Finding Code That Explodes Under Symbolic Evaluation

James Bornholt and **Emina Torlak**

University of Washington

A new technique for helping programmers build performant solver-aided tools

A new technique for helping programmers build performant solver-aided tools

Background

A new technique for helping programmers build performant solver-aided tools

Problem

Background

symbolic profiling

Approach

A new technique for helping programmers build performant solver-aided tools

Problem

Background

symbolic profiling

Approach

Evaluation

A new technique for helping programmers build performant solver-aided tools

Problem

Background

symbolic
profiling

Approach

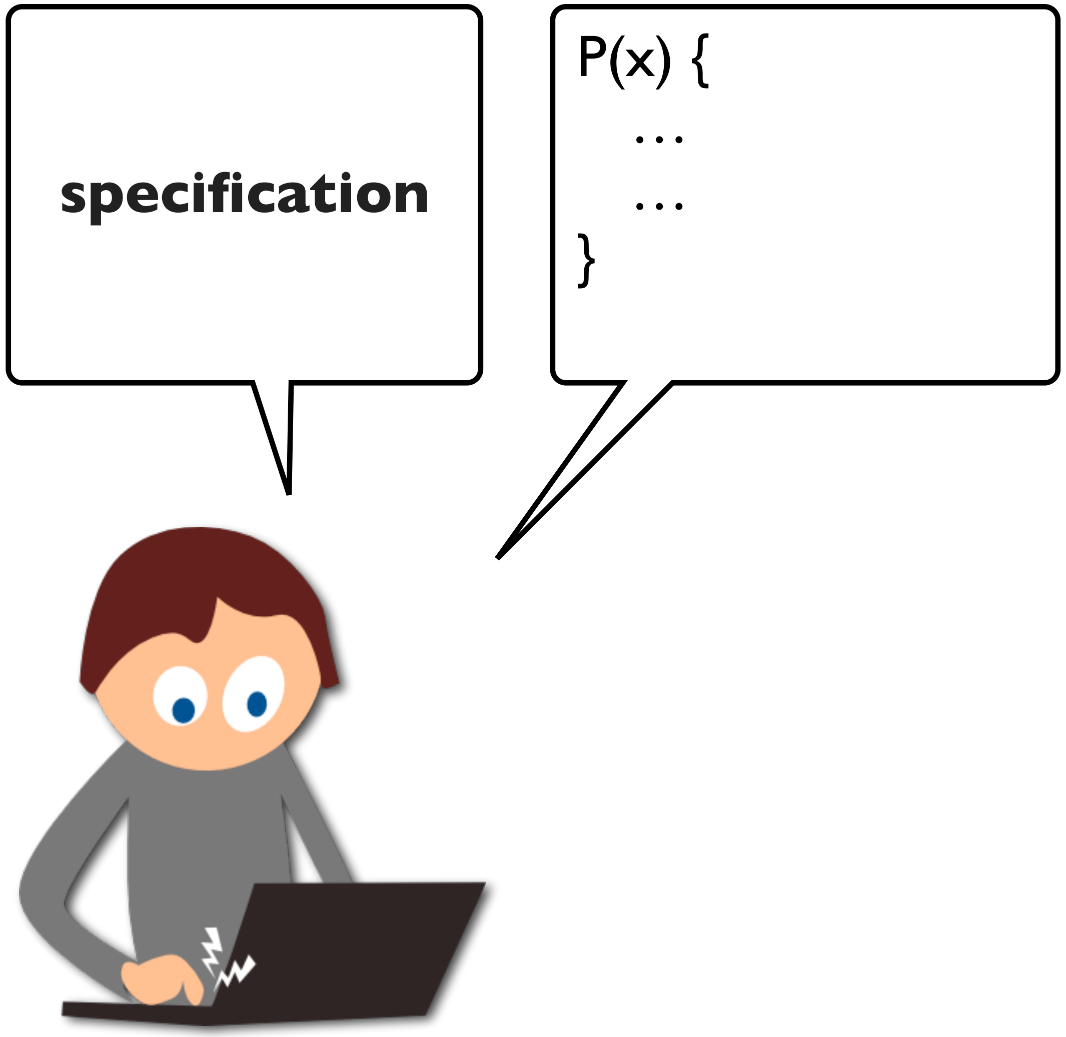
Evaluation

A new technique for helping programmers build performant **solver-aided tools**

Problem

Background

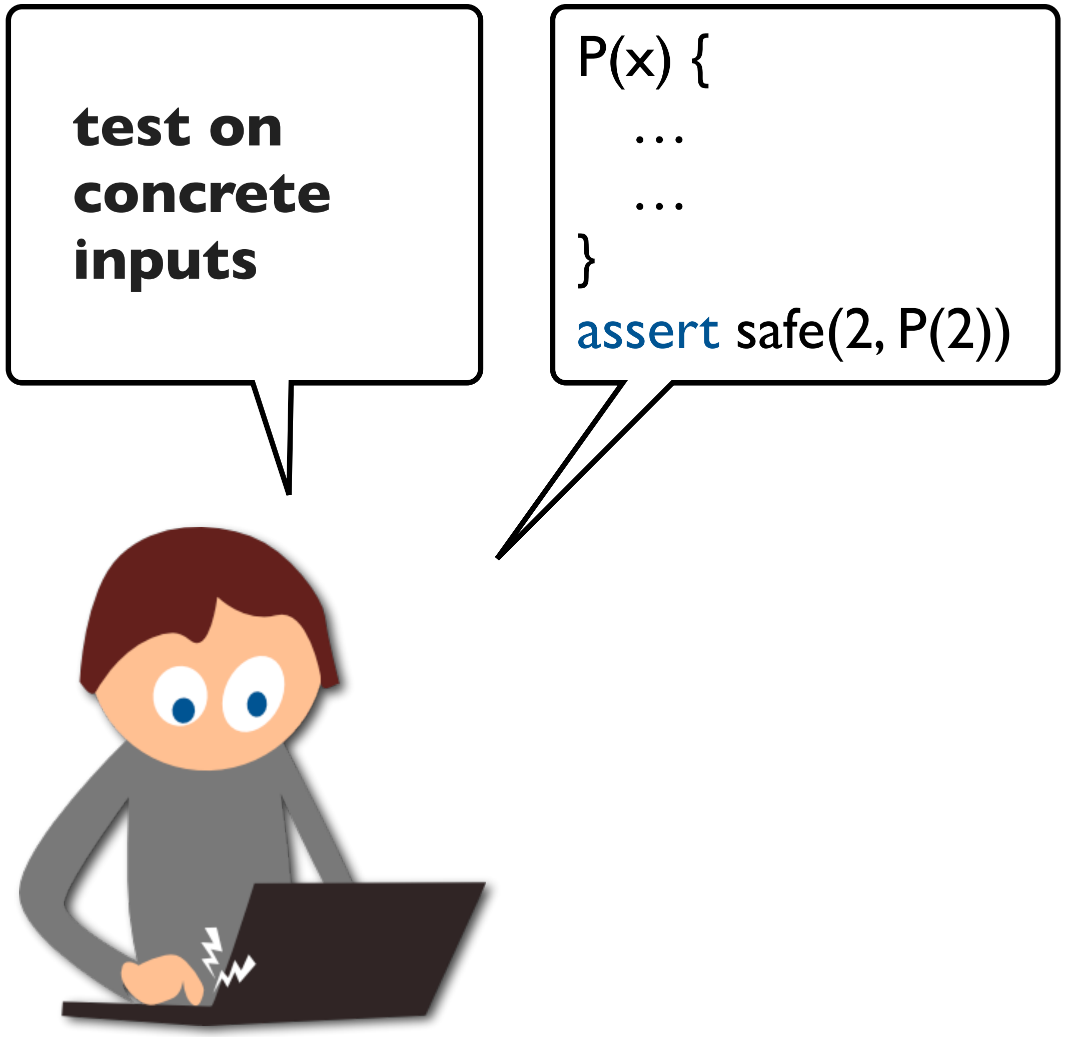
Programming ...



specification

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

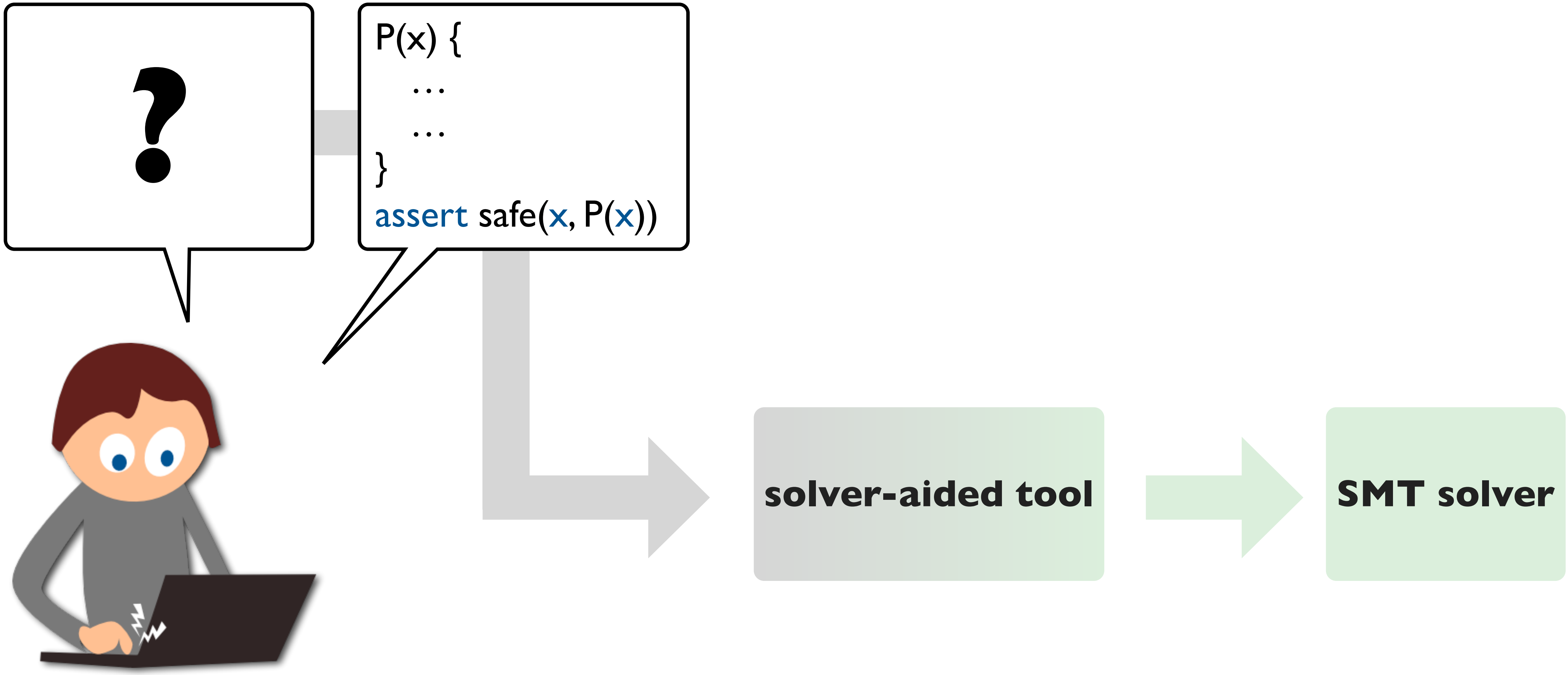
Programming …



test on concrete inputs

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

Programming with a solver-aided tool

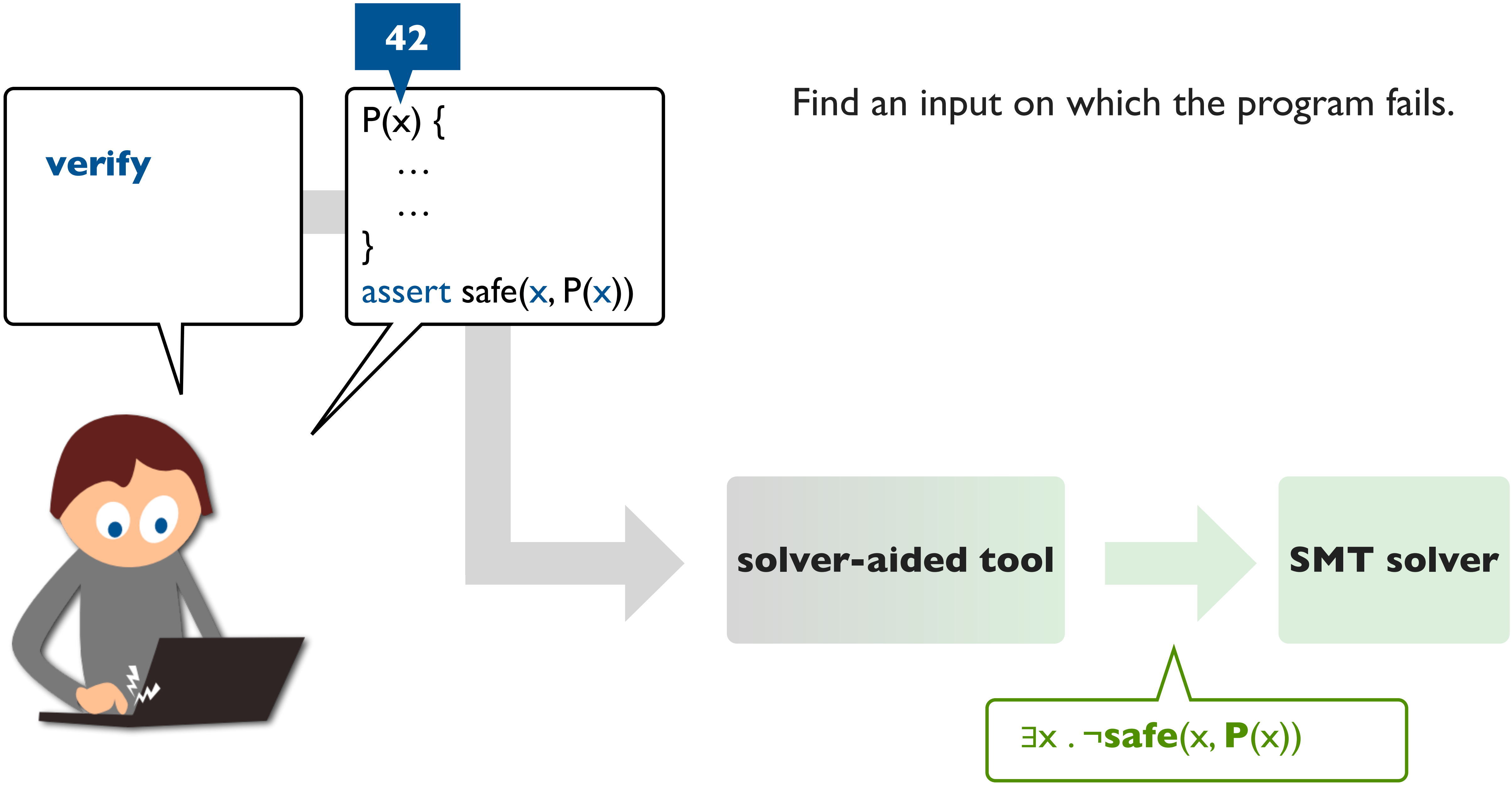


```
P(x) {
    …
    …
}
assert safe(x, P(x))
```

solver-aided tool

SMT solver

Programming with a solver-aided tool



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

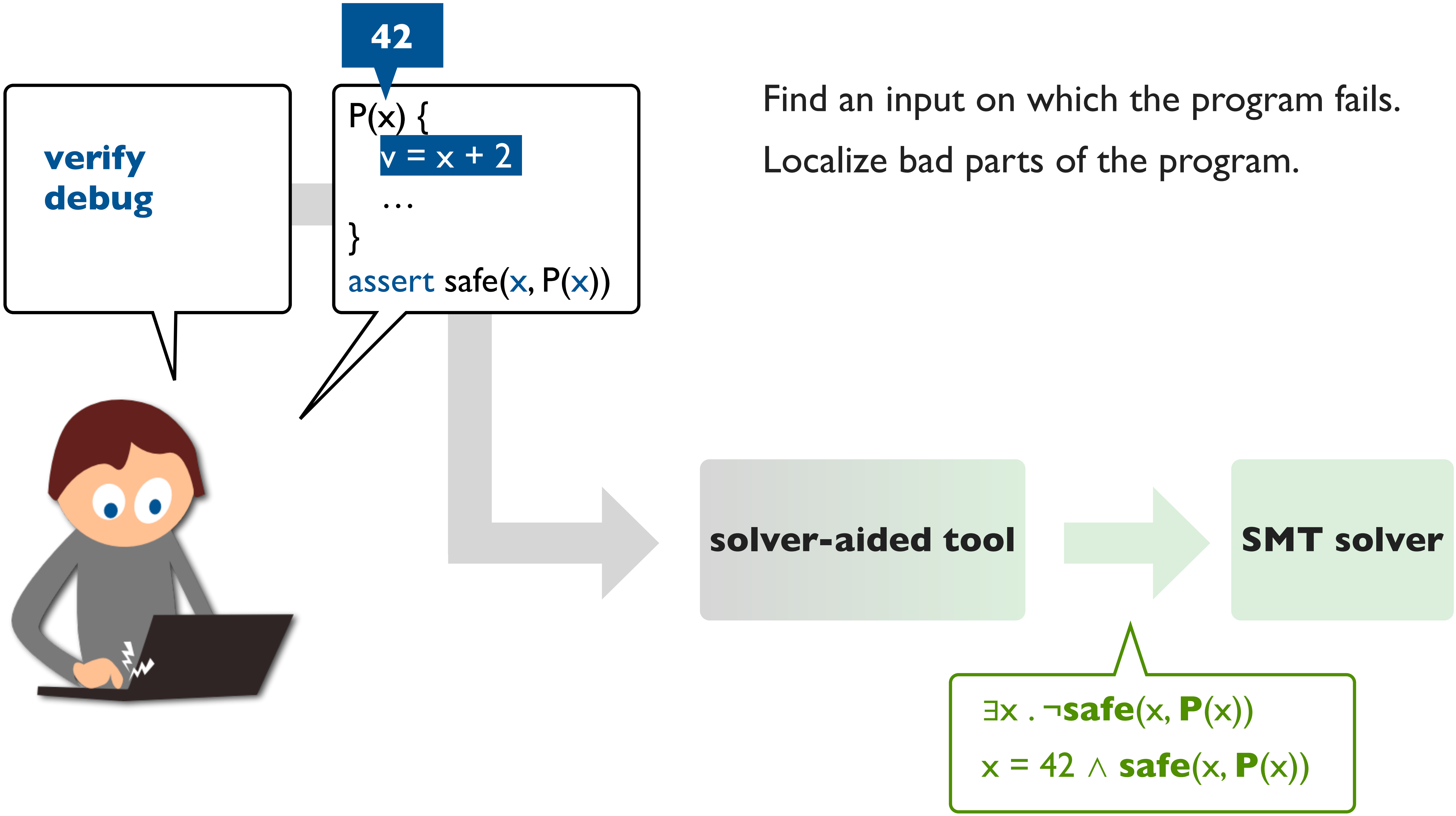
Find an input on which the program fails.

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

Programming with a solver-aided tool

Find an input on which the program fails.

Localize bad parts of the program.

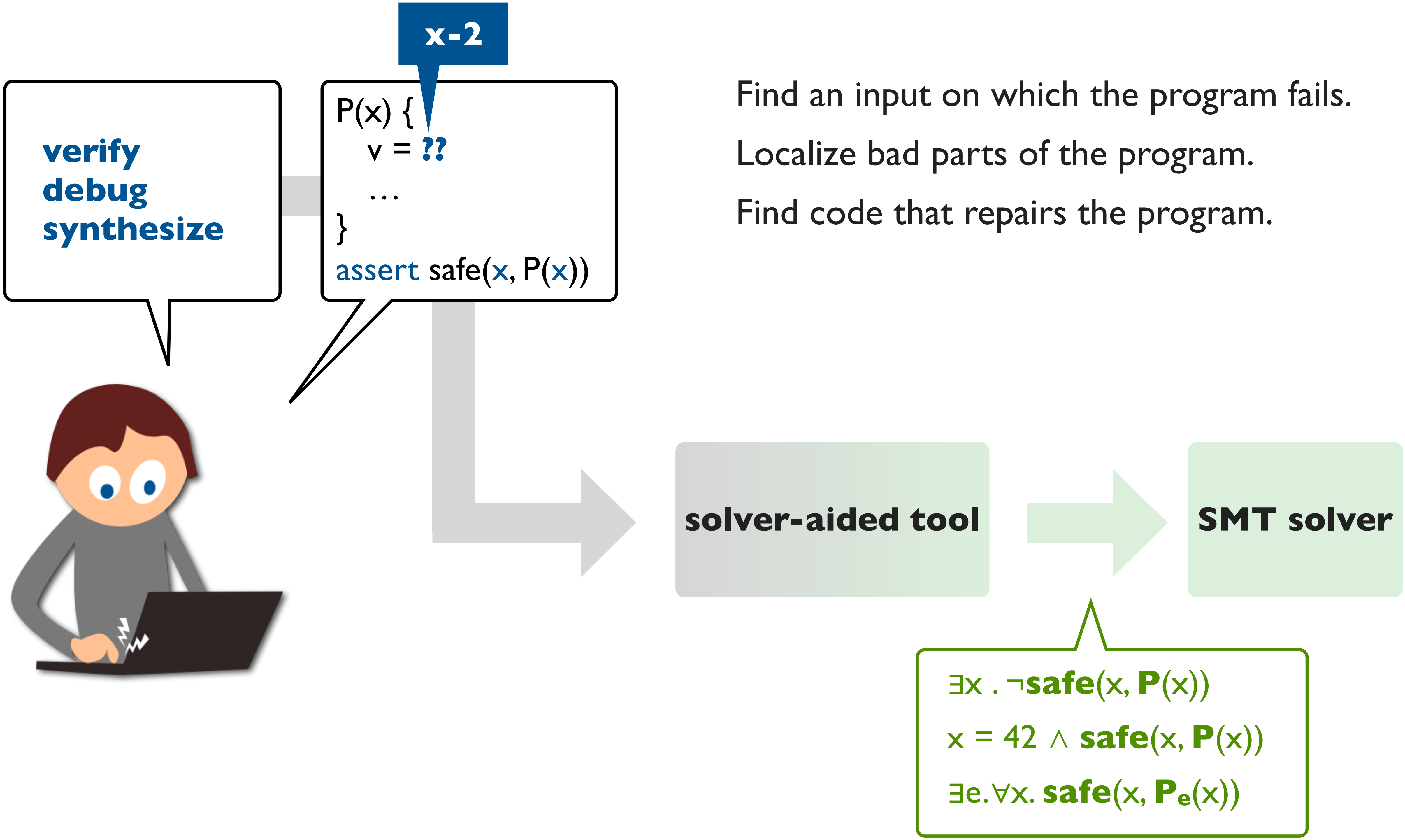


```
P(x) {
  v = x + 2
  …
}
assert safe(x, P(x))
```

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

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

Programming with a solver-aided tool



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

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

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

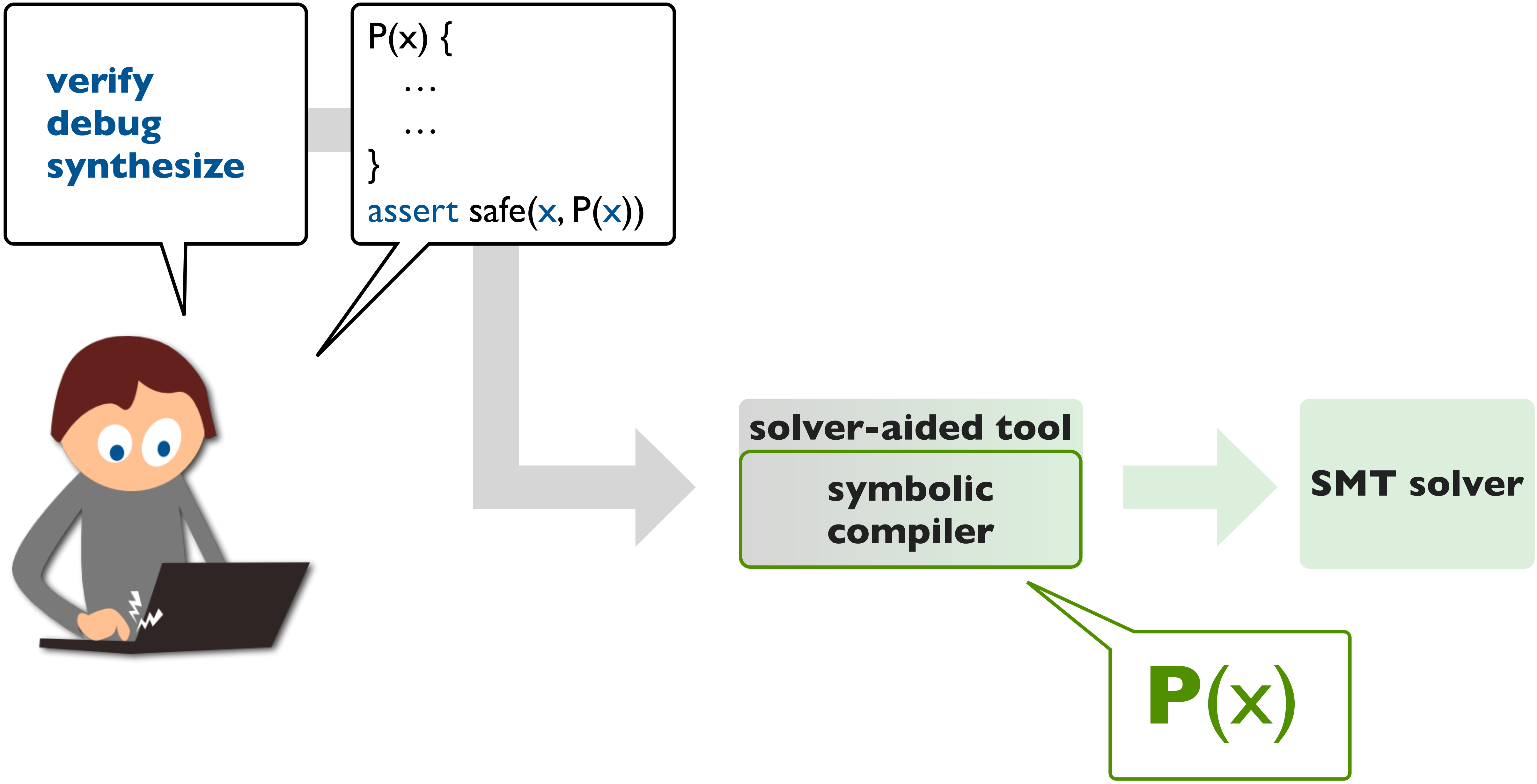
$\exists e. \forall x.\ \mathbf{safe}(x, \mathbf{P}_e(x))$

Find an input on which the program fails.

Localize bad parts of the program.

Find code that repairs the program.

The classic (hard) way to build a tool



verify
debug
synthesize

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

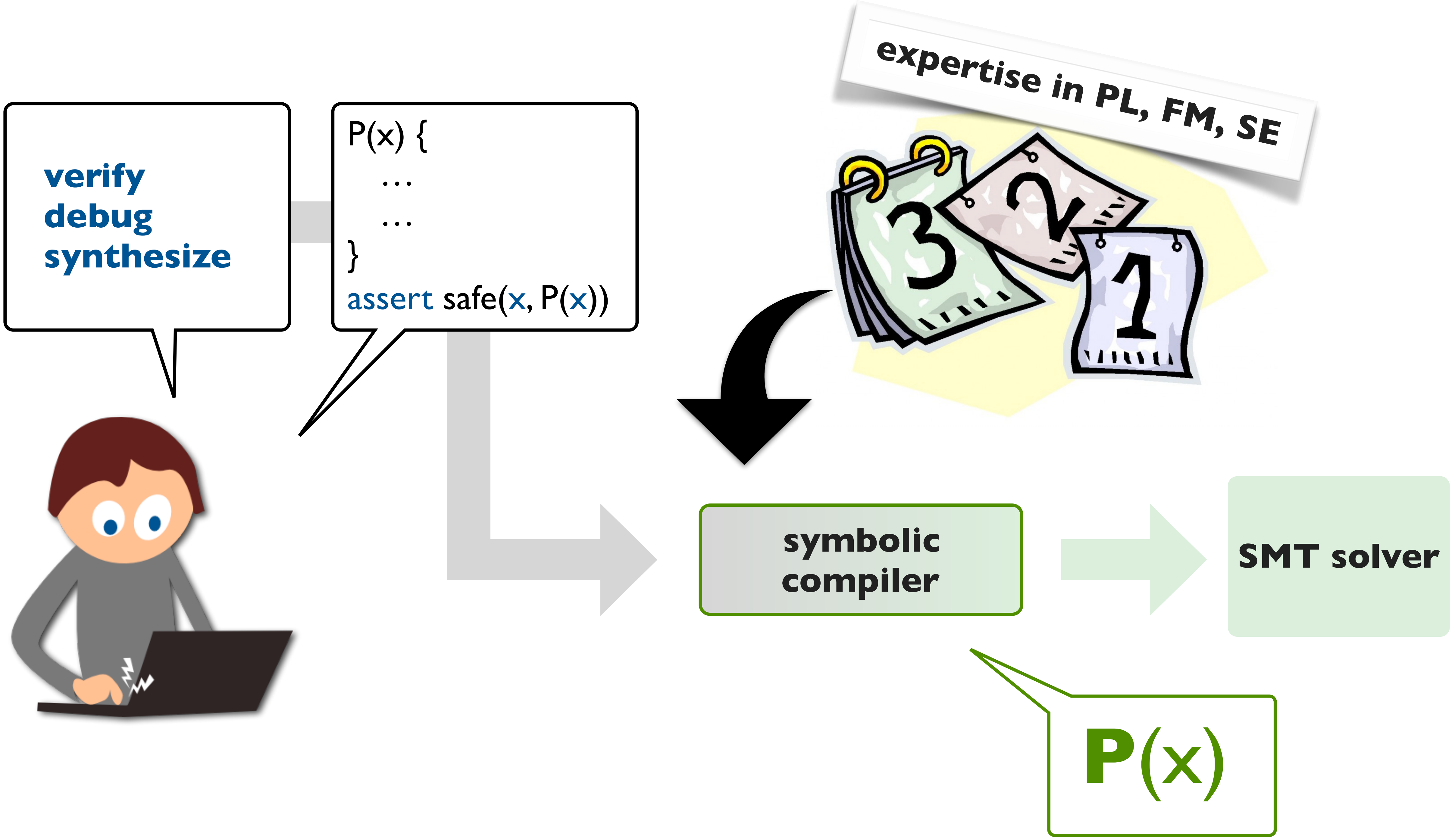
solver-aided tool

symbolic compiler

SMT solver

P(x)

The classic (hard) way to build a tool



verify
debug
synthesize

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

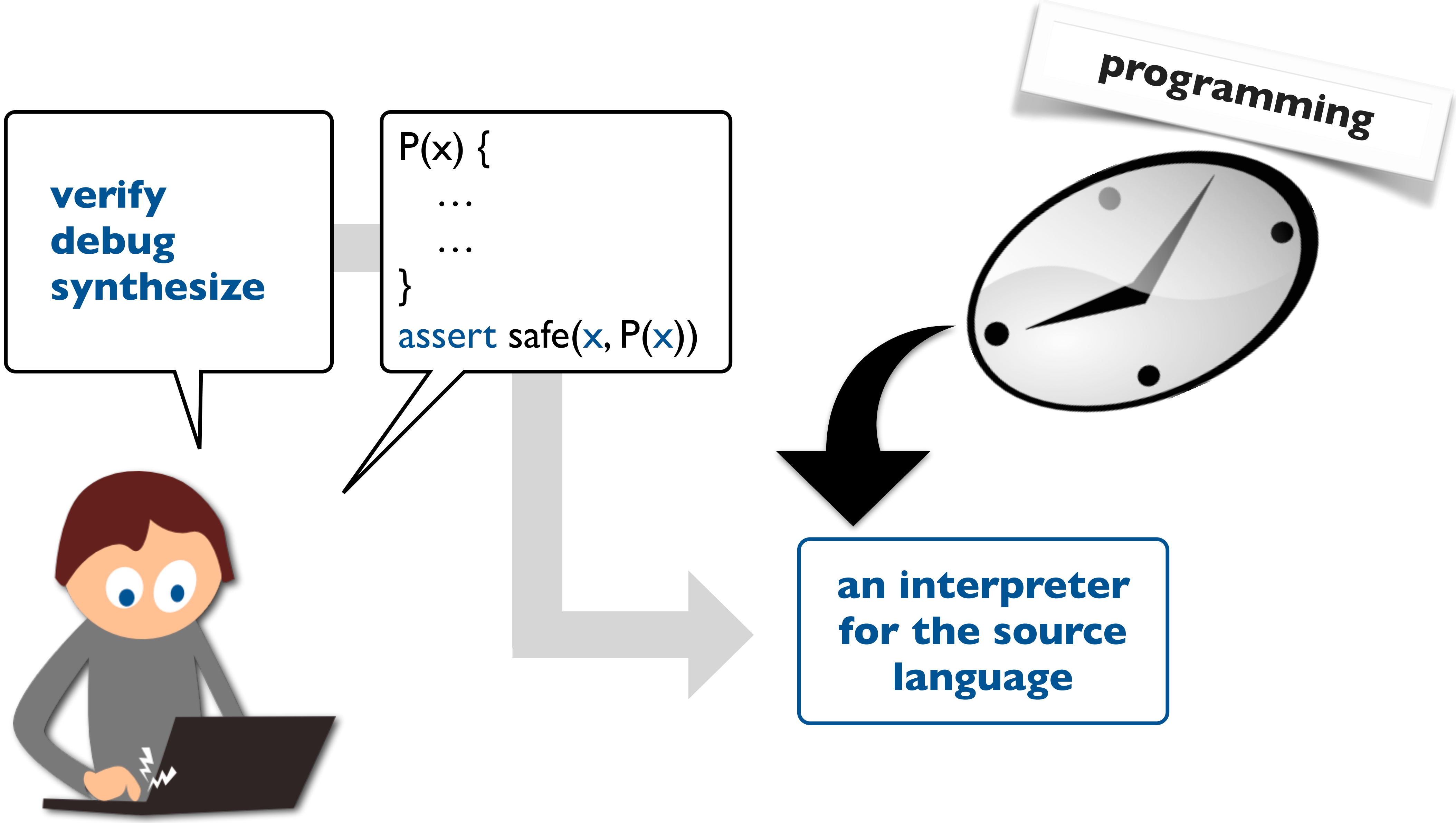
expertise in PL, FM, SE

symbolic compiler

SMT solver

P(x)

A newer, easier way to build tools



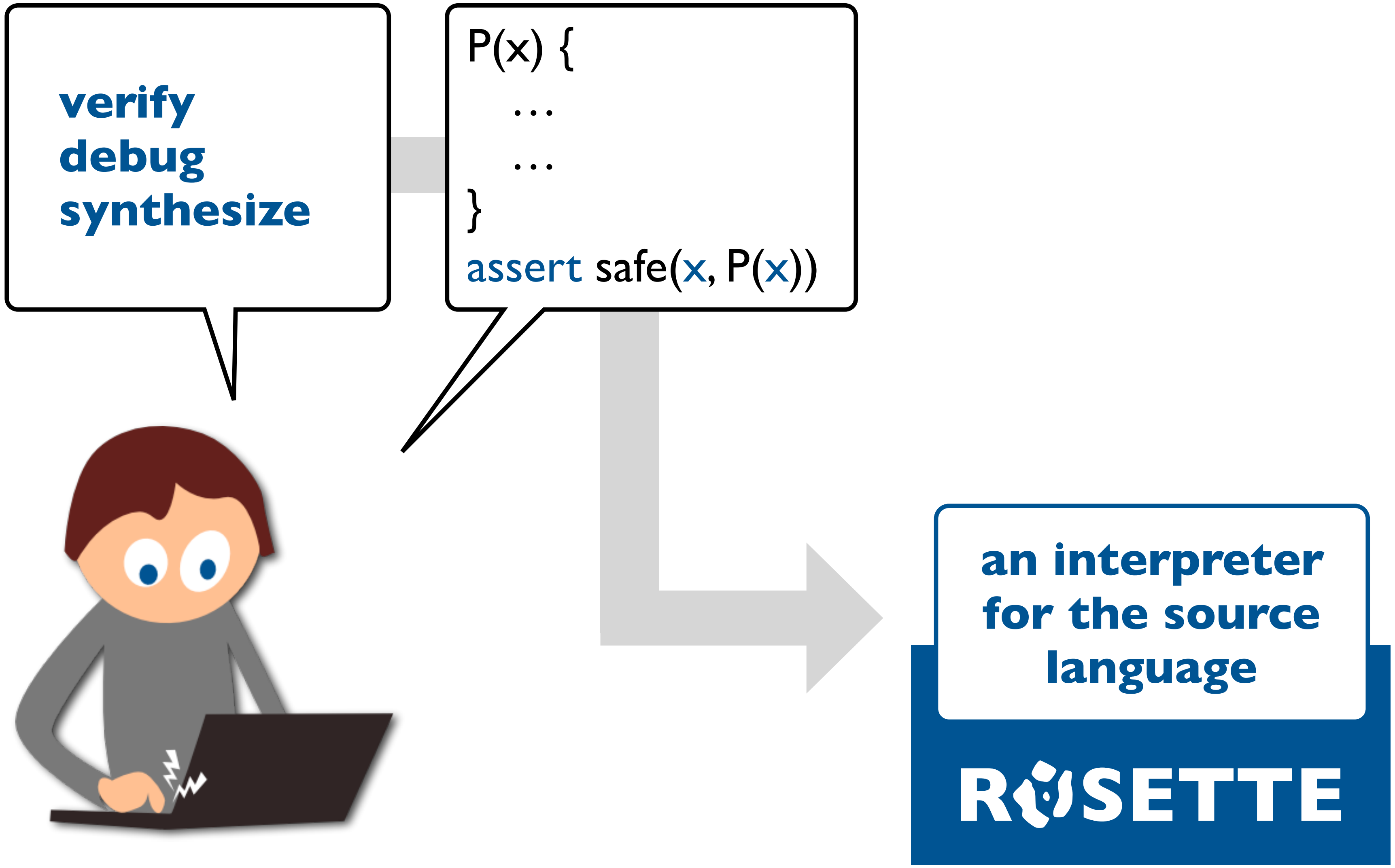
verify
debug
synthesize

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

programming

an interpreter for the source language

A newer, easier way to build tools



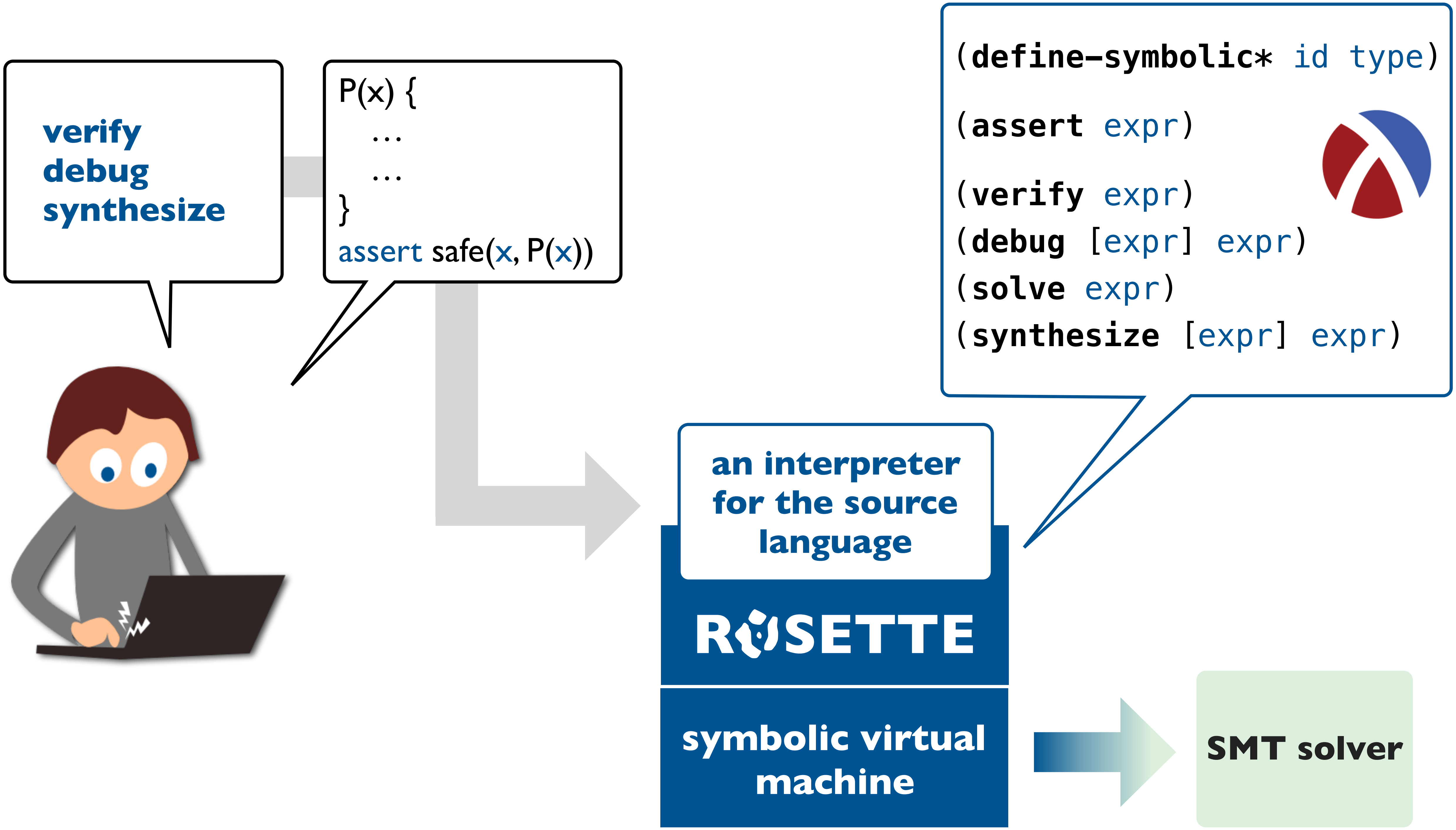
verify
debug
synthesize

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

an interpreter for the source language

ROSETTE

A newer, easier way to build tools



verify
debug
synthesize

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

```
(define-symbolic* id type)

(assert expr)

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

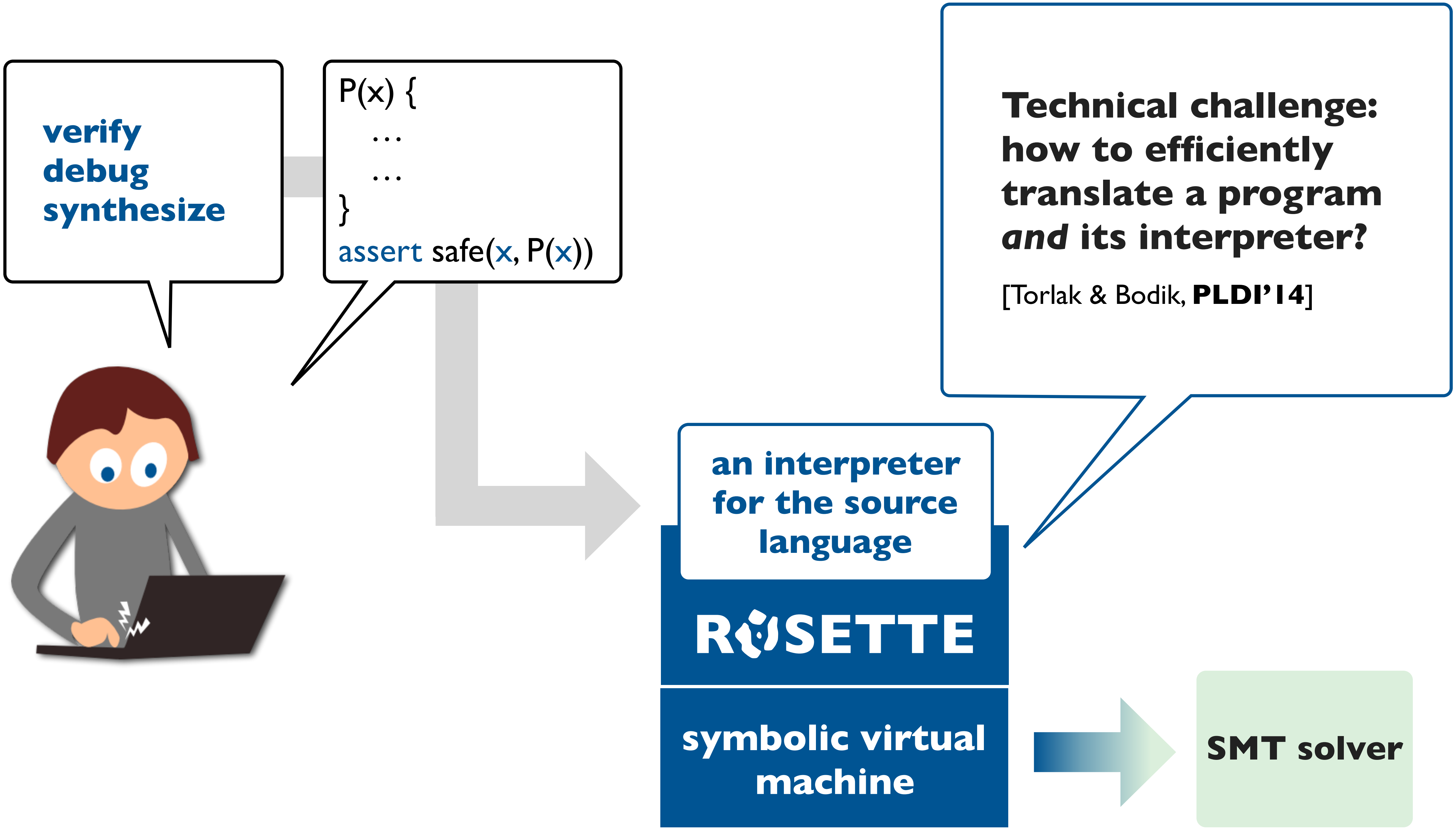
an interpreter for the source language

ROSETTE

symbolic virtual machine

SMT solver

A newer, easier way to build tools



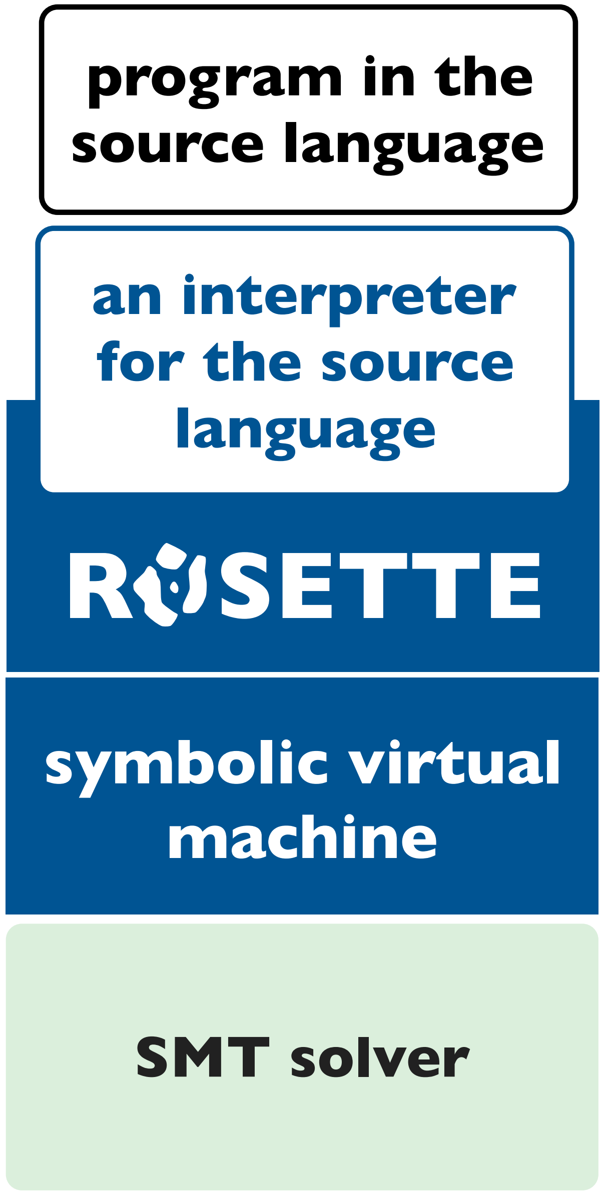
verify
debug
synthesize

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

Technical challenge: how to efficiently translate a program *and* its interpreter?

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

Many tools by a wide range of programmers …



program in the source language

an interpreter for the source language

ROSETTE

symbolic virtual machine

SMT solver

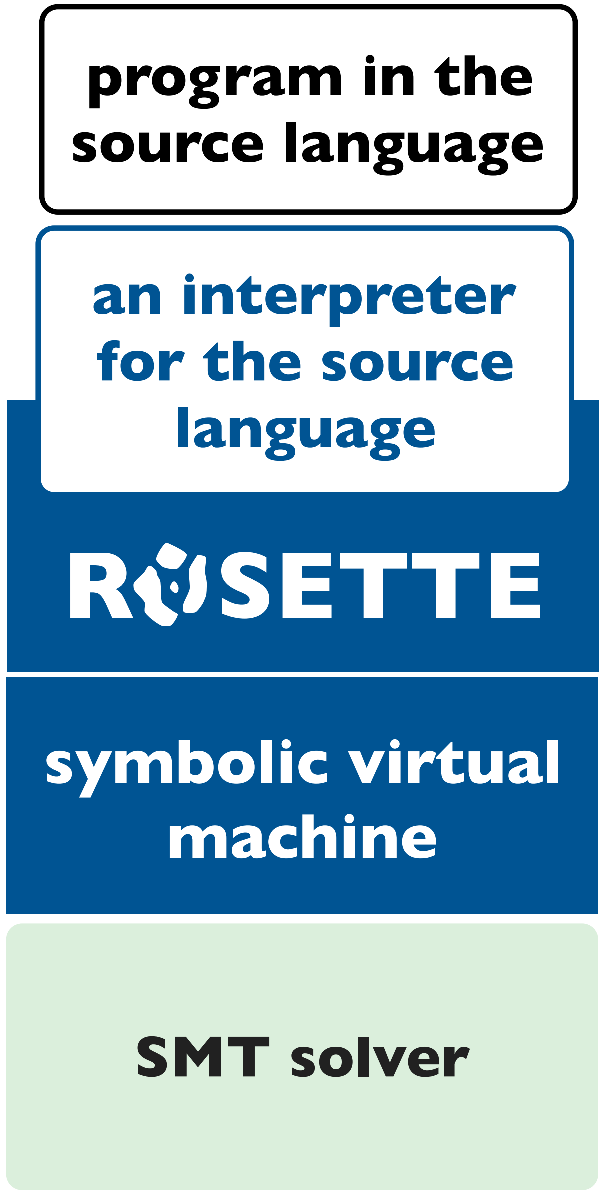
Used by two startup companies

- Enlearn (education)
- LinkiTools (compiler backends)

Applied to many practical problems

- Type system soundness bugs (POPL’18)
- Memory model synthesis (PLDI’17)
- MOOC feedback generation (ITiCSE'17)
- Radiation therapy (CAV’16)
- BGP configuration (OOPSLA’16)
- VMCAI’18 x 2, OOPSLA’17, ICFP’17, FDG’17, CIDR’17, ASPLOS’16 x 2, POPL’16, PLDI’14, …

Many tools by a wide range of programmers …



program in the source language

an interpreter for the source language

ROSETTE

symbolic virtual machine

SMT solver

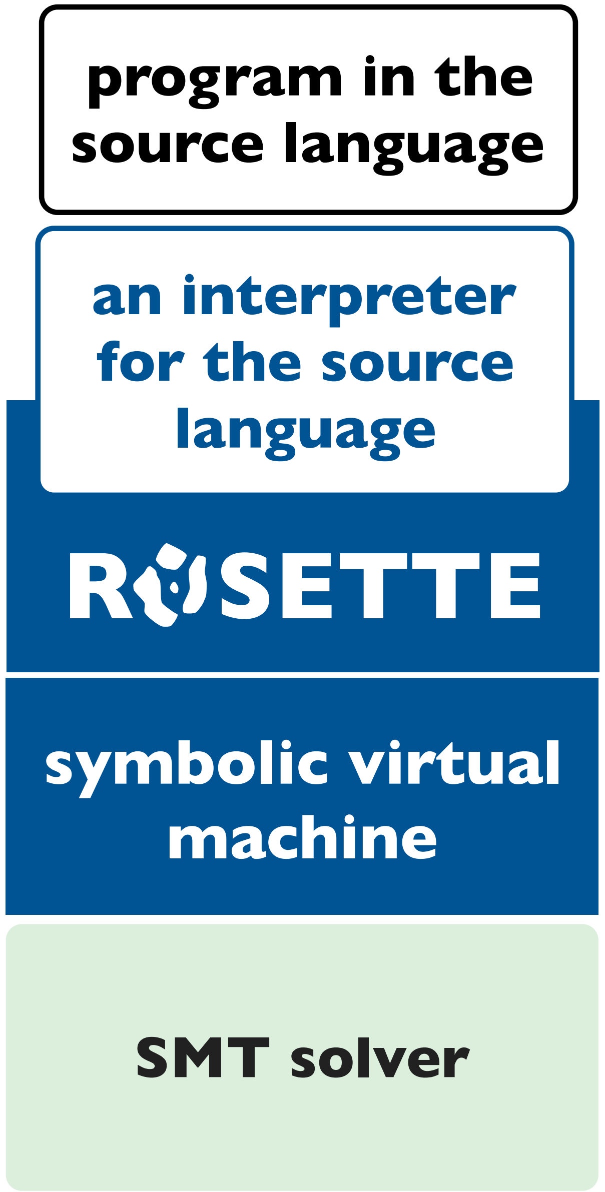
Used by two startup companies

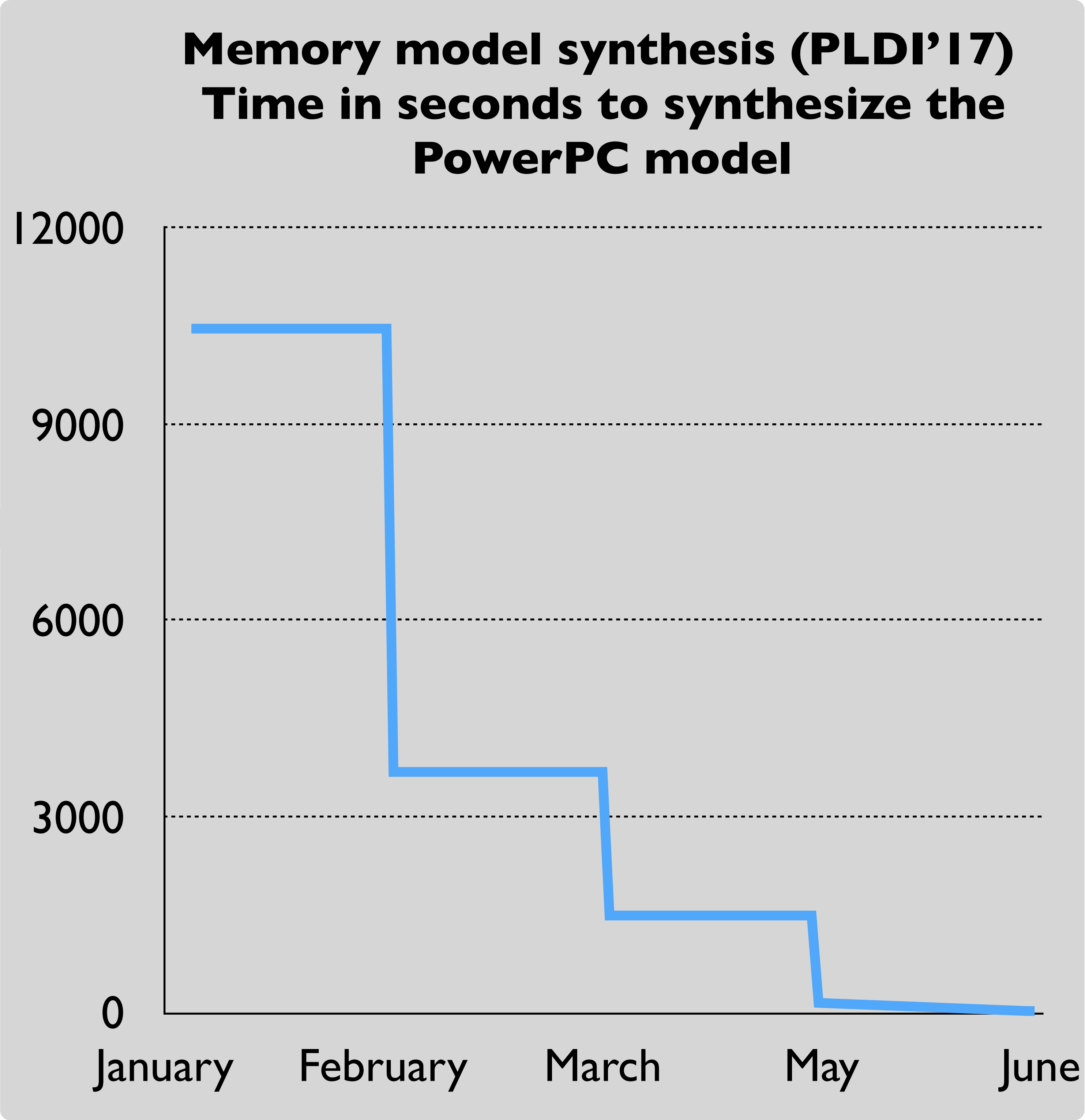
- Enlearn (education)
- LinkiTools (compiler backends)

Applied to many practical problems

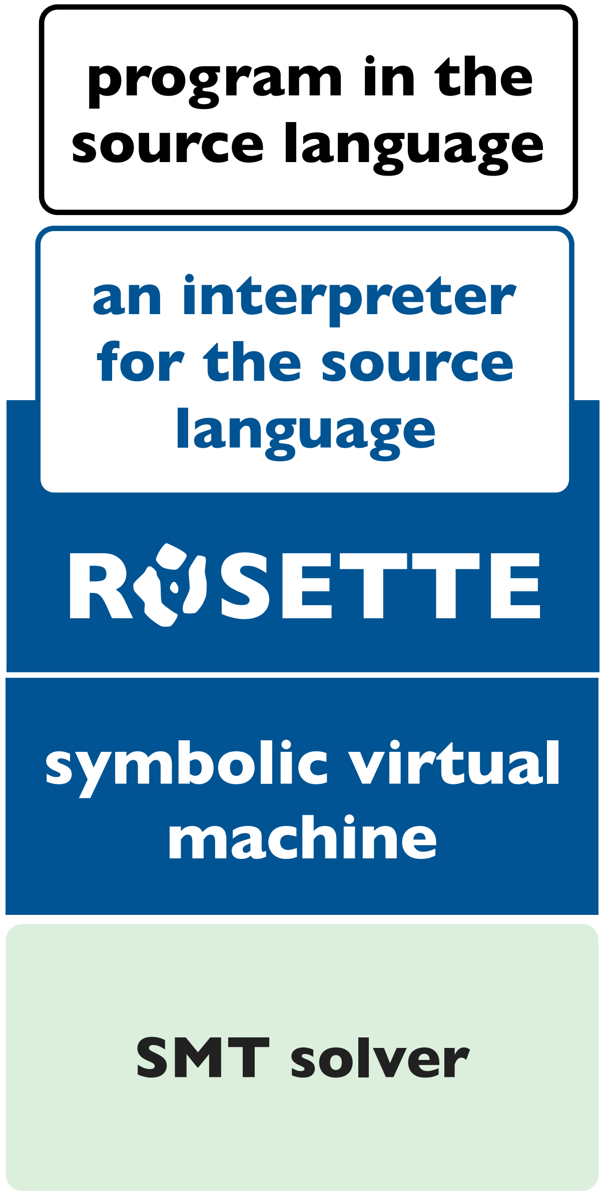
- Type system soundness bugs (POPL'18)
- **Memory model synthesis (PLDI'17)**
- MOOC feedback generation (ITiCSE'17)
- Radiation therapy (CAV'16)
- BGP configuration (OOPSLA'16)
- VMCAI'18 x 2, OOPSLA'17, ICFP'17, FDG'17, CIDR'17, ASPLOS'16 x 2, POPL'16, PLDI'14, …

But performance requires careful programming


program in the source language
an interpreter for the source language
ROSETTE
symbolic virtual machine
SMT solver



But performance requires careful programming



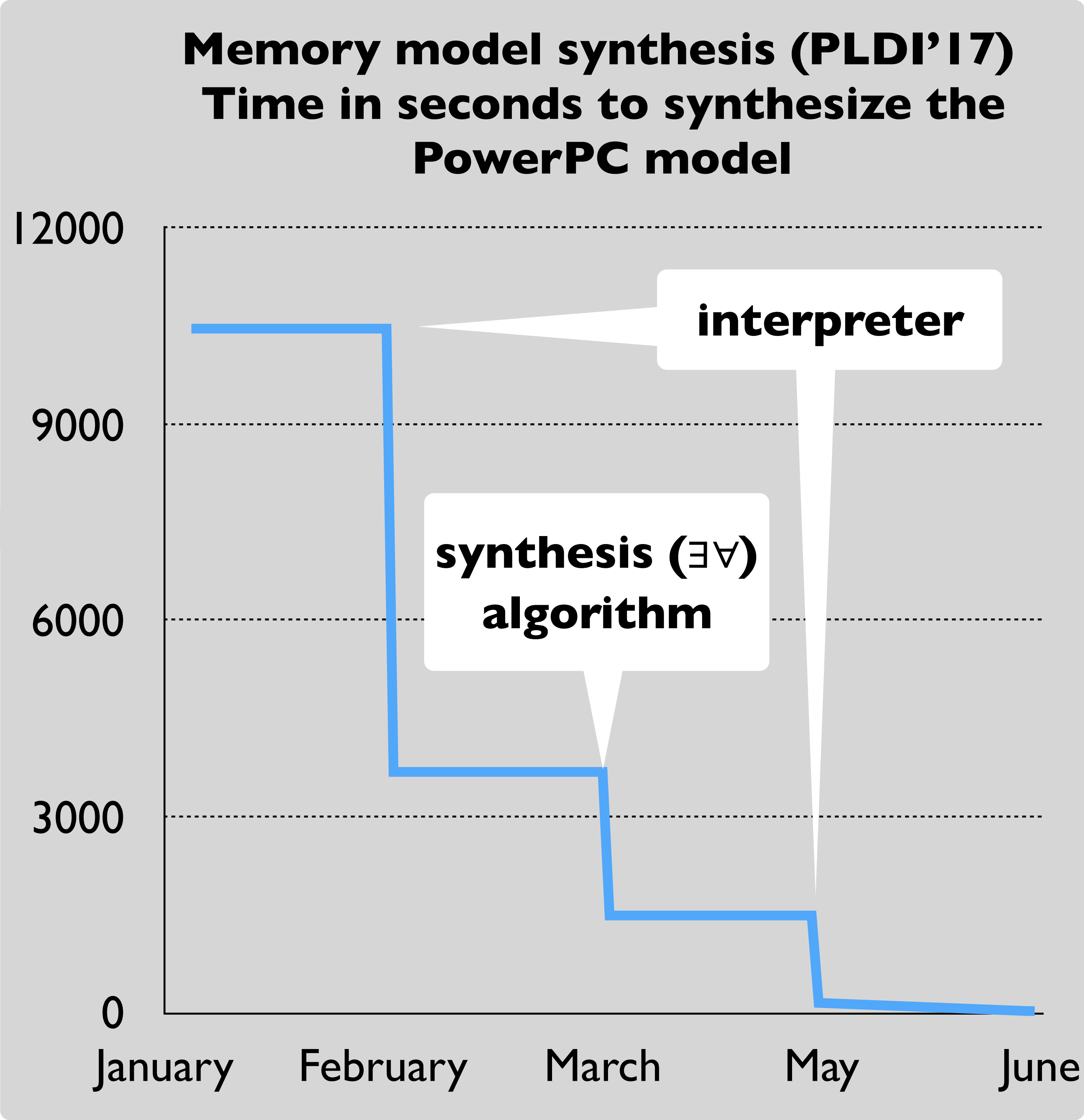
program in the source language

an interpreter for the source language

ROSETTE

symbolic virtual machine

SMT solver



Memory model synthesis (PLDI'17)
Time in seconds to synthesize the PowerPC model

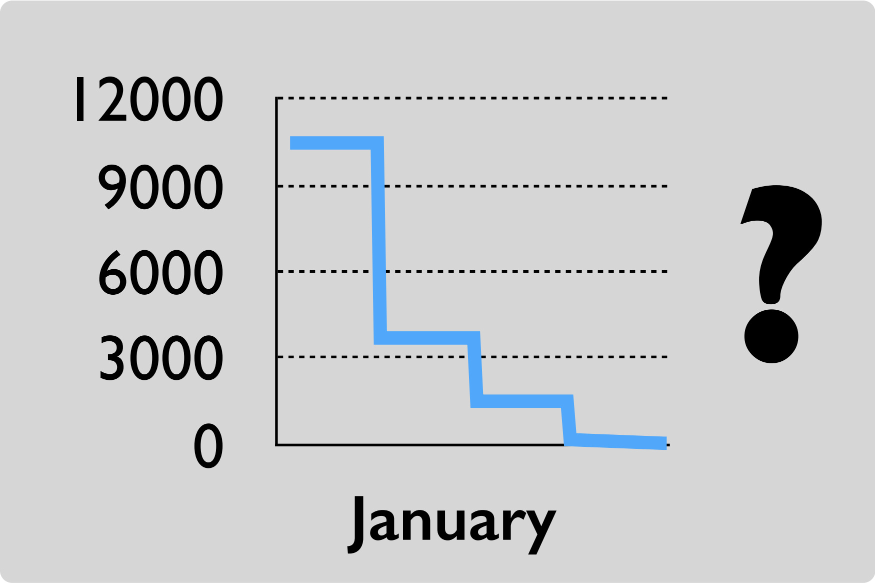
symbolic
profiling

Approach

Evaluation

A new technique for helping programmers **build performant solver-aided tools**

Problem



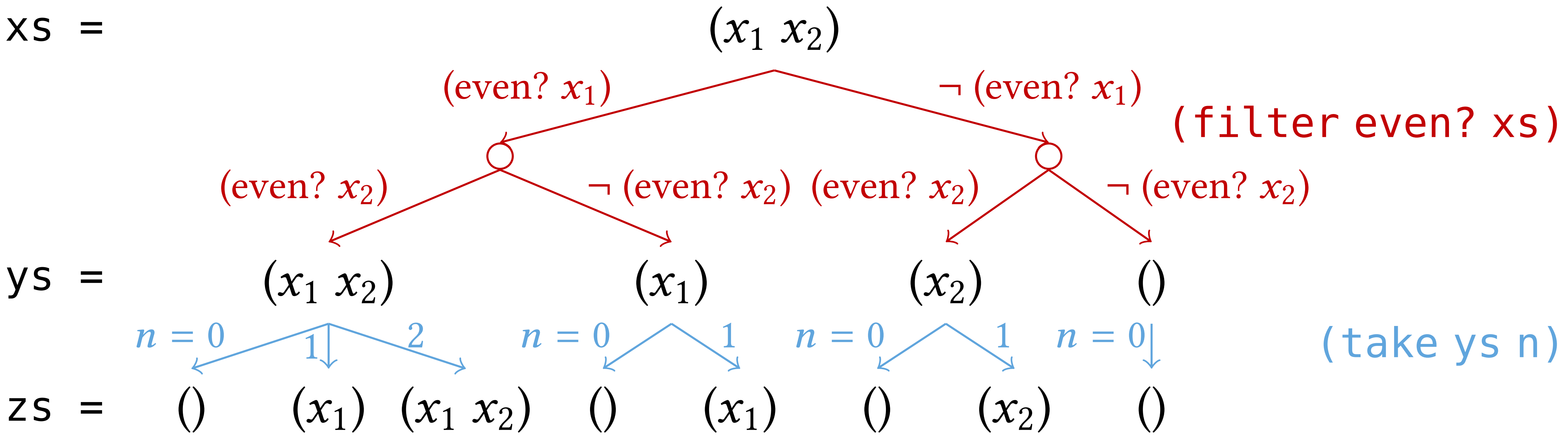
Background

A toy solver-aided program with a bottleneck

```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N]) ; xs is a list of N symbolic integers.
  (define-symbolic* n integer?)       ; n is a single symbolic integer.
  (define ys (filter even? xs))       ; ys contains the even integers from xs.
  (define zs (take ys n))             ; zs contains the first n elements of ys.
  (assert (even? (apply + zs))))      ; Check that the sum of zs is even.
```

A toy solver-aided program with a bottleneck

```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define ys (filter even? xs))
  (define zs (take ys n))
  (assert (even? (apply + zs))))
```



xs = $(x_1\ x_2)$

(even? x_1) ¬ (even? x_1)

(filter even? xs)

(even? x_2) ¬ (even? x_2) (even? x_2) ¬ (even? x_2)

ys = $(x_1\ x_2)$ (x_1) (x_2) $()$

$n = 0$ 1 2 $n = 0$ 1 $n = 0$ 1 $n = 0$

(take ys n)

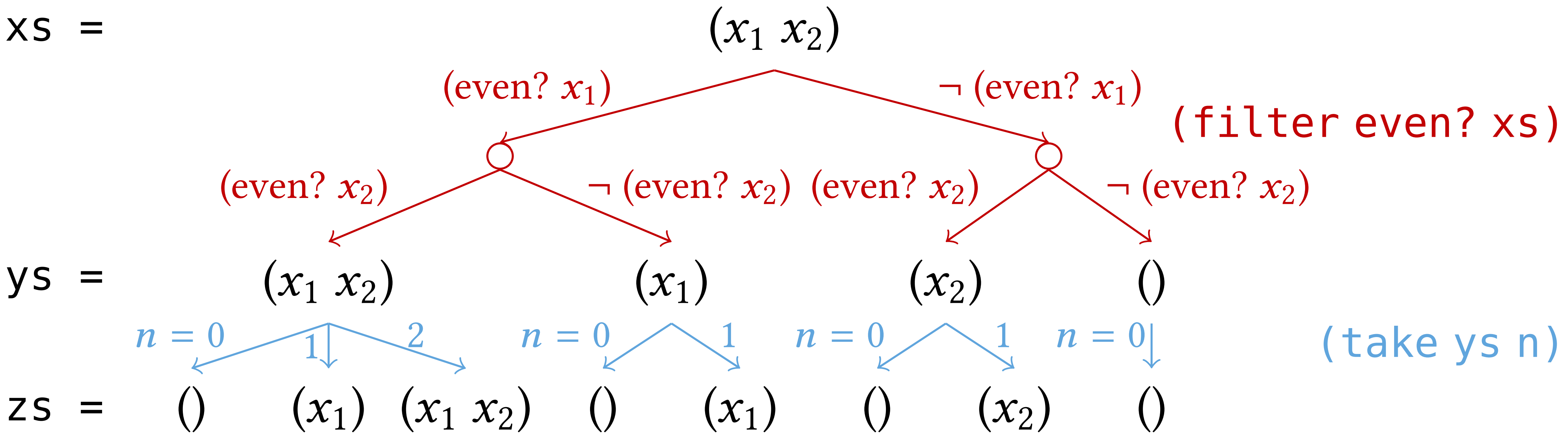
zs = $()$ (x_1) $(x_1\ x_2)$ $()$ (x_1) $()$ (x_2) $()$

symbolic execution for N = 2

A toy solver-aided program with a bottleneck

```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define ys (filter even? xs))
  (define zs (take ys n))
  (assert (even? (apply + zs))))
```

filter is the root cause of the bottleneck but a standard time-based profiler blames **take**.

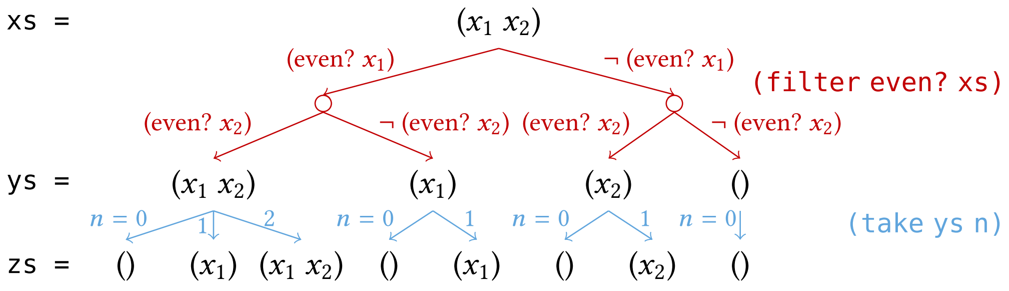


symbolic execution for N = 2

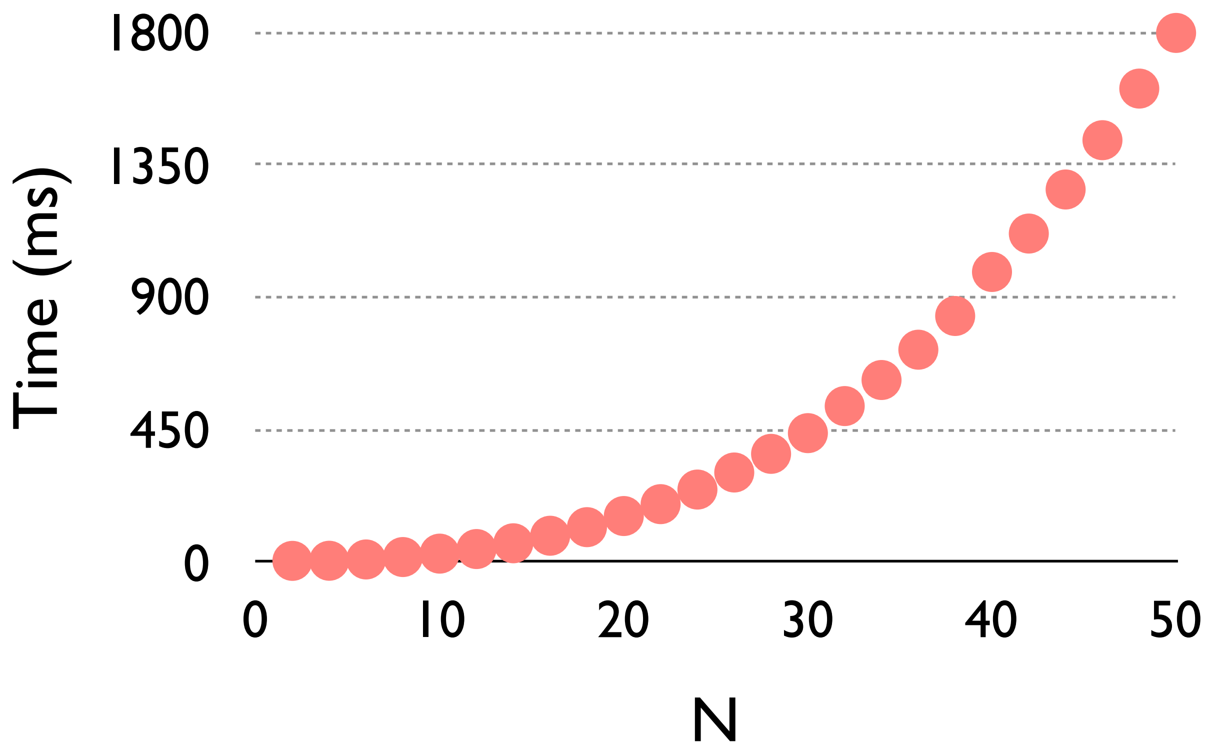
A toy solver-aided program with a bottleneck

```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define ys (filter even? xs))
  (define zs (take ys n))
  (assert (even? (apply + zs))))
```

filter is the root cause of the bottleneck but a standard time-based profiler blames **take**.



symbolic execution for N = 2



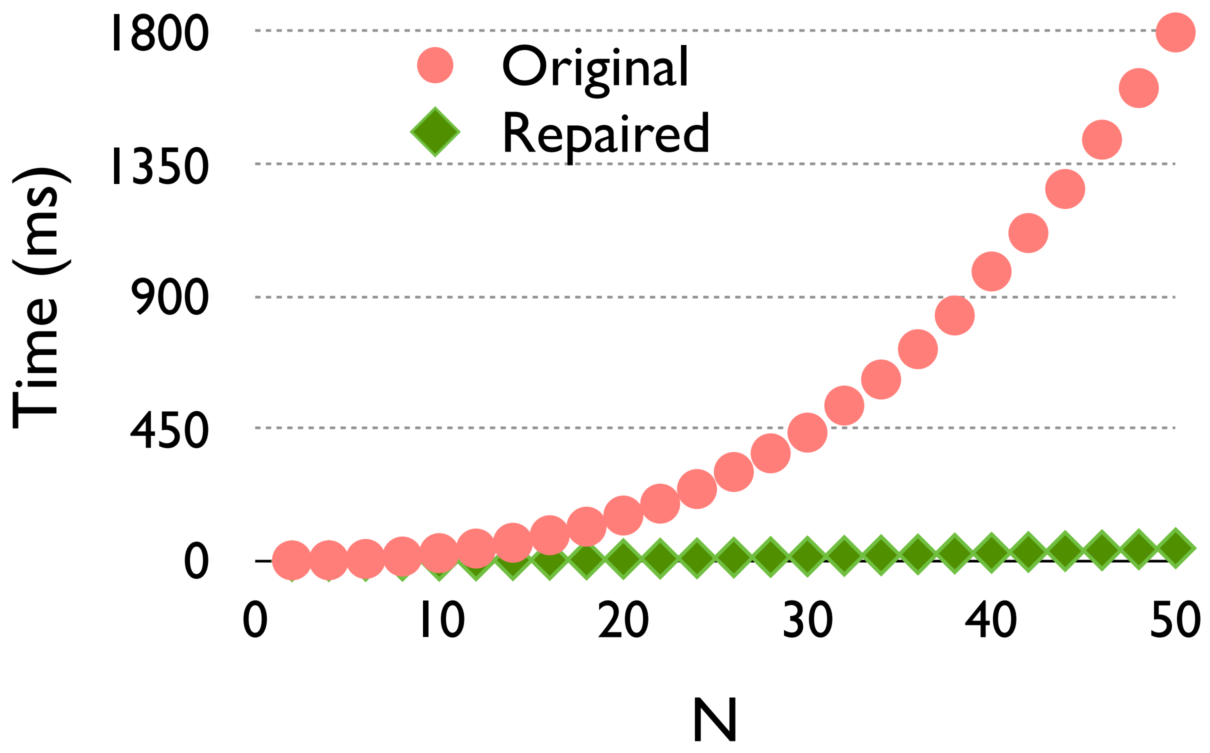
rosette execution for N <= 50

Repairing the toy program

```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define ys (filter even? xs))
  (define zs (take ys n))
  (assert (even? (apply + zs))))
```

Programming anti-pattern: *irregular representation*

```
(define (sum-of-evens-is-even* N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define zs (take xs n))
  (when (andmap even? zs)
    (assert (even? (apply + zs)))))
```



rosette execution for N <= 50

What do we need from a symbolic profiler?

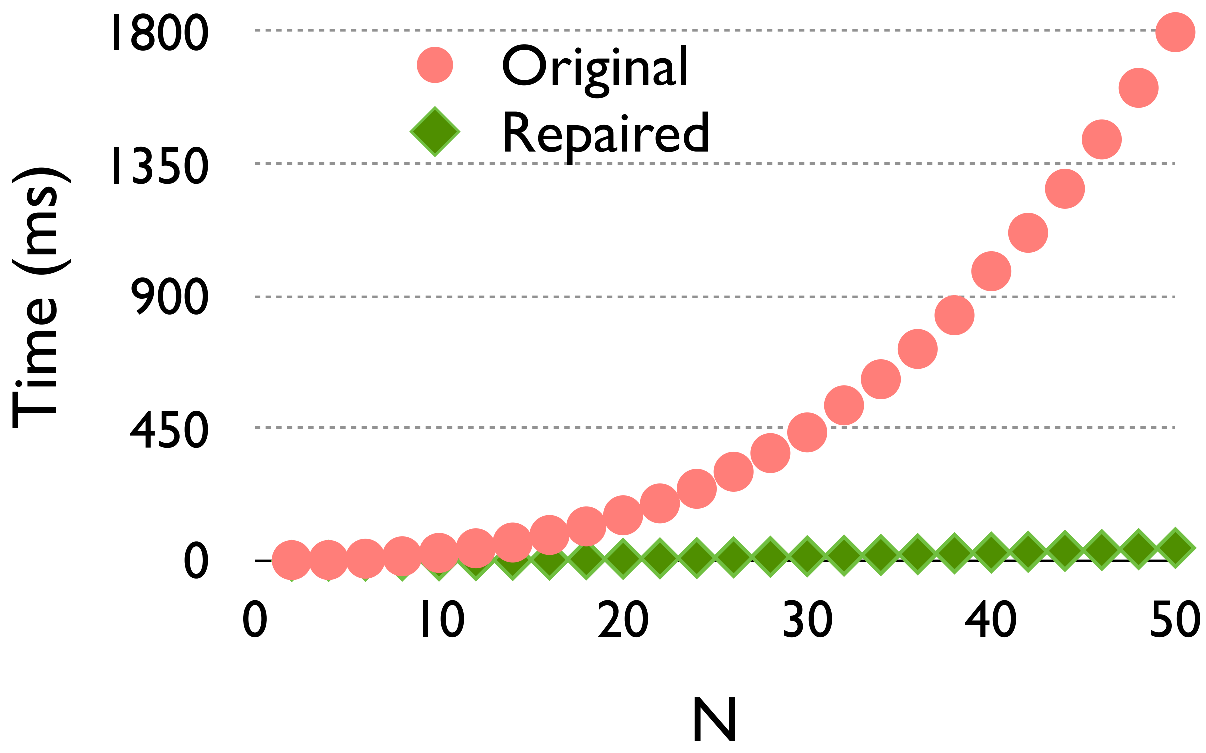
```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define ys (filter even? xs))
  (define zs (take ys n))
  (assert (even? (apply + zs))))
```

```
(define (sum-of-evens-is-even* N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define zs (take xs n))
  (when (andmap even? zs)
    (assert (even? (apply + zs)))))
```

Actionable: identifies root causes of performance bottlenecks.

Explainable: provides an abstract framework for understanding symbolic evaluation, without having to understand the implementation details.

General: applies to all forms of symbolic evaluation (from SE to BMC).



rosette execution for N <= 50

What do we need from a symbolic profiler?

```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define ys (filter even? xs))
  (define zs (take ys n))
  (assert (even? (apply + zs))))
```

```
(define (sum-of-evens-is-even* N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define zs (take xs n))
  (when (andmap even? zs)
    (assert (even? (apply + zs)))))
```

Actionable: identifies root causes of performance bottlenecks.

Explainable: provides an abstract framework for understanding symbolic evaluation, without having to understand the implementation details.

General: applies to all forms of symbolic evaluation (from SE to BMC).

Tried many ideas that didn’t work

- Time- and memory-based profiling
- Input-sensitive profiling (PLDI’02)
- Path-based profiling inspired by heuristics in SE engines (PLDI’12)
- Transformation-based profiling inspired by causal profiling (SOSP’15)

Actionable

Explainable

General

Approach

symbolic
profiling

Evaluation

A *new technique* for helping programmers build performant solver-aided tools

Problem

Background

Key idea: a uniform view of symbolic evaluation

The behavior of every (symbolic) evaluator can be understood in terms of two abstract data structures.

Symbolic heap: a DAG of all symbolic values (constants, terms) created during the symbolic evaluation of a program.

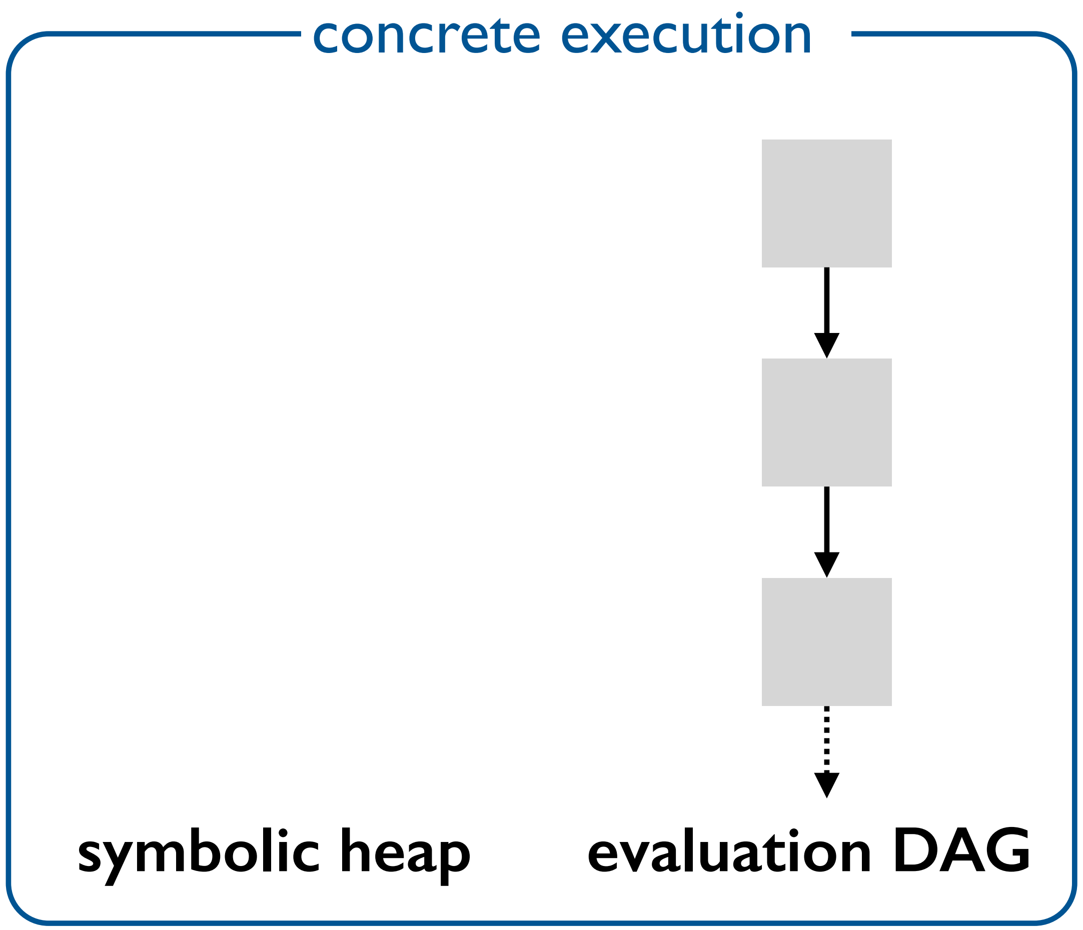
Symbolic evaluation graph: a DAG over program states that reflects the symbolic evaluation strategy.

Key idea: a uniform view of symbolic evaluation

The behavior of every (symbolic) evaluator can be understood in terms of two abstract data structures.

Symbolic heap: a DAG of all symbolic values (constants, terms) created during the symbolic evaluation of a program.

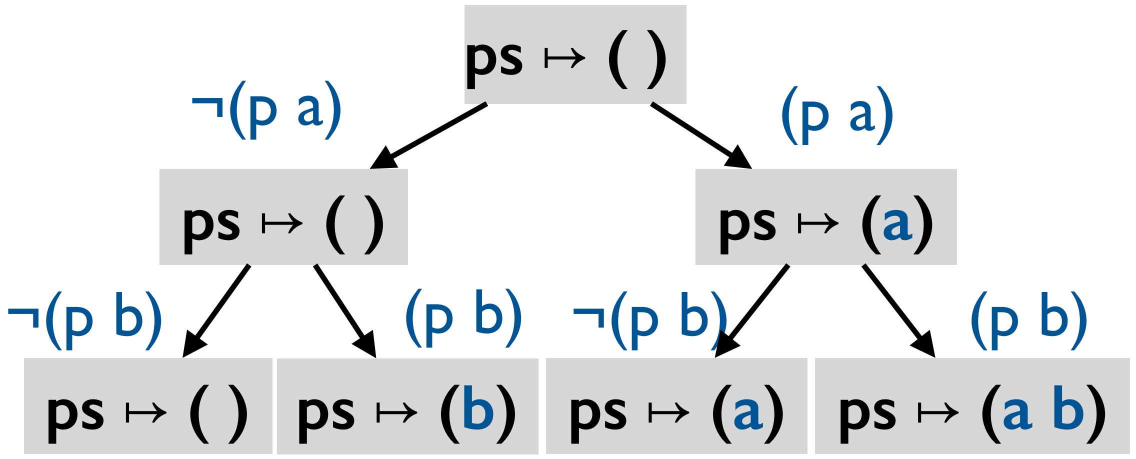
Symbolic evaluation graph: a DAG over program states that reflects the symbolic evaluation strategy.



Concrete evaluation is a special case of symbolic evaluation where the symbolic heap is empty and the evaluation graph is a single path.

Key idea: a uniform view of symbolic evaluation

symbolic execution of (filter p (a b))



ps ↦ ( )

¬(p a)

(p a)

ps ↦ ( )

ps ↦ (a)

¬(p b)

(p b)

¬(p b)

(p b)

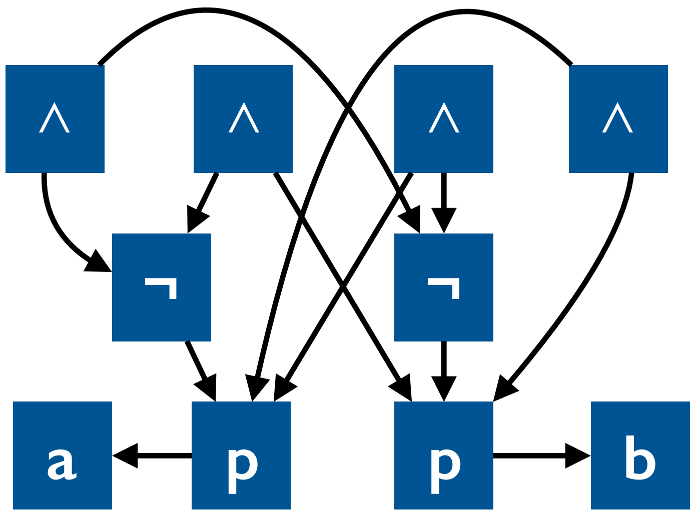
ps ↦ ( )

ps ↦ (b)

ps ↦ (a)

ps ↦ (a b)

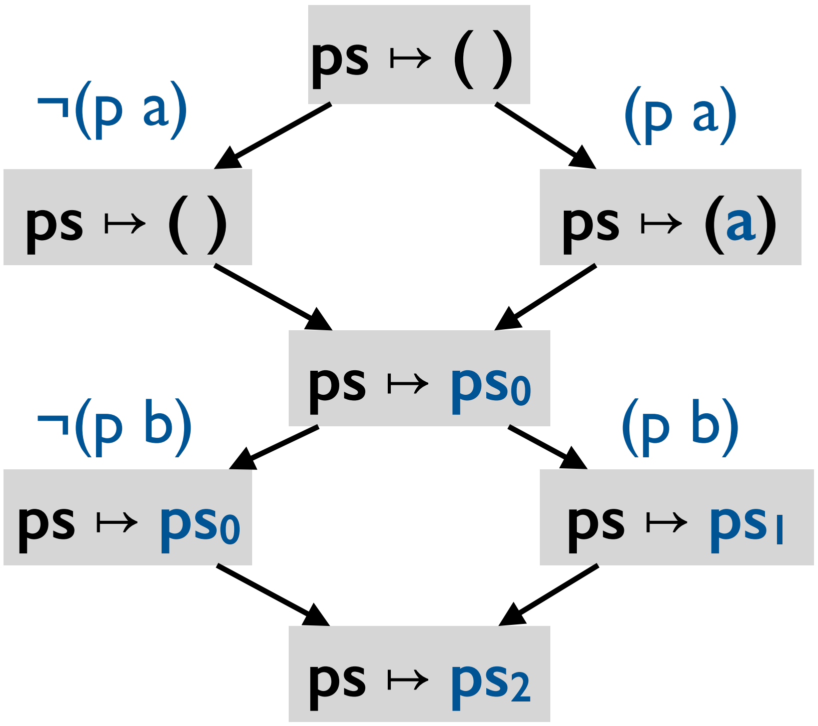
evaluation DAG



symbolic heap

bounded model checking of (filter p (a b))

evaluation DAG



ps ↦ ( )

¬(p a)

(p a)

ps ↦ ( )

ps ↦ (a)

ps ↦ ps$_0$

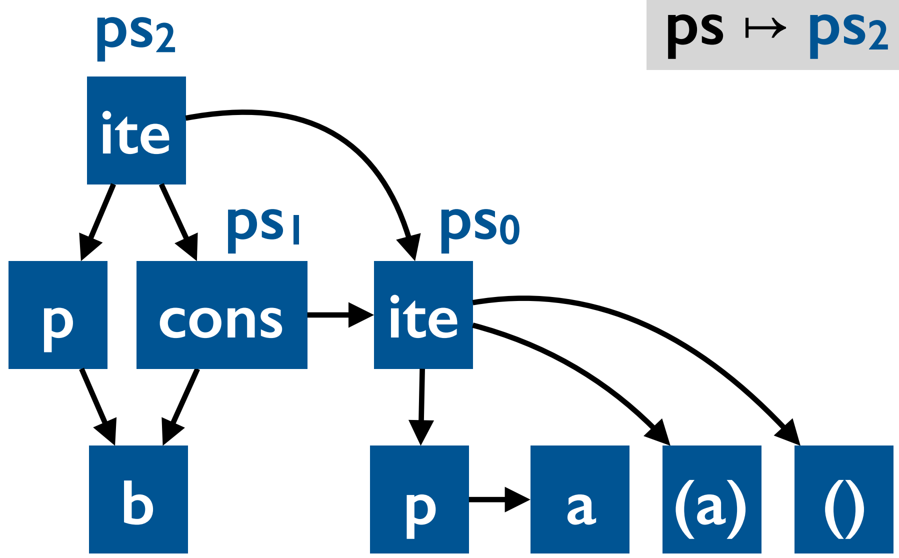
¬(p b)

(p b)

ps ↦ ps$_0$

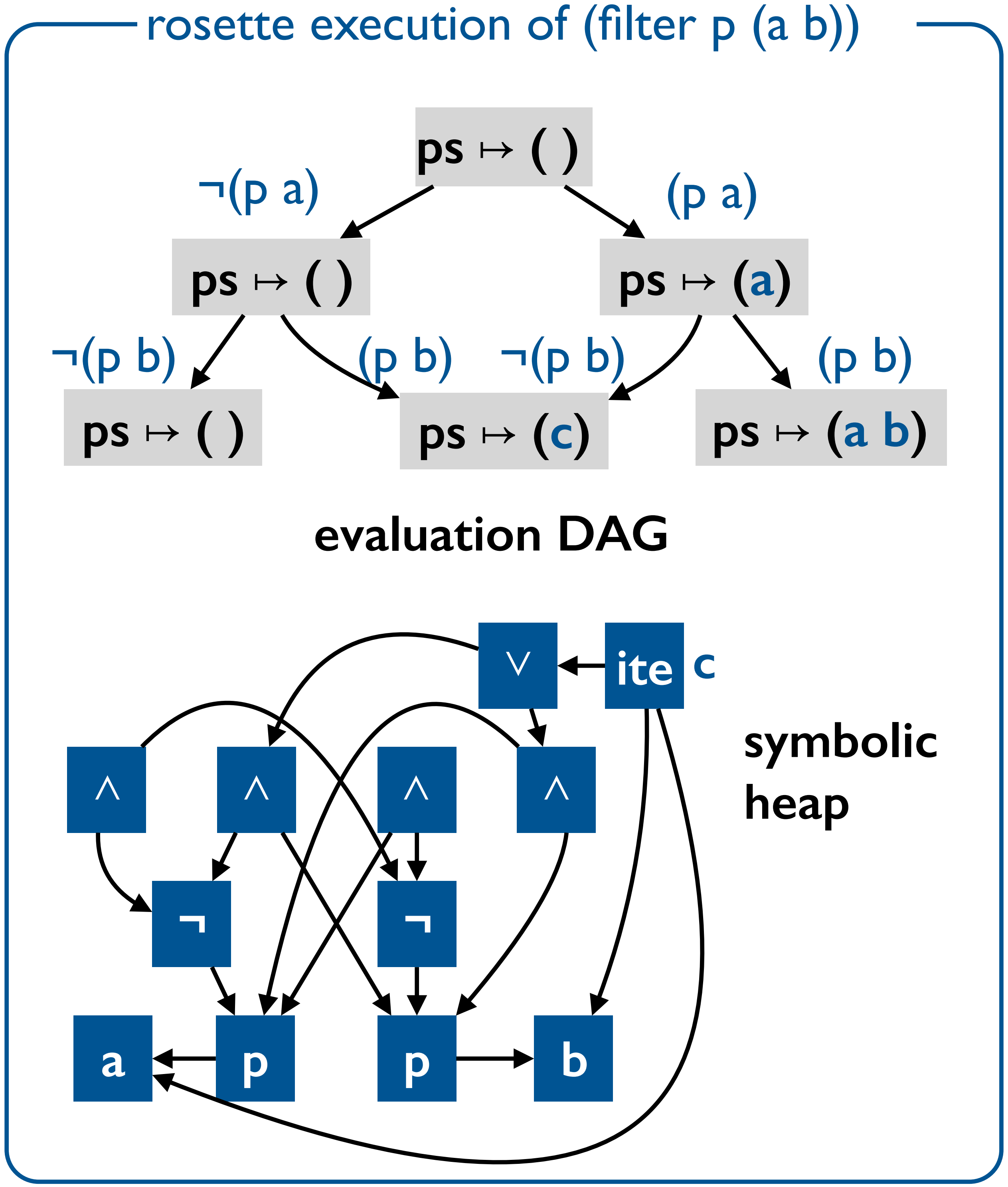
ps ↦ ps$_1$

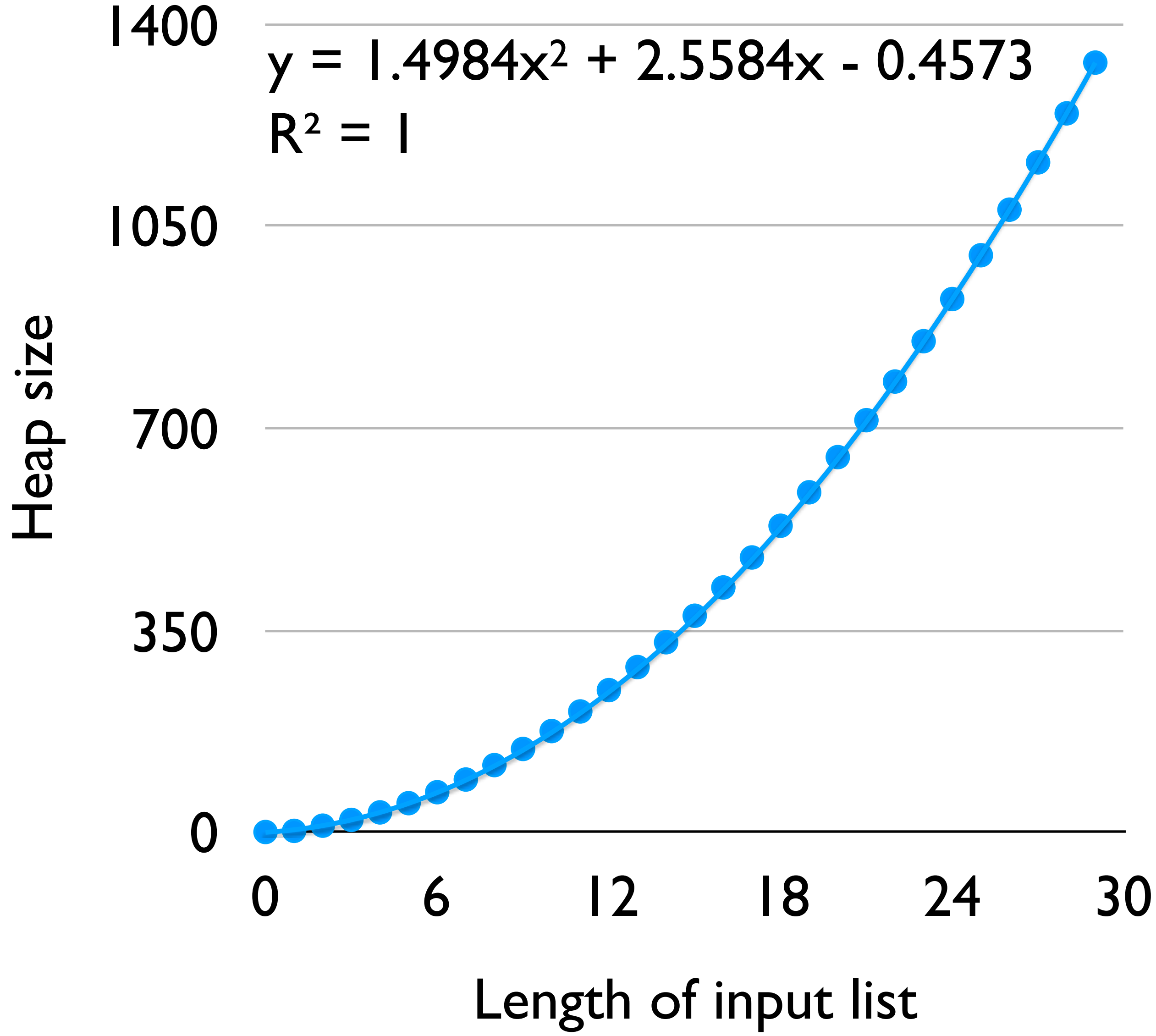
ps ↦ ps$_2$



symbolic heap

Key idea: a uniform view of symbolic evaluation


rosette execution of (filter p (a b))



Approach: expose a symbolic profiling interface

new(x, L): create a fresh symbolic constant x at program location L.

new(op, x…, L): create a symbolic term (op x …) at location L.

step(s_0, <g_1, e_1>, …, <g_n, e_n>): evaluate each e_i starting from state s_0 under guard g_i and return the resulting k >= n states

merge(<g_1, s_1>, …, <g_n, s_n>, L): merge the given states at the program location L and return the resulting k <= states

solve(x, L): call the solver at location L to determine the satisfiability of x.

Can be cheaply implemented in all symbolic evaluators.

Approach: collect per-call summary statistics

new(x, L): create a fresh symbolic constant x at program location L.

new(op, x…, L): create a symbolic term (op x …) at location L.

step(s_0, <g_1, e_1>, …, <g_n, e_n>): evaluate each e_i starting from state s_0 under guard g_i and return the resulting k >= n states

merge(<g_1, s_1>, …, <g_n, s_n>, L): merge the given states at the program location L and return the resulting k <= states

solve(x, L): call the solver at location L to determine the satisfiability of x.

Time **is the exclusive wall-clock time spent in the call.**

Term count **is the number of symbolic values added to the symbolic heap.**

Unused terms **is the number of those values not sent to the solver.**

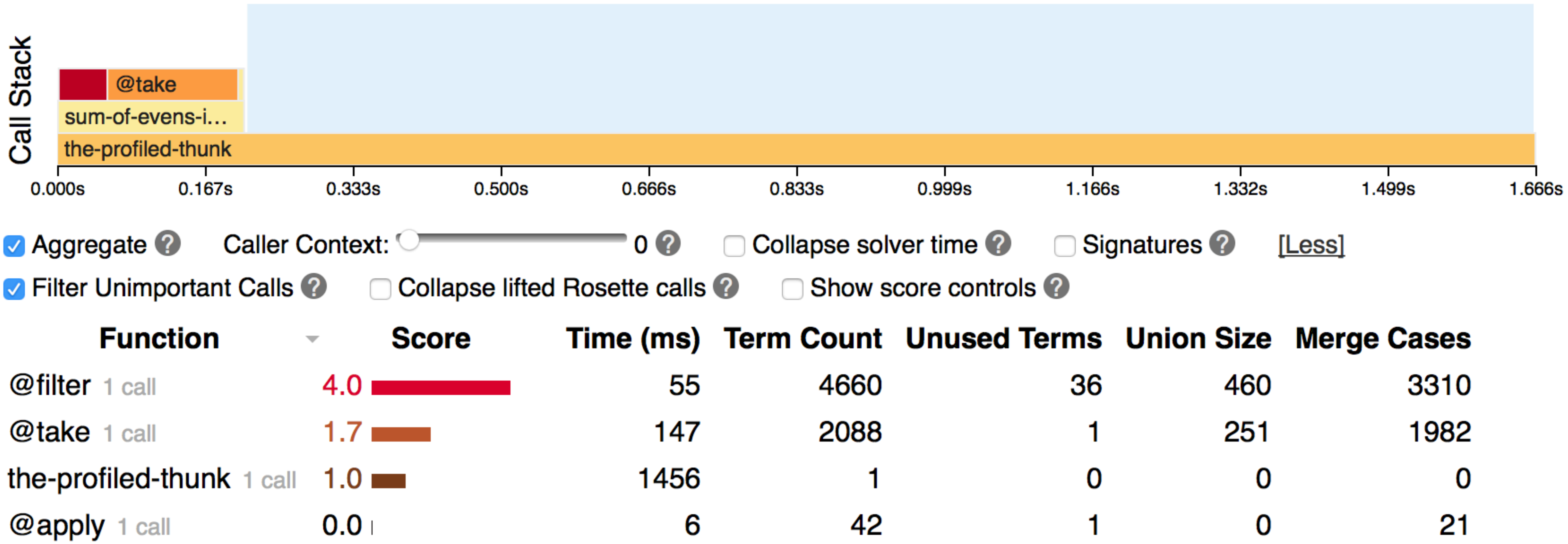
Union size **is the sum of the out-degrees of all nodes added to the evaluation graph.**

Merge cases **is the sum of the in-degrees of all nodes added to the evaluation graph.**

Approach: rank calls based on the statistics

```
(define (sum-of-evens-is-even N)
  (define-symbolic* xs integer? [N])
  (define-symbolic* n integer?)
  (define ys (filter even? xs))
  (define zs (take ys n))
  (assert (even? (apply + zs))))
```

```
(verify (sum-of-evens-is-even 20))
```



☑ Aggregate ? Caller Context: 0 ? □ Collapse solver time ? □ Signatures ?

☑ Filter Unimportant Calls ? □ Collapse lifted Rosette calls ? □ Show score controls ?

Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
@filter 1 call	4.0	55	4660	36	460	3310
@take 1 call	1.7	147	2088	1	251	1982
the-profiled-thunk 1 call	1.0	1456	1	0	0	0
@apply 1 call	0.0	6	42	1	0	21

symbolic profiling

Approach

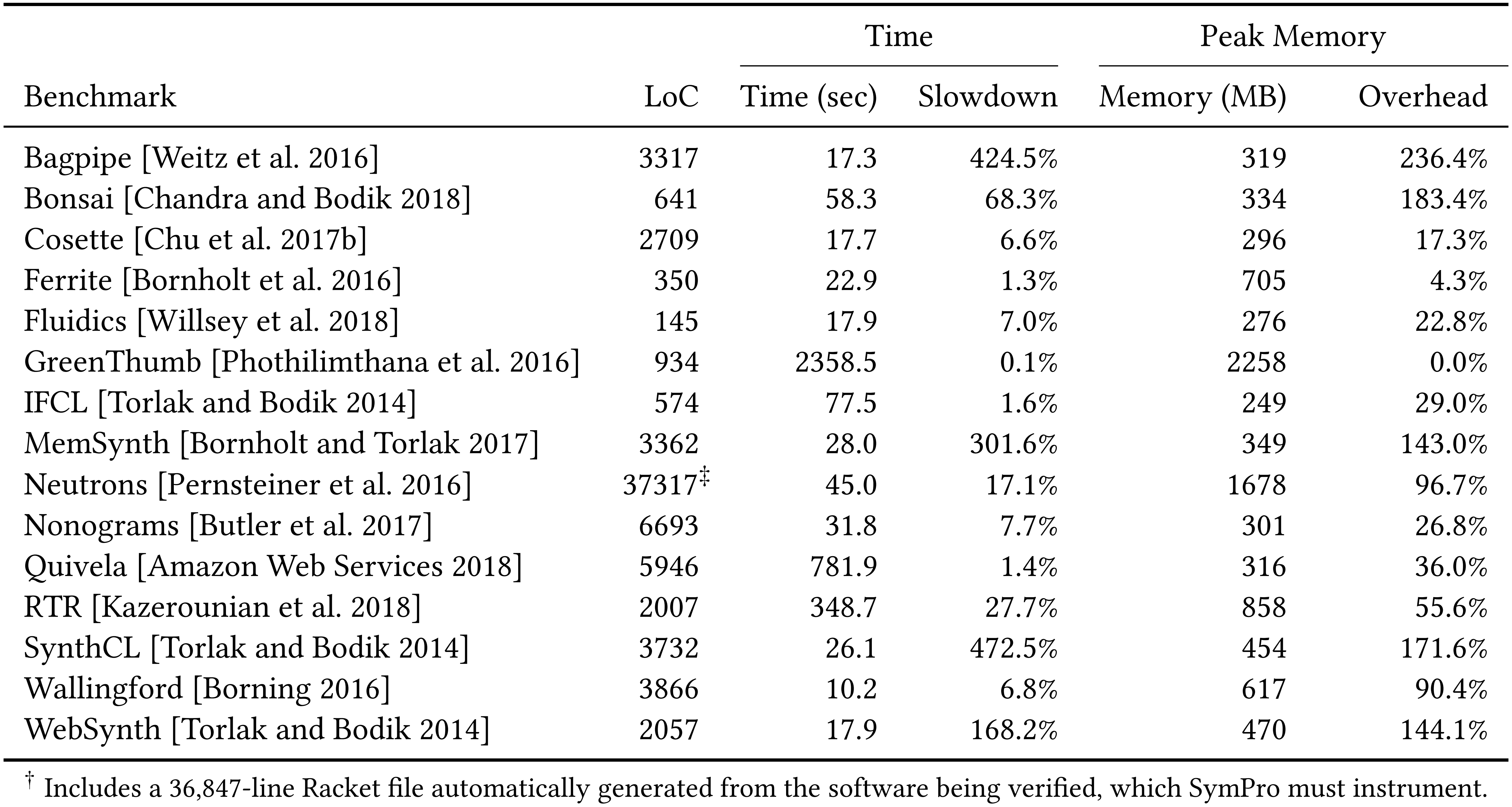
Evaluation

A new technique for **helping programmers** build performant solver-aided tools

Problem

Background

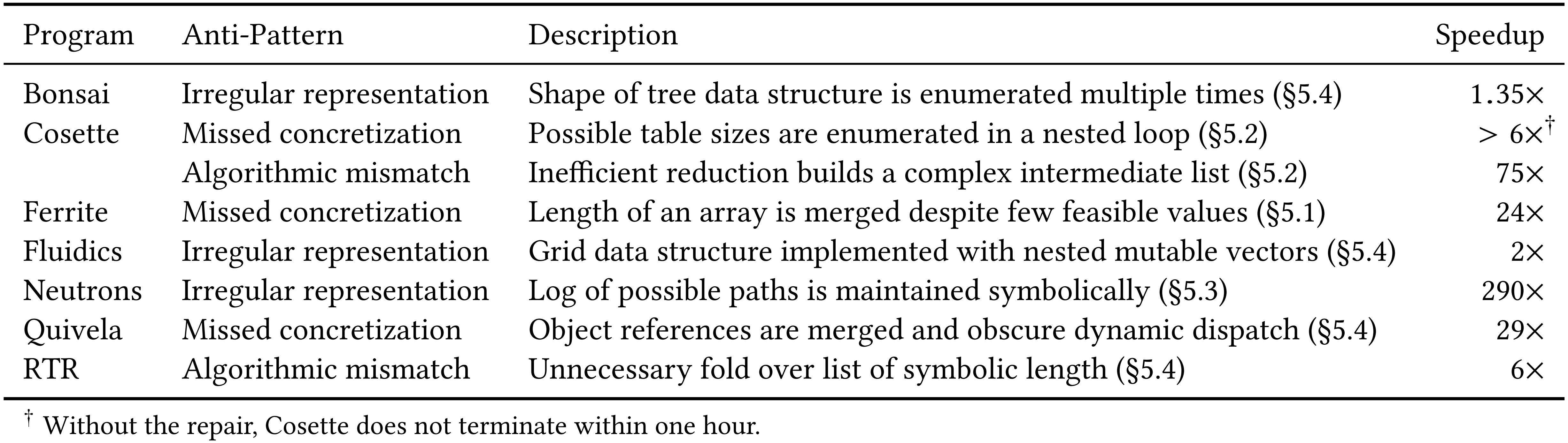
Actionable for state-of-the-art tools



		Time		Peak Memory	
Benchmark	LoC	Time (sec)	Slowdown	Memory (MB)	Overhead
Bagpipe [Weitz et al. 2016]	3317	17.3	424.5%	319	236.4%
Bonsai [Chandra and Bodik 2018]	641	58.3	68.3%	334	183.4%
Cosette [Chu et al. 2017b]	2709	17.7	6.6%	296	17.3%
Ferrite [Bornholt et al. 2016]	350	22.9	1.3%	705	4.3%
Fluidics [Willsey et al. 2018]	145	17.9	7.0%	276	22.8%
GreenThumb [Phothilimthana et al. 2016]	934	2358.5	0.1%	2258	0.0%
IFCL [Torlak and Bodik 2014]	574	77.5	1.6%	249	29.0%
MemSynth [Bornholt and Torlak 2017]	3362	28.0	301.6%	349	143.0%
Neutrons [Pernsteiner et al. 2016]	37317[‡]	45.0	17.1%	1678	96.7%
Nonograms [Butler et al. 2017]	6693	31.8	7.7%	301	26.8%
Quivela [Amazon Web Services 2018]	5946	781.9	1.4%	316	36.0%
RTR [Kazerounian et al. 2018]	2007	348.7	27.7%	858	55.6%
SynthCL [Torlak and Bodik 2014]	3732	26.1	472.5%	454	171.6%
Wallingford [Borning 2016]	3866	10.2	6.8%	617	90.4%
WebSynth [Torlak and Bodik 2014]	2057	17.9	168.2%	470	144.1%

[†] Includes a 36,847-line Racket file automatically generated from the software being verified, which SymPro must instrument.

Actionable for state-of-the-art tools



Program	Anti-Pattern	Description	Speedup
Bonsai	Irregular representation	Shape of tree data structure is enumerated multiple times (§5.4)	1.35×
Cosette	Missed concretization	Possible table sizes are enumerated in a nested loop (§5.2)	> 6×[†]
	Algorithmic mismatch	Inefficient reduction builds a complex intermediate list (§5.2)	75×
Ferrite	Missed concretization	Length of an array is merged despite few feasible values (§5.1)	24×
Fluidics	Irregular representation	Grid data structure implemented with nested mutable vectors (§5.4)	2×
Neutrons	Irregular representation	Log of possible paths is maintained symbolically (§5.3)	290×
Quivela	Missed concretization	Object references are merged and obscure dynamic dispatch (§5.4)	29×
RTR	Algorithmic mismatch	Unnecessary fold over list of symbolic length (§5.4)	6×

[†] Without the repair, Cosette does not terminate within one hour.

Found 8 performance bottlenecks, repaired to get 35% to 290x speedups, and 3 patches accepted by developers.

Generalizes to different symbolic evaluators

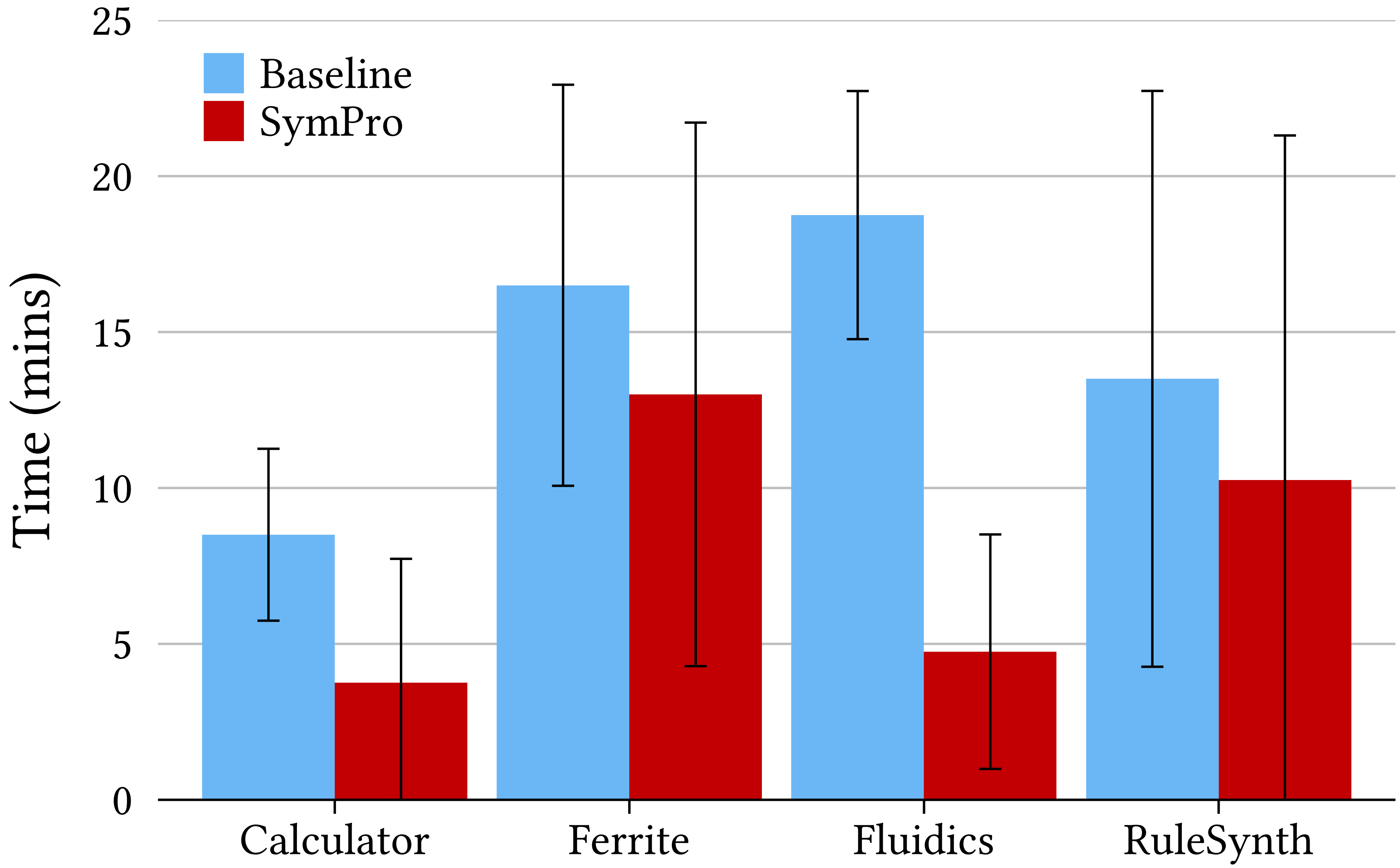
Rosette solver-aided language

Jalangi framework for JavaScript

Found 8 performance bottlenecks, repaired to get 35% to 290x speedups, and 3 patches accepted by developers.

Found 3 performance bottlenecks, repaired to get 10% to 2x speedups on largest benchmarks (though still small).

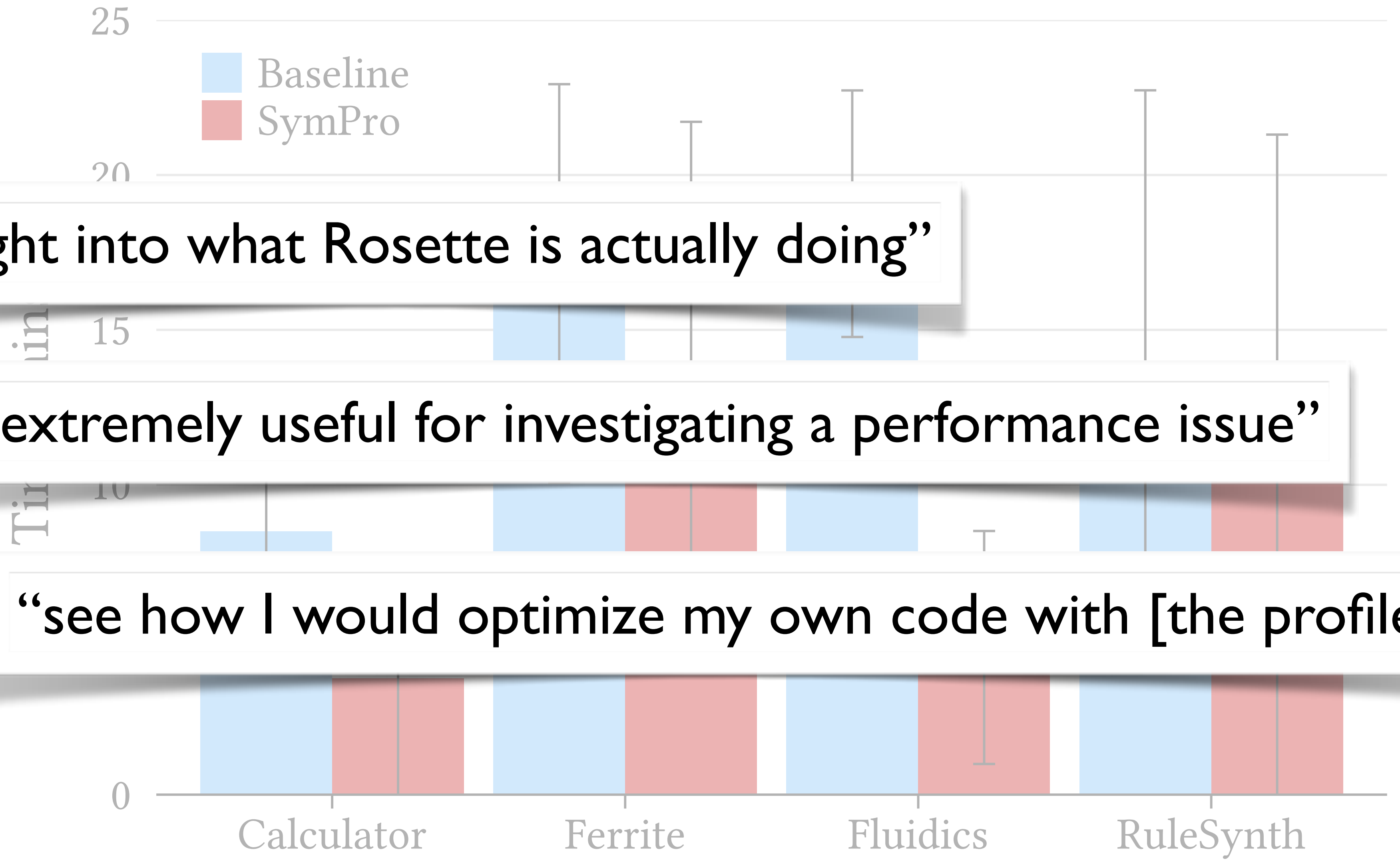
Explainable to programmers



Small user study with 8 Rosette programmers with a range of experience.

6 cases where programmers in the baseline group failed to find the issue within 20 minutes. No such cases with SymPro.

Explainable to programmers



“insight into what Rosette is actually doing”

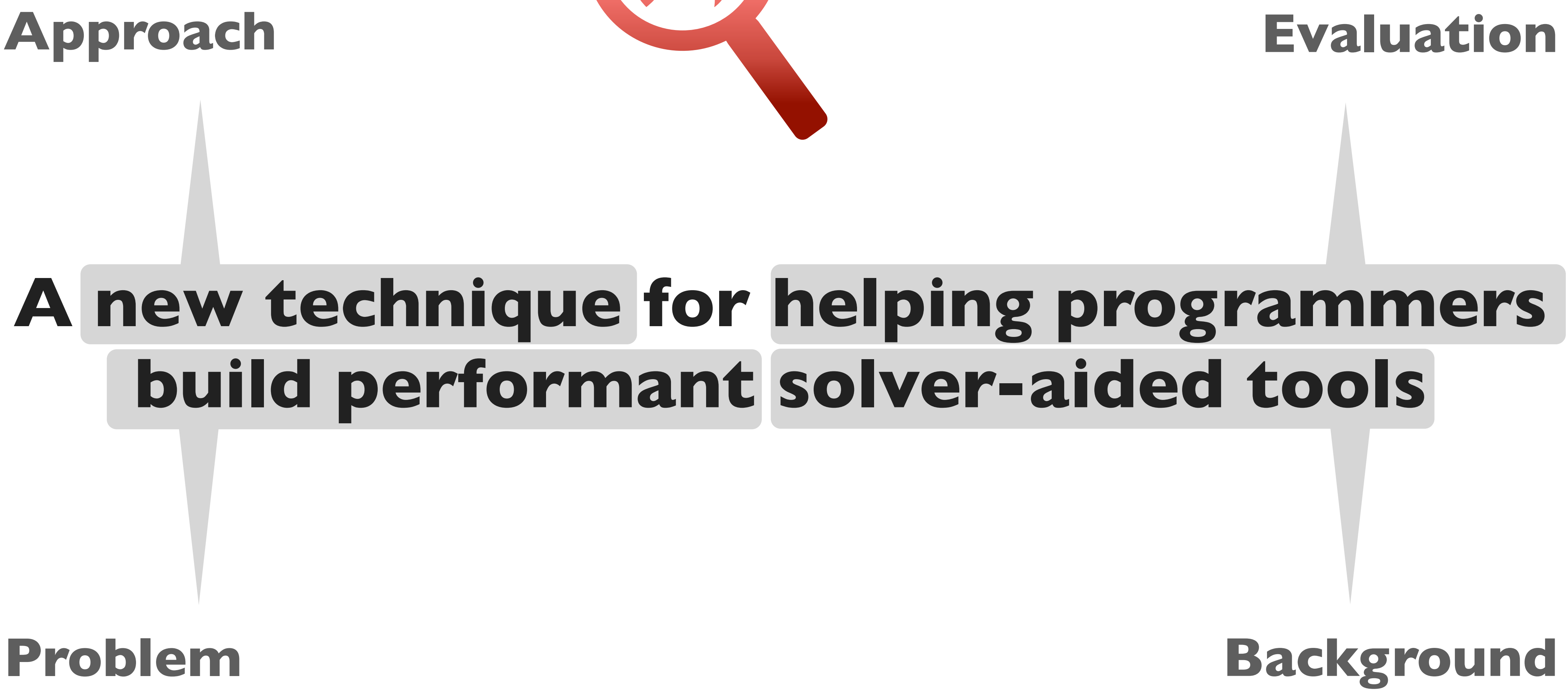
“extremely useful for investigating a performance issue”

“see how I would optimize my own code with [the profiler]”

Small user study with 8 Rosette programmers with a range of experience.

6 cases where programmers in the baseline group failed to find the issue within 20 minutes. No such cases with SymPro.

symbolic
profiling



Approach

Evaluation

A new technique for helping programmers build performant solver-aided tools

Problem

Background