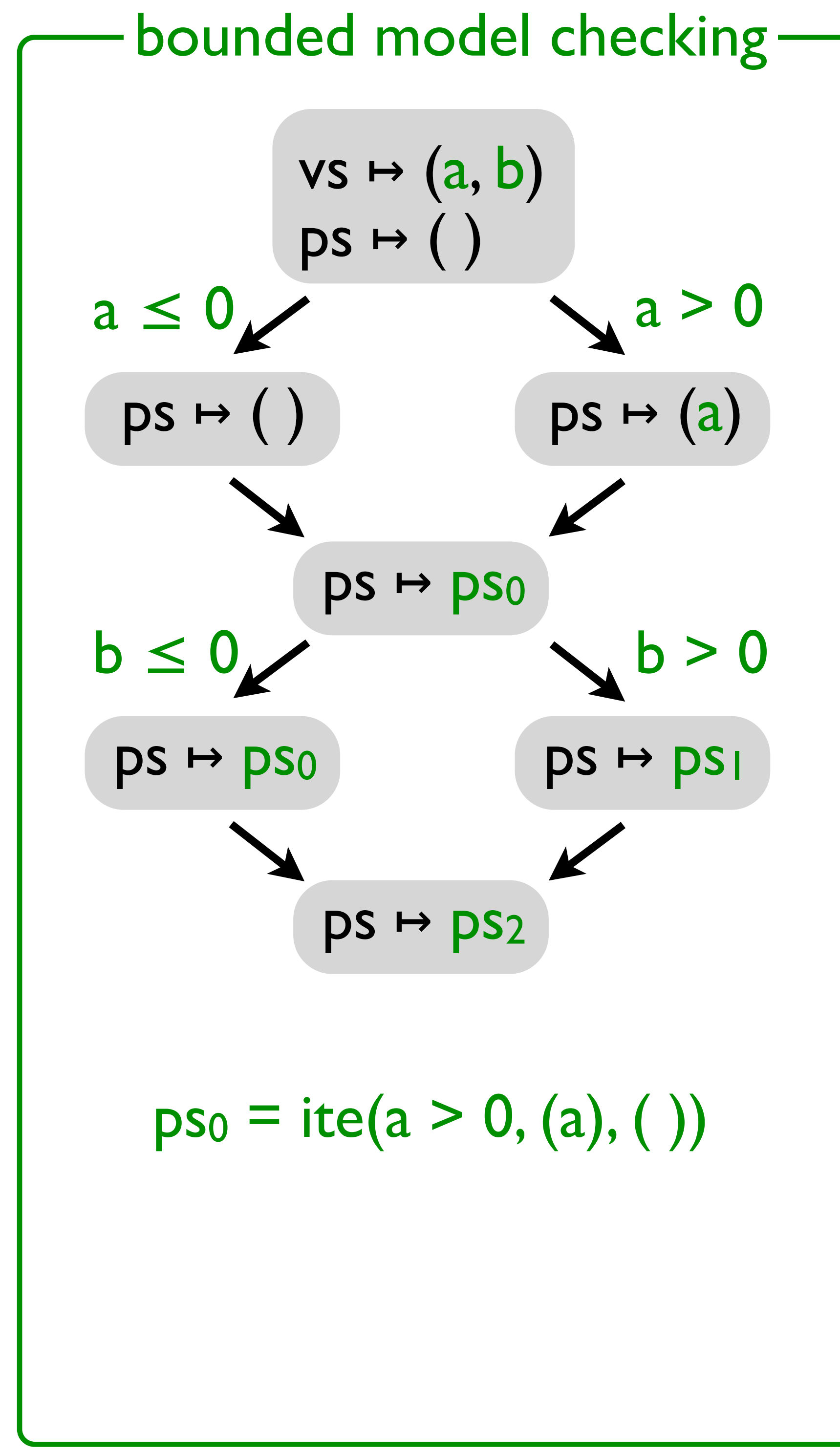
constraints

`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)
```

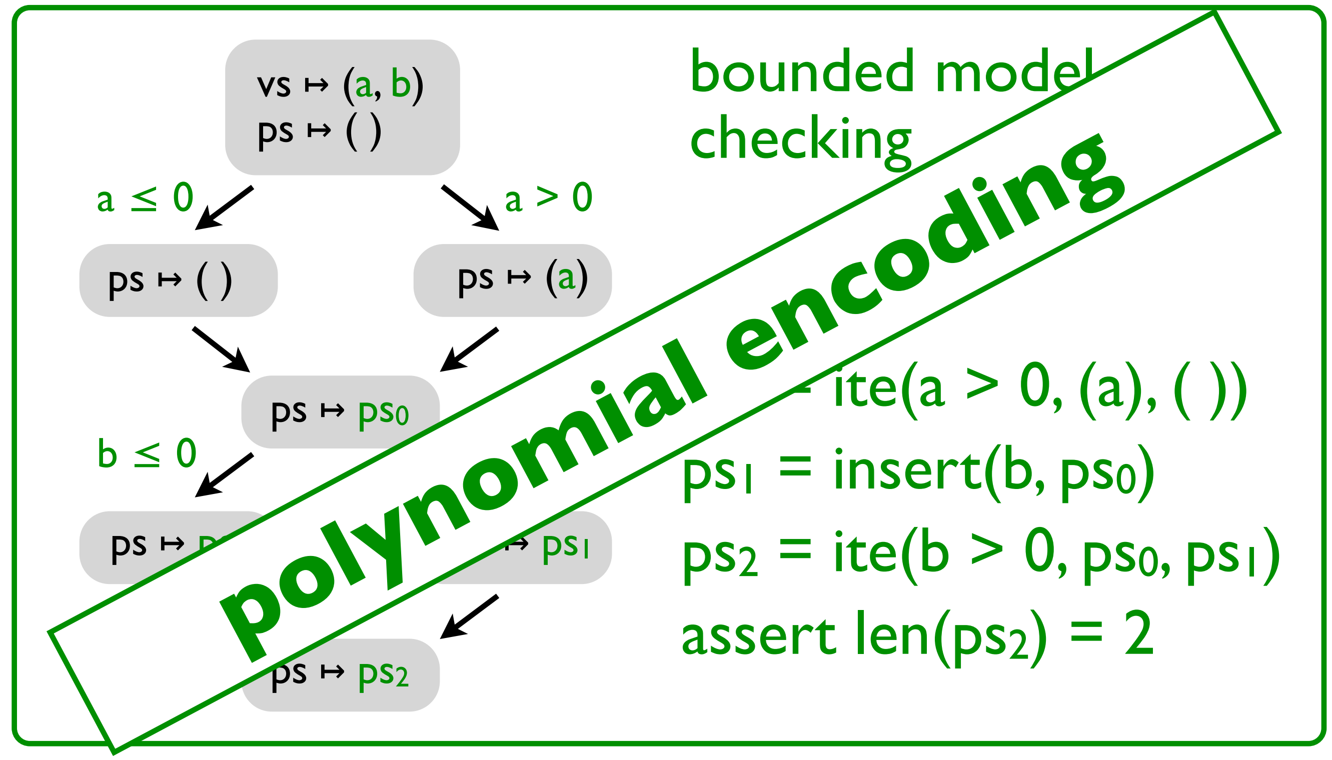


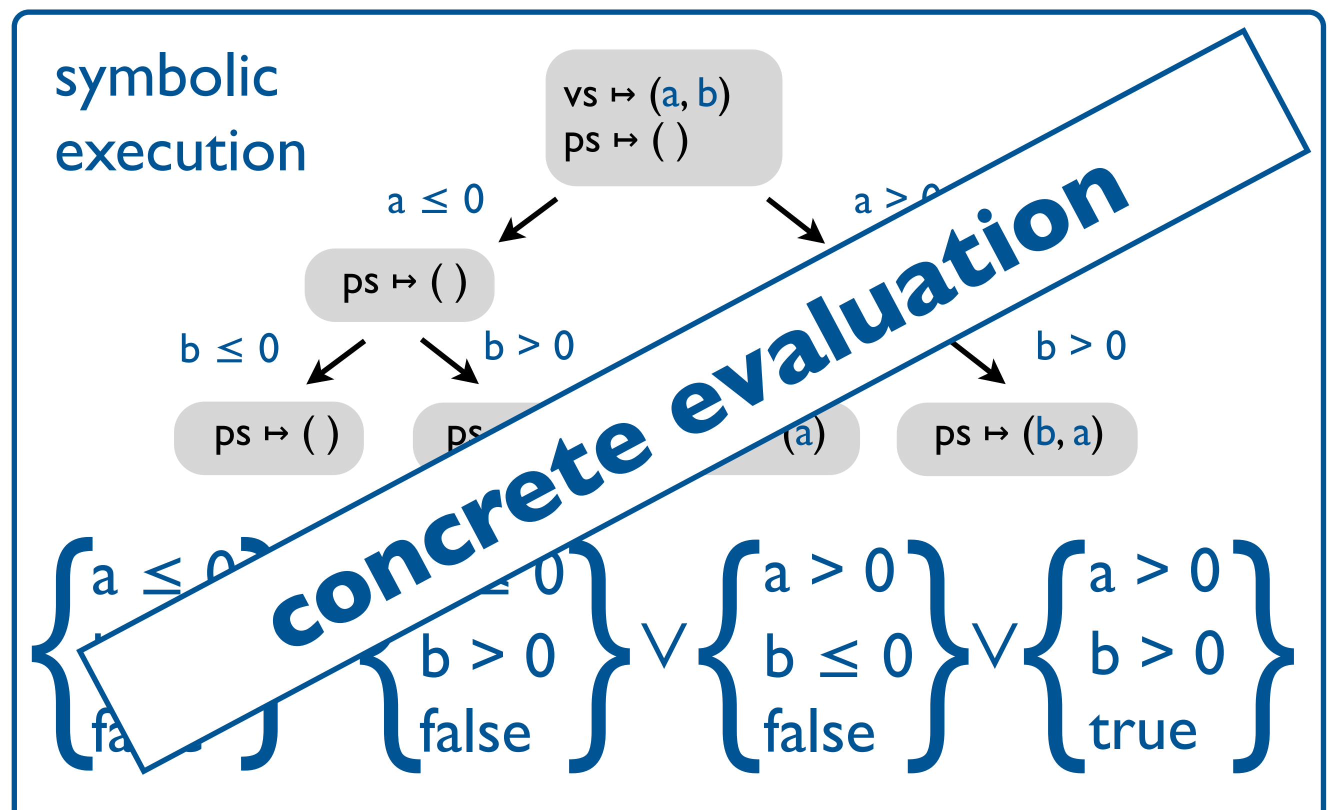


Design space of symbolic encodings: best of all worlds?

```
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)?



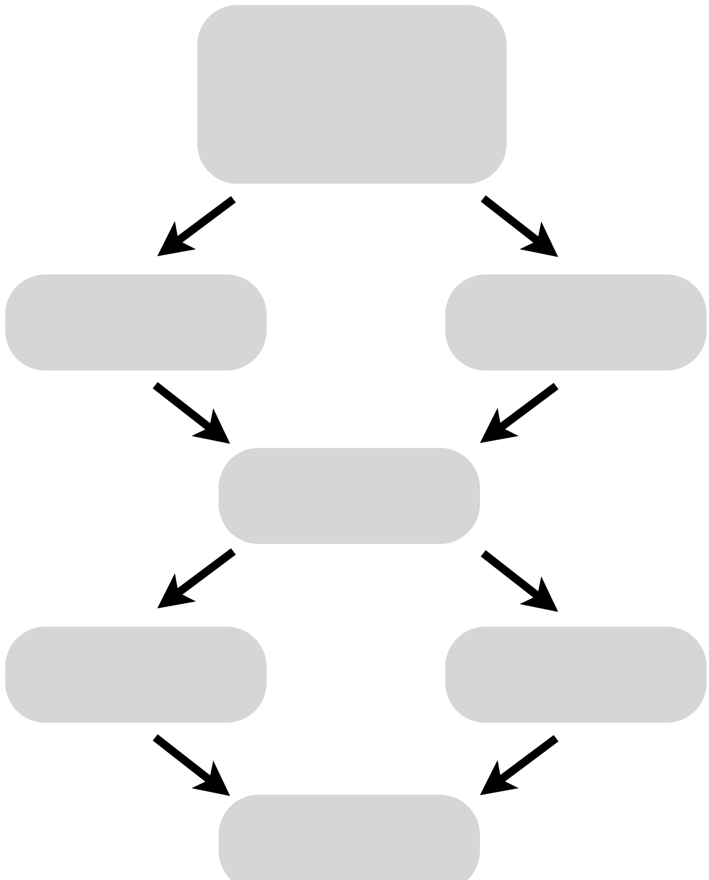


A new design: type-driven state merging

```
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

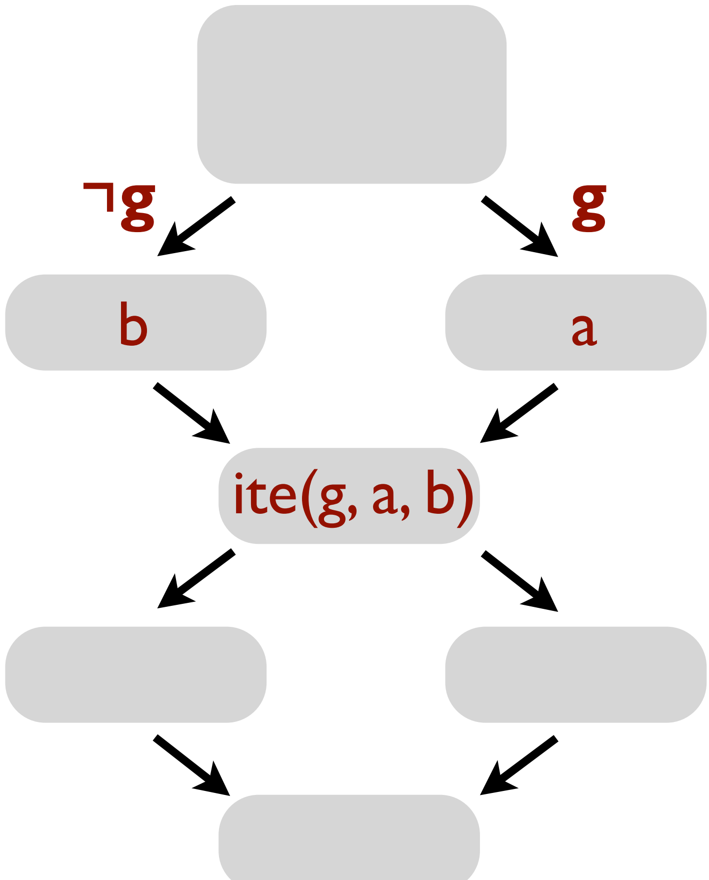


A new design: type-driven state merging

```
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

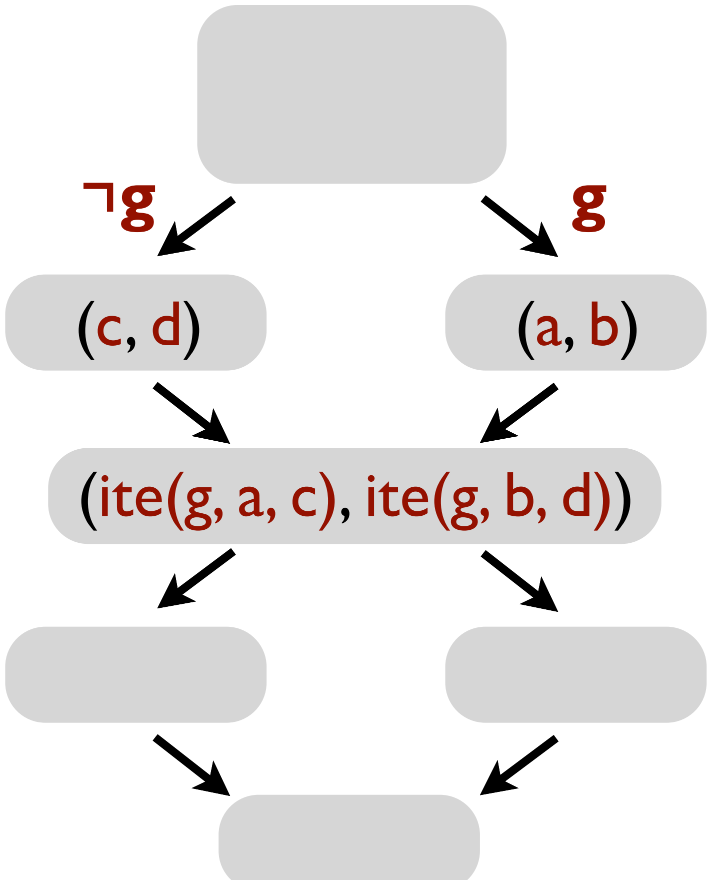


A new design: type-driven state merging

```
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

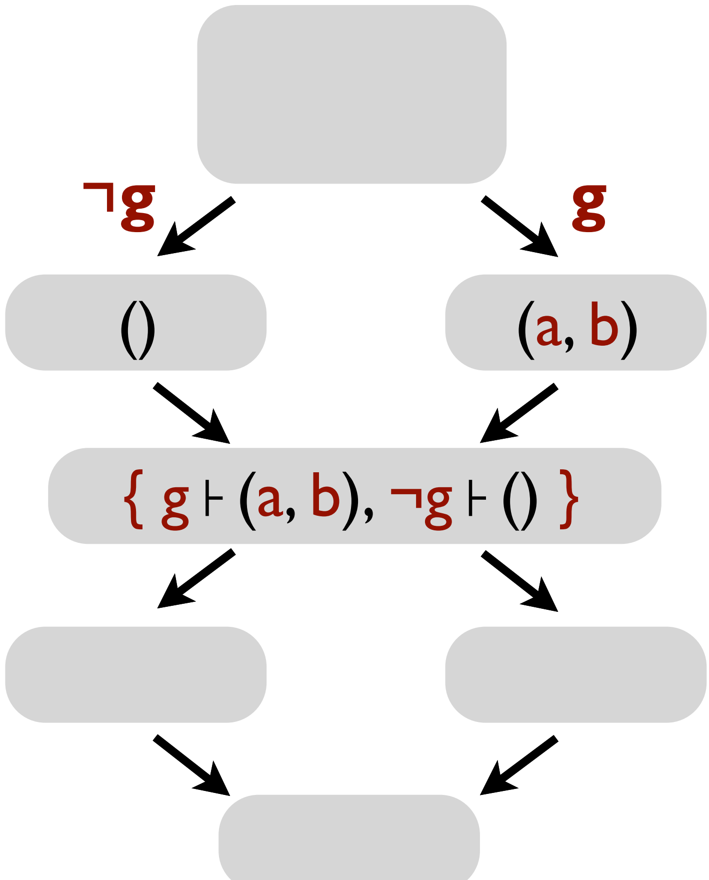


A new design: type-driven state merging

```
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



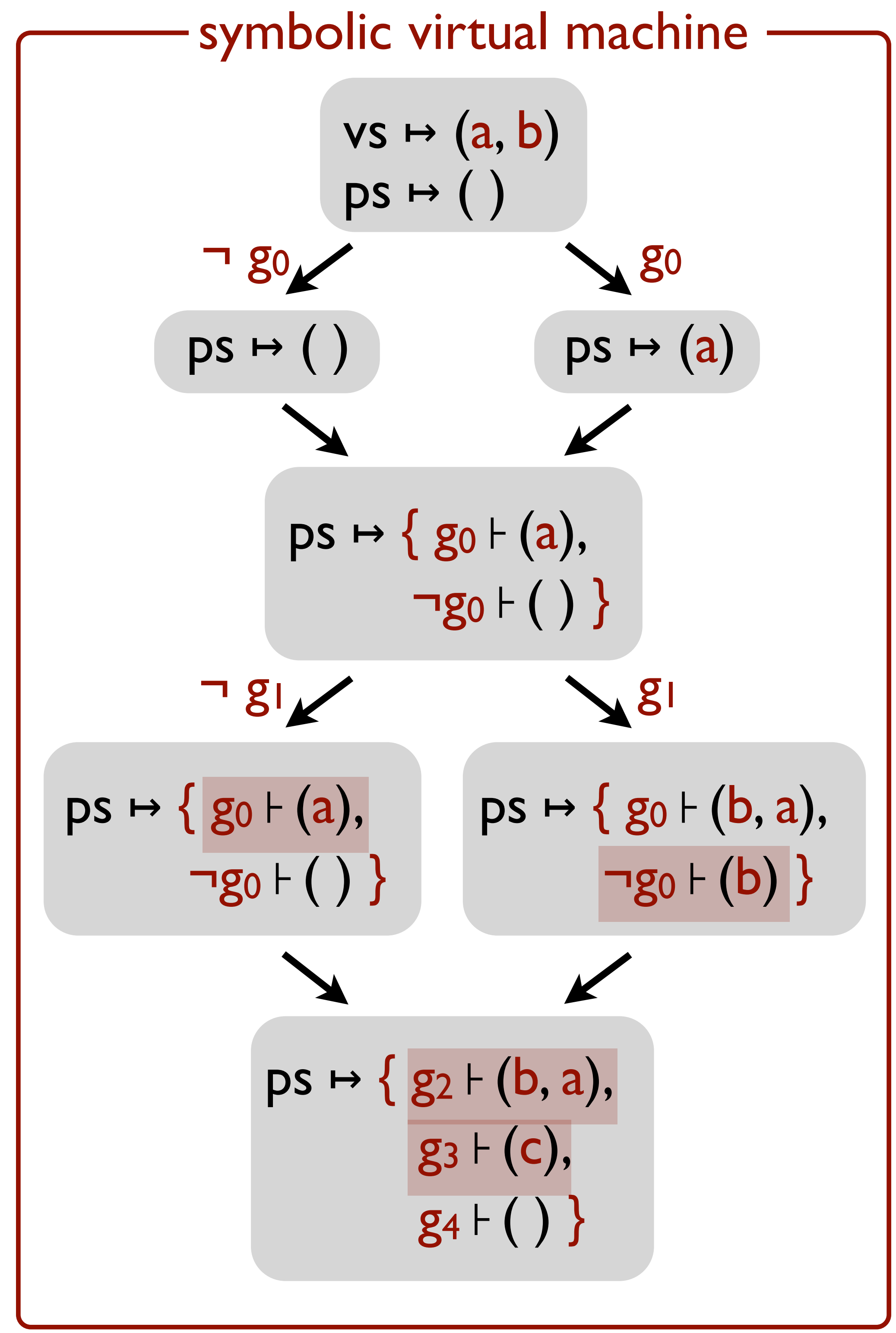
A new design: type-driven state merging

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

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

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

$g_0 = a > 0$



A new design: type-driven state merging

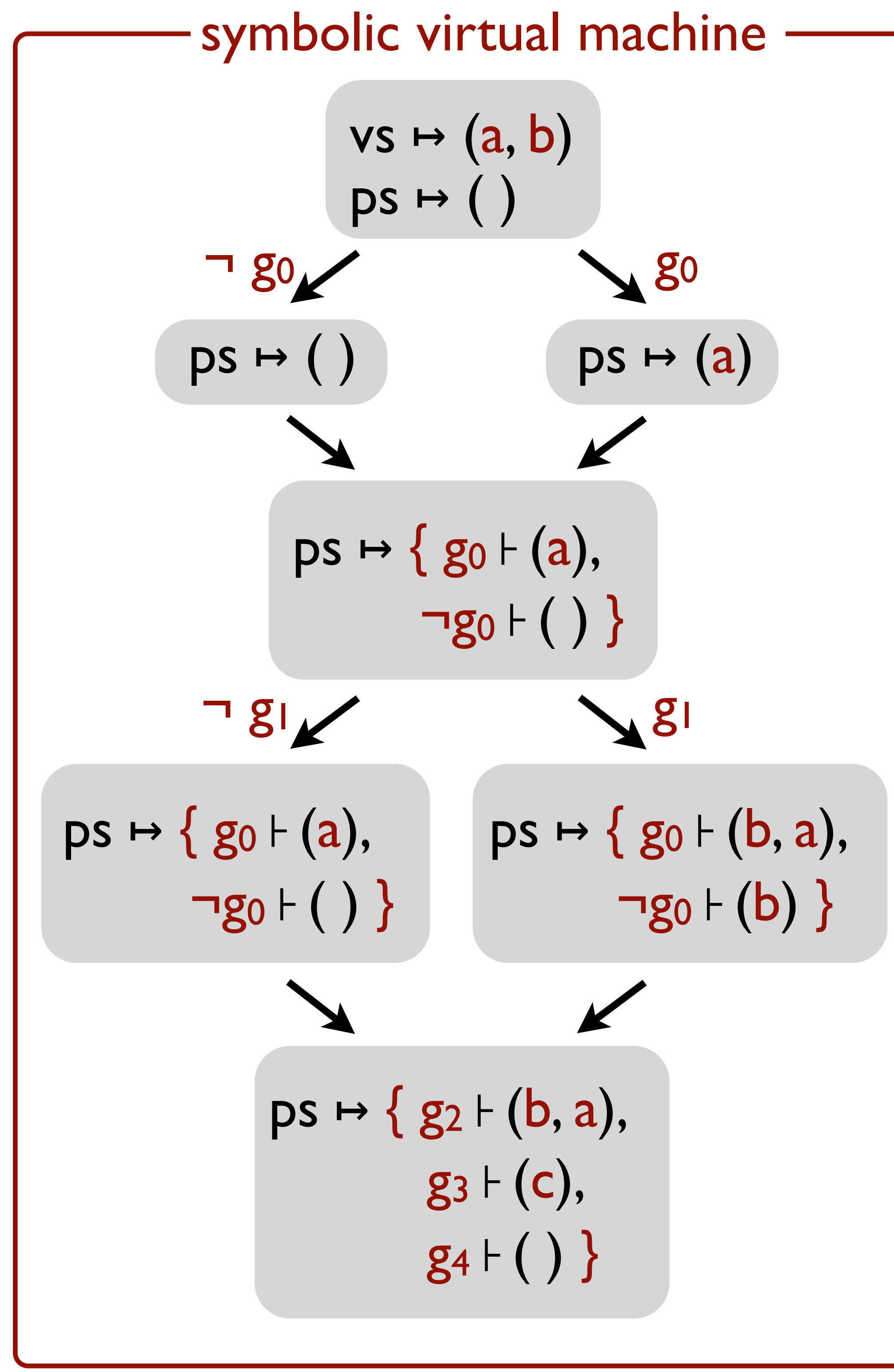
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

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

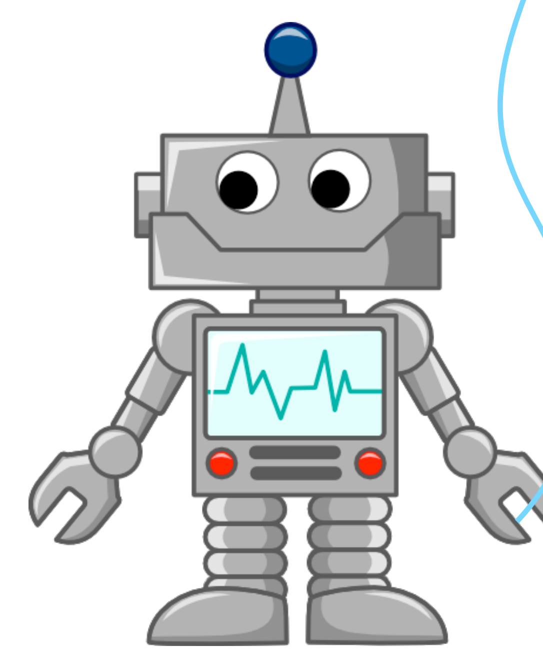


apps

solver-aided applications for all

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

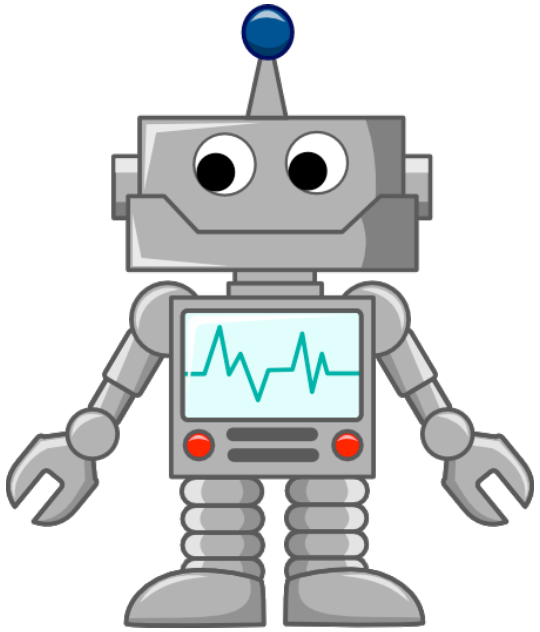
- hints and feedback
- problem generation
- problem-solving strategies
- autograding

apps

solver-aided applications for all

systems, architecture, networks, security, and databases

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

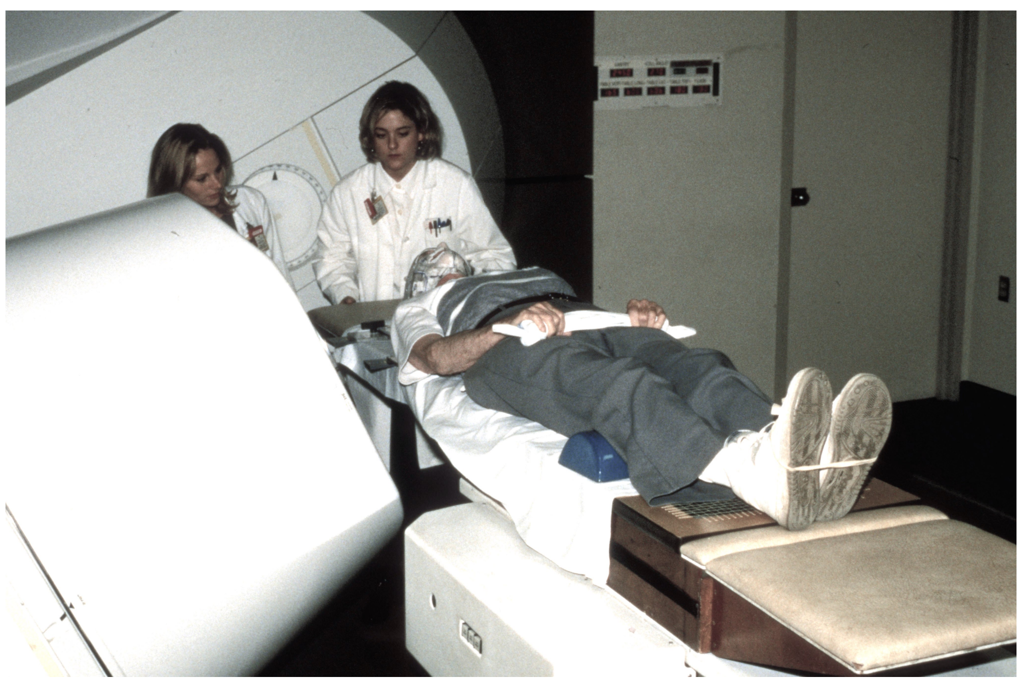


Data movement synthesis for GPU kernels

Swizzle

Verifying a radiation therapy system

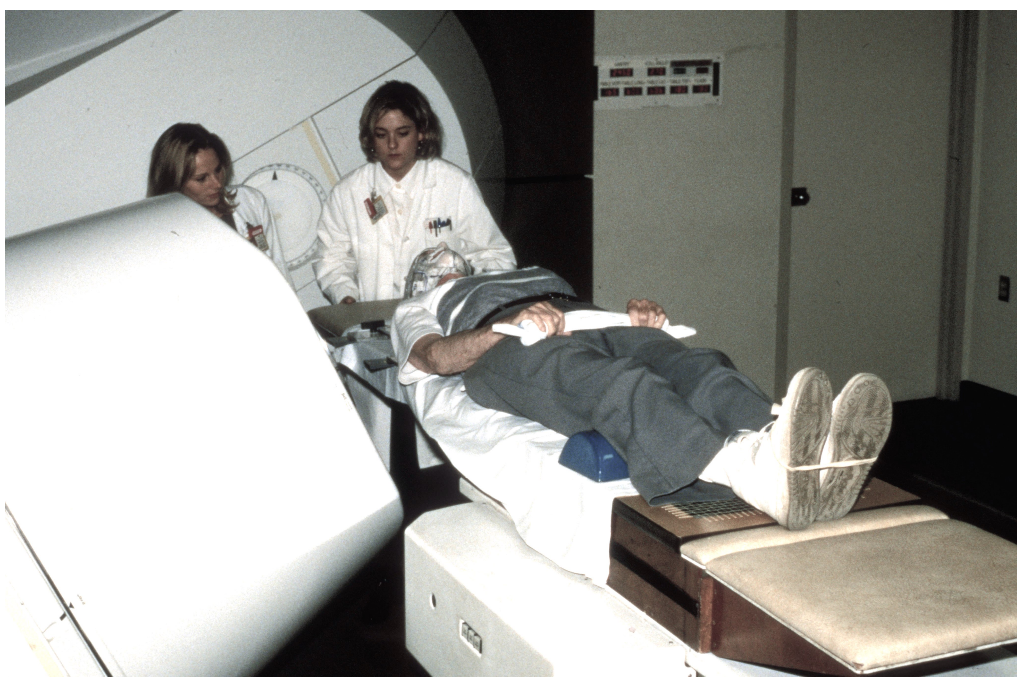
Clinical Neutron Therapy System (CNTS) at UW



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

Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW



Prescription

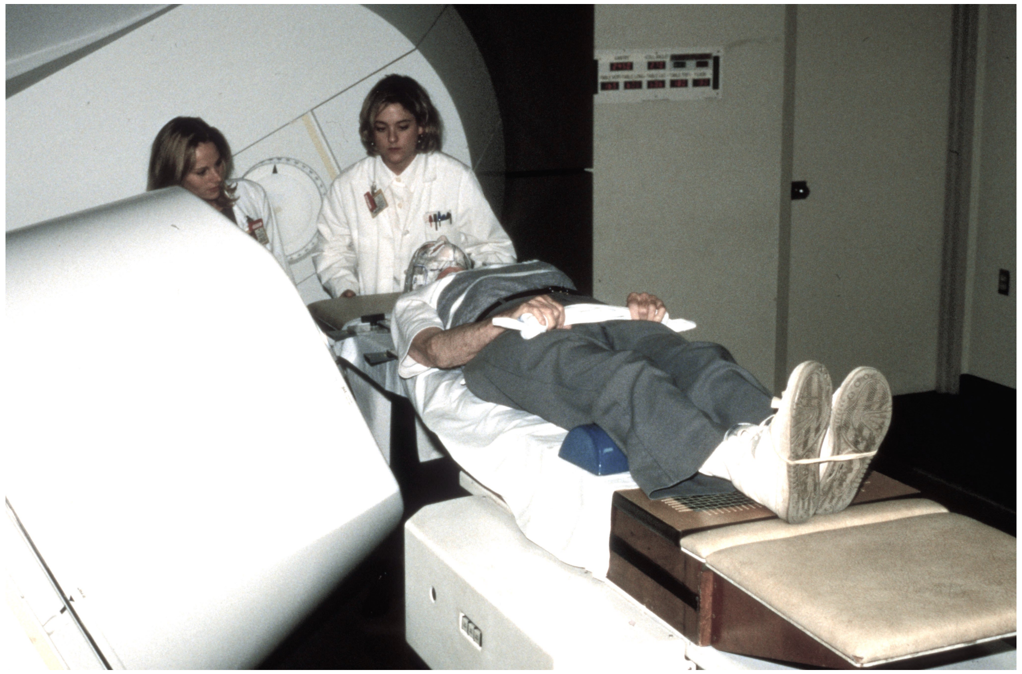
Sensors

Therapy Control Software

Beam, motors, etc.

Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW

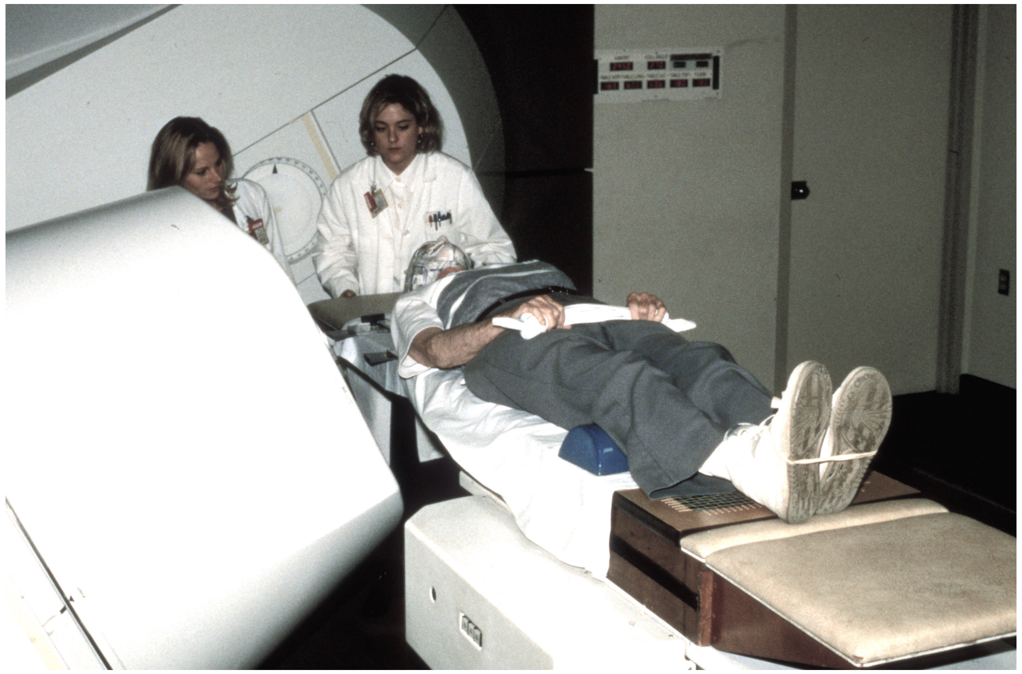


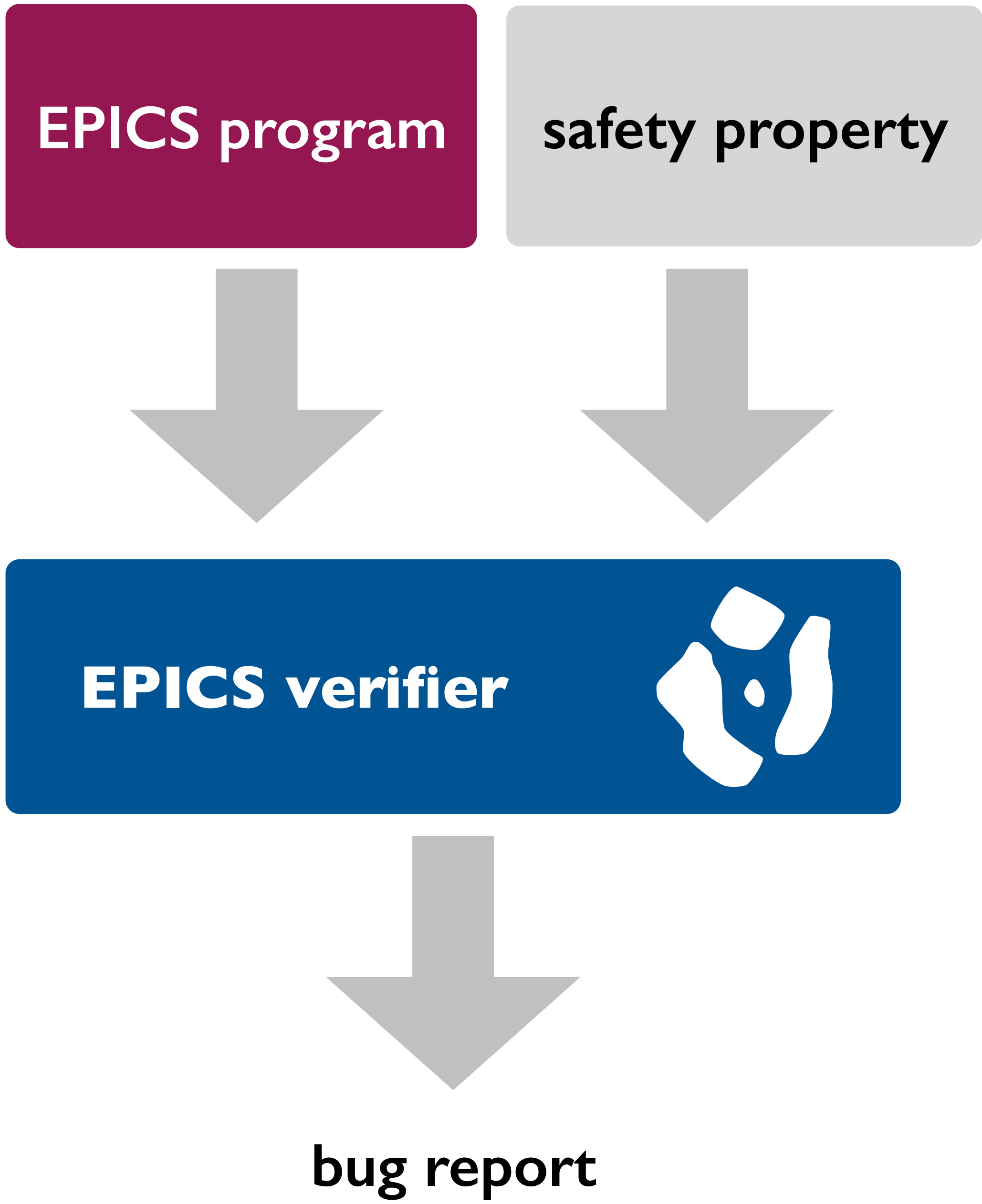
Experimental Physics and Industrial Control System (EPICS) Dataflow Language

Therapy Control Software

Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW





Verifying a radiation therapy system

EPICS program

safety property

EPICS verifier

bug report

Prototyped in a few days and found safety critical bugs.

Calvin Loncaric

Verifying a radiation therapy system

EPICS program

safety property

EPICS verifier

bug report

[Pernsteiner et al., CAV'16, ICALEPCS'17]

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

Used by CNTS staff to verify changes to the controller.

thanks

your SDSL

verify

debug

solve

synthesize

ROSETTE

symbolic virtual machine