Syntax-Guided Synthesis in an SMT Solver: Past and Future Applications

Andrew Reynolds

February 22, 2023

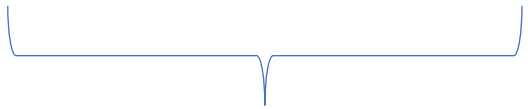
Outline

- Syntax-guided Synthesis (SyGuS) in an SMT Solver (cvc5)
  - Problem statement, how it works
- Past successful applications
  - Rewrite rule synthesis, test case generation, solving quantified T-constraints
- Future applications
  - Code optimization, abduction for ITP
  - Use of user-provided *oracles*

Syntax-Guided Synthesis

Synthesis Conjectures

$$\exists f.\forall x.P(f,x)$$



There exists a term `f` for which property `P` holds for all `x`

Synthesis Conjectures

$$\exists f.\forall x.P(f,x)$$

There exists a term `f` for which property `P` holds for all `x`

- `P` is typically expressed as a formula in an SMT theory
- `f` may correspond to:
  - A program-to-synthesize
  - A solved form for a constraint
  - A sufficient/necessary condition for a desired approximation
    - Interpolants, abducts
  - An interesting test input
  - …

Syntax-Guided Synthesis Conjectures Modulo T

$$\exists f.\forall x.P(f,x)$$

spec(f)

There exists a term `f` for which property `P` holds for all `x`

$$A \to A+A \mid -A \mid x \mid y \mid 0 \mid 1 \mid ite(B,A,A)$$
$$B \to B \wedge B \mid \neg B \mid A=A \mid A \geq A \mid \bot$$

syntax(f)

The body of `f` is generated by the above grammar with start symbol `A`

$\Rightarrow$ *Syntax-guided synthesis "SyGuS"* **[Alur et al 2013]**

Enumerative Cex-Guided Inductive Synthesis (CEGIS)

```
syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥
```

```
spec(f):
∀xy.f(x,y)=f(y,x)+f(0,1)
```

Solution *Enumerator*

Solution Verifier

[Solar-Lezama 2006, Alur et al 2013]

Enumerative Cex-Guided Inductive Synthesis

```
syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥
```

```
spec(f):
∀xy.f(x,y)=f(y,x)+f(0,1)
```


f:=λxy.x?

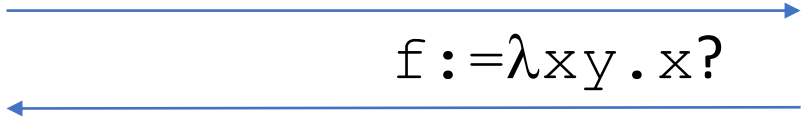
Solution
Enumerator

Solution
Verifier

Enumerative Cex-Guided Inductive Synthesis

```
syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥
```

```
spec(f):
∀xy.f(x,y)=f(y,x)+f(0,1)
```




```
f(0,1)=f(1,0)+f(0,1)
```

Solution *Enumerator*

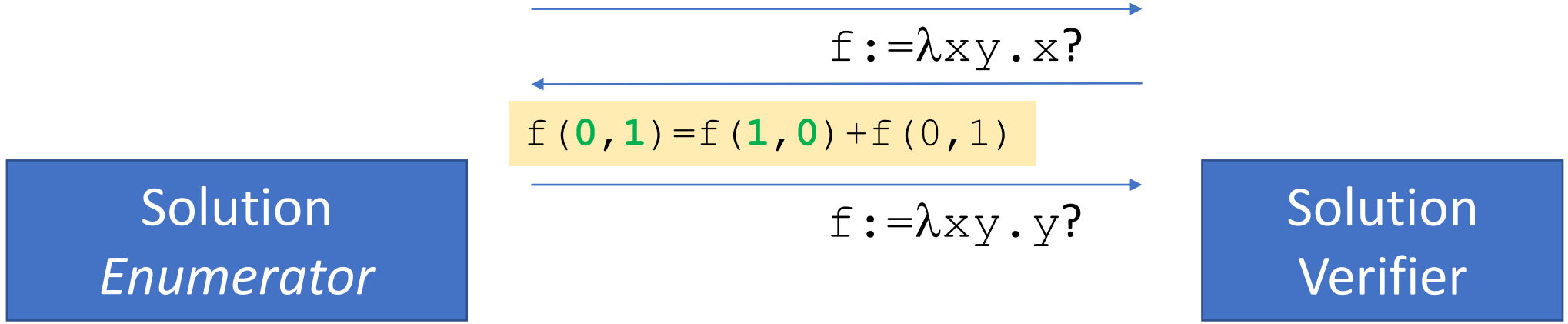
Solution Verifier

..counterexample: spec does not hold for **(x,y)=(0,1)**

Enumerative Cex-Guided Inductive Synthesis

```
syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥
```

```
spec(f):
∀xy.f(x,y)=f(y,x)+f(0,1)
```

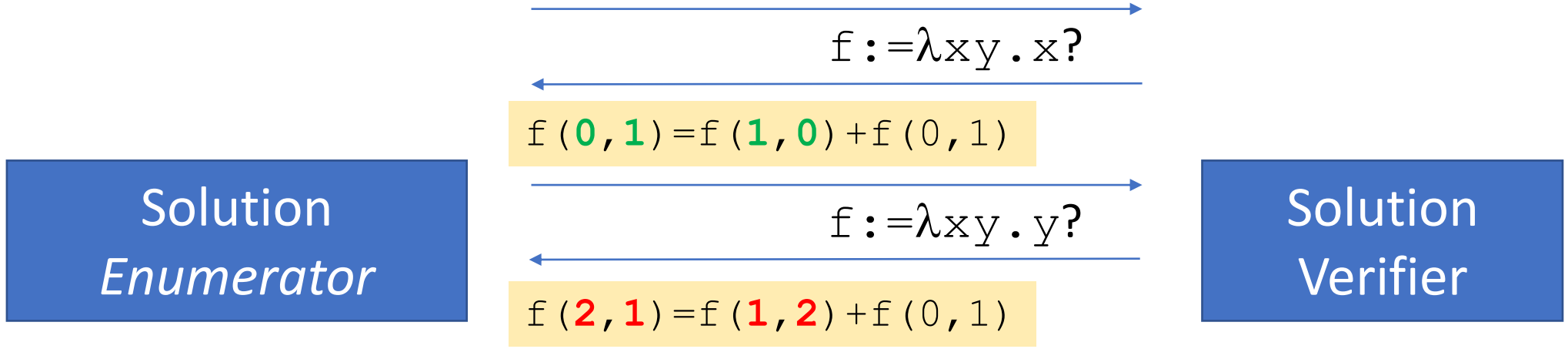


…new candidate λxy.y satisfies **(x,y)=(0,1)**

Enumerative Cex-Guided Inductive Synthesis

```
syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥
```

```
spec(f):
∀xy.f(x,y)=f(y,x)+f(0,1)
```

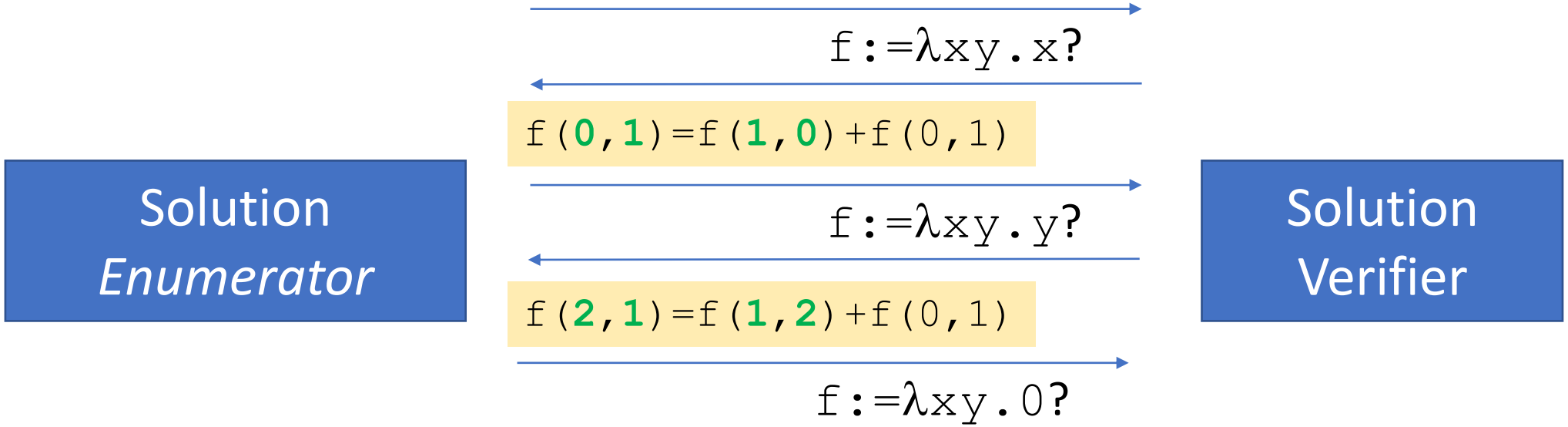


…counterexample: spec does not hold for **(x,y)=(2,1)**

Enumerative Cex-Guided Inductive Synthesis

```
syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥
```

```
spec(f):
∀xy.f(x,y)=f(y,x)+f(0,1)
```



…new candidate λxy.0 satisfies **(x,y)=(0,1),(2,1)**

Enumerative Cex-Guided Inductive Synthesis

syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥

spec(f):
∀xy.f(x,y)=f(y,x)+f(0,1)

Solution *Enumerator*

f:=λxy.x?

f(0,1)=f(1,0)+f(0,1)

f:=λxy.y?

f(2,1)=f(1,2)+f(0,1)

f:=λxy.0?

Solution Verifier

f:=λxy.0

…new candidate λxy.0 has no counterexamples wrt **spec(f)**

SyGuS using SMT solvers

```
syntax(f):
A->A+A|-A|x|y|0|1|ite(B,A,A)
B->B∧B|¬B|A=A|A≥A|⊥
```

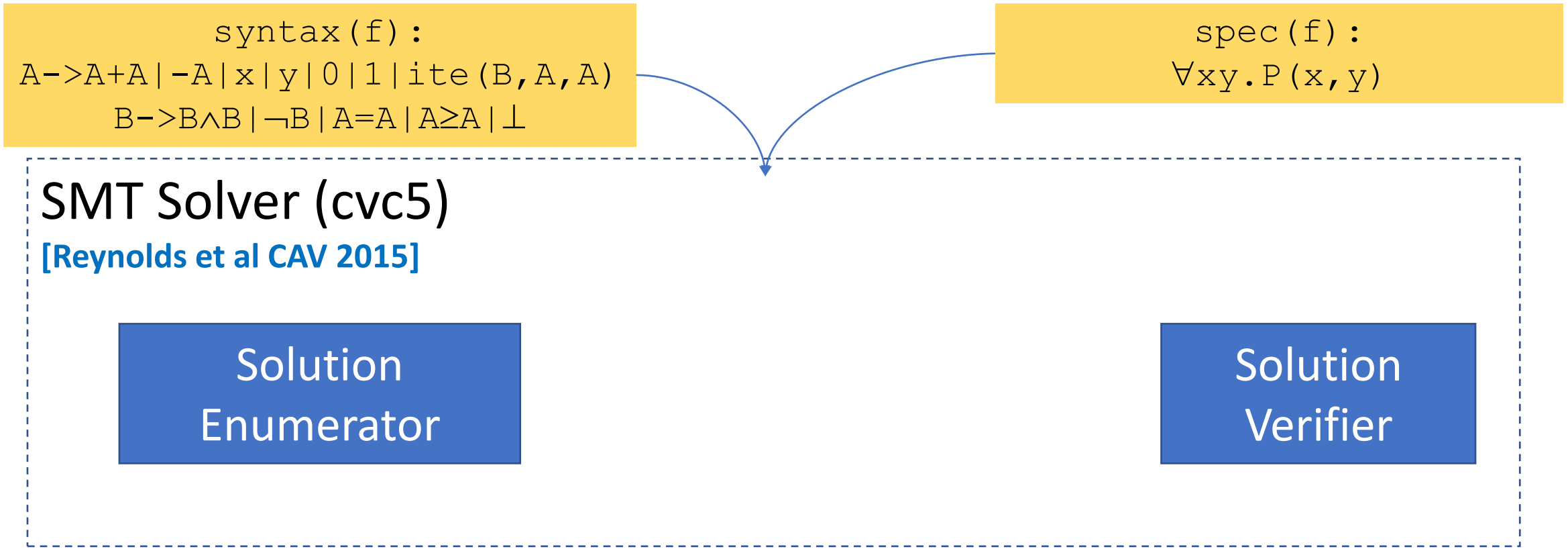
```
spec(f):
∀xy.P(x,y)
```

Solution Enumerator

SMT Solver

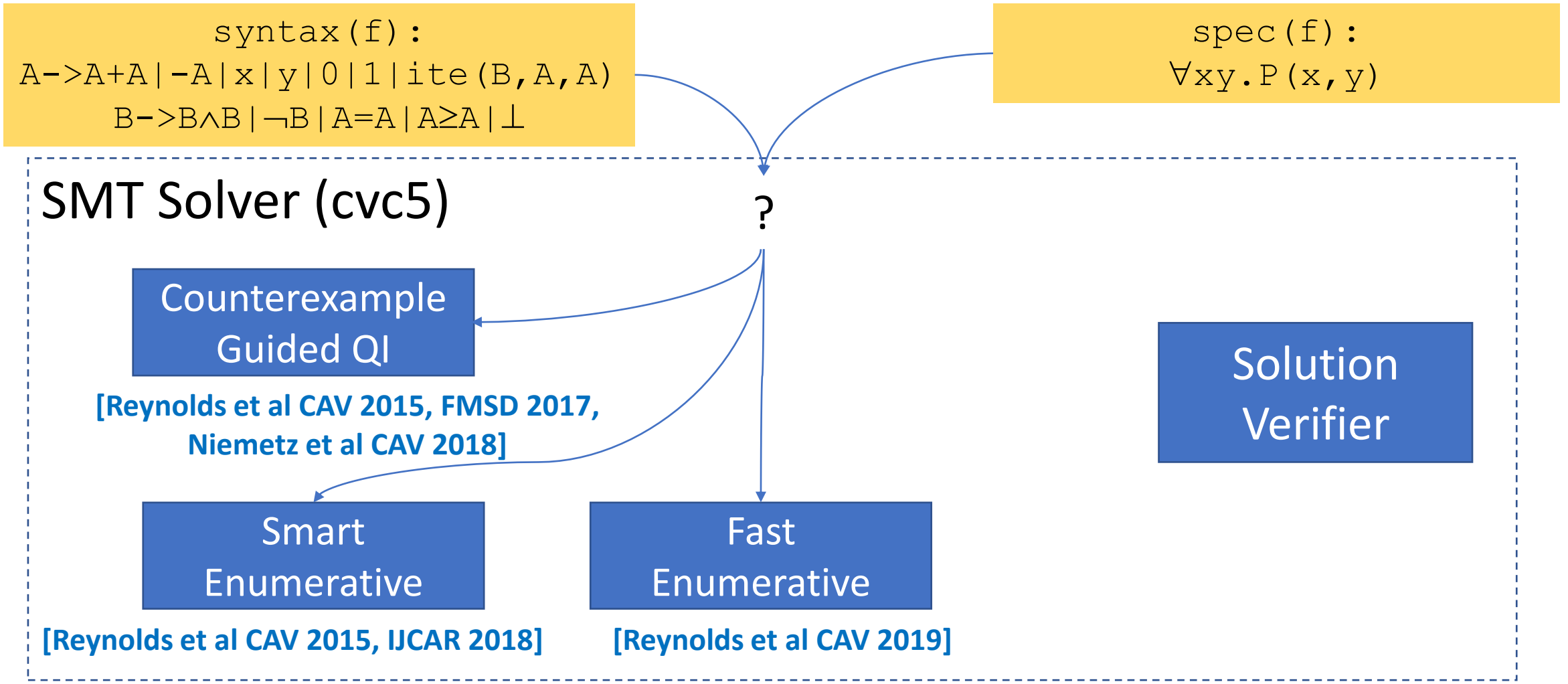
Solution Verifier

SyGuS inside an SMT solver



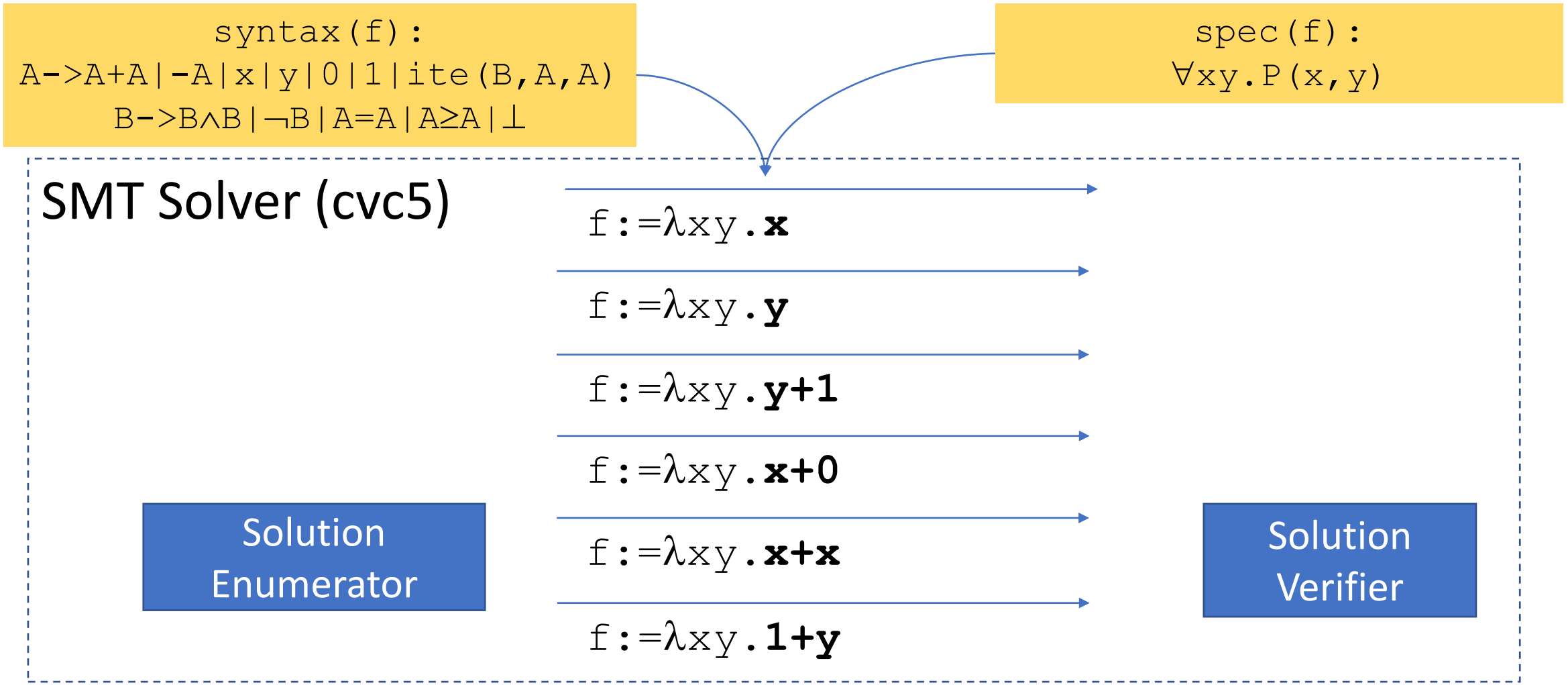
- ✔ Synthesis algorithms that use internal state of SMT solver
- ✔ Tight integration between enumerator and verifier

SyGuS inside an SMT solver

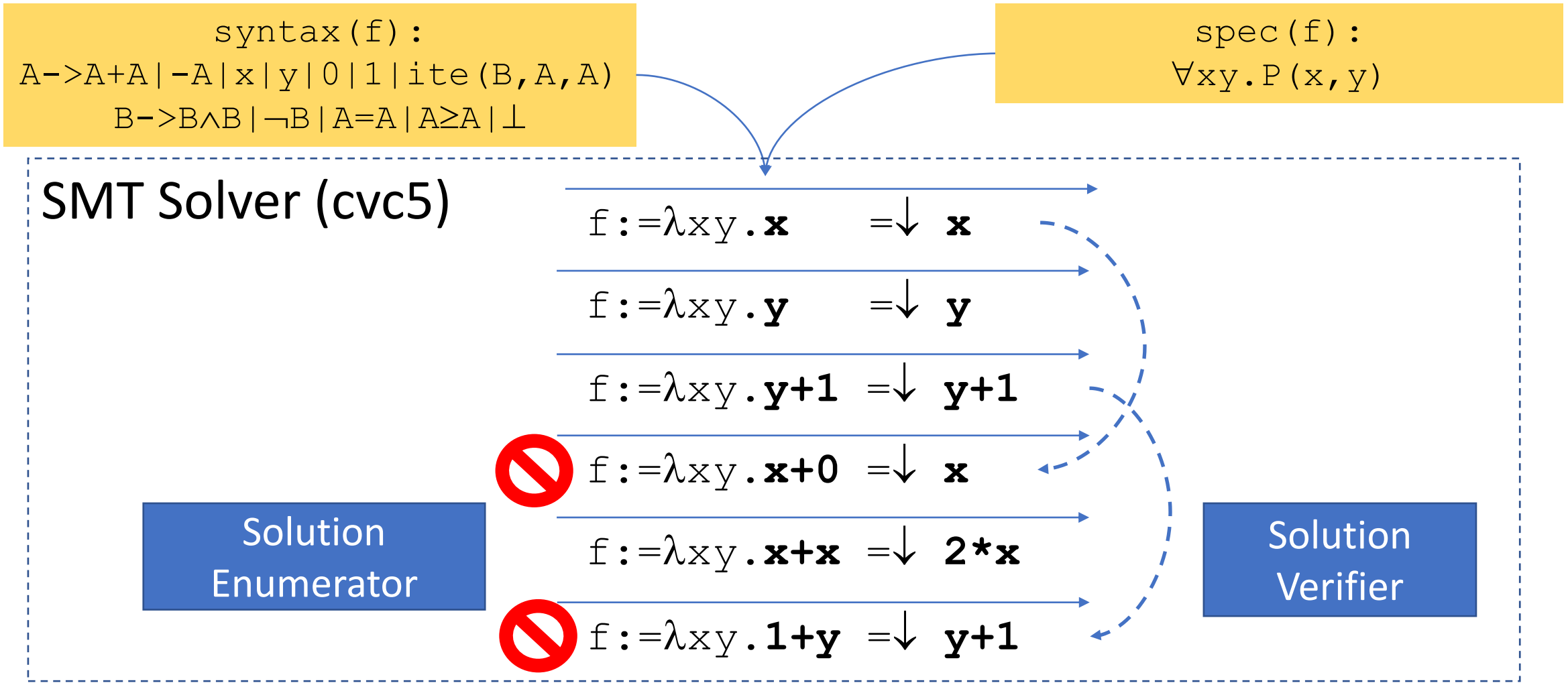


⇒ Best approach to apply depends on the conjecture

SyGuS inside an SMT solver

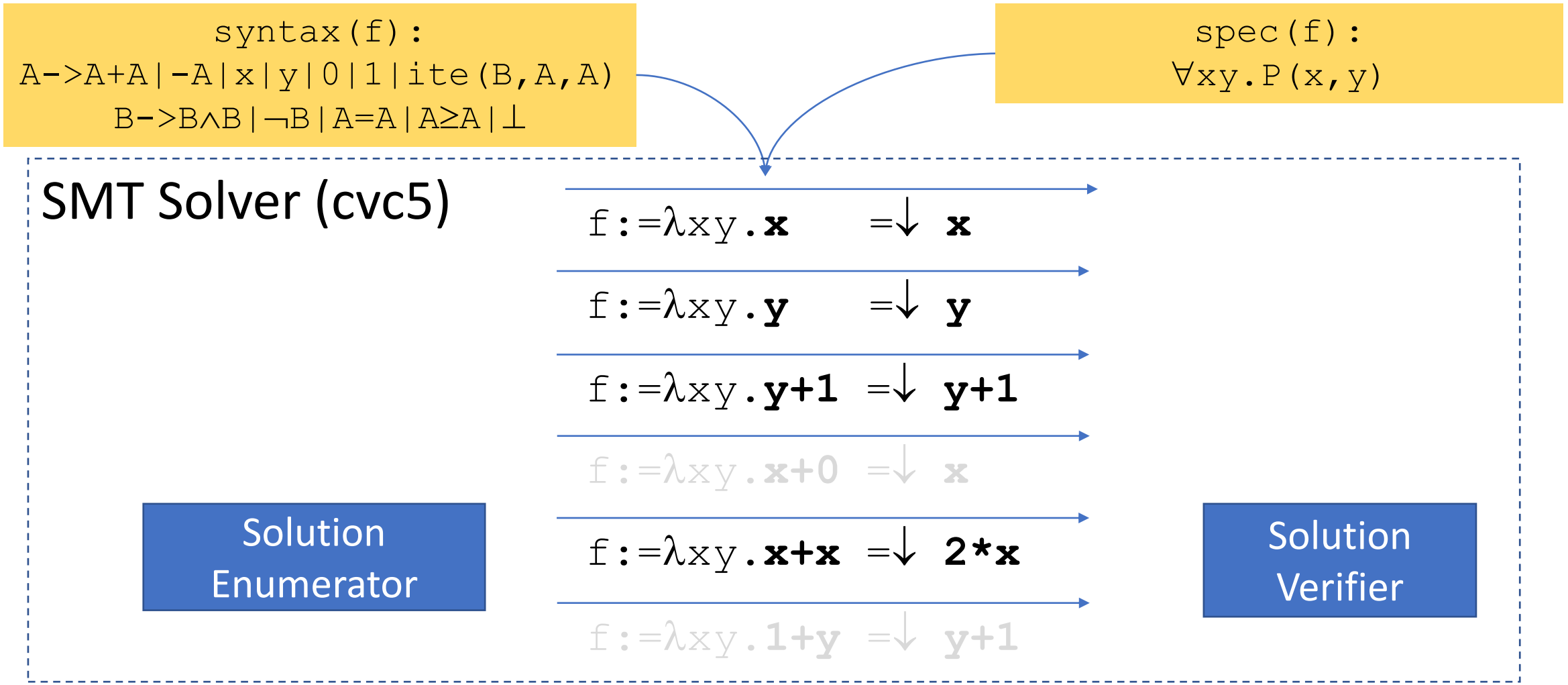


SyGuS inside an SMT solver

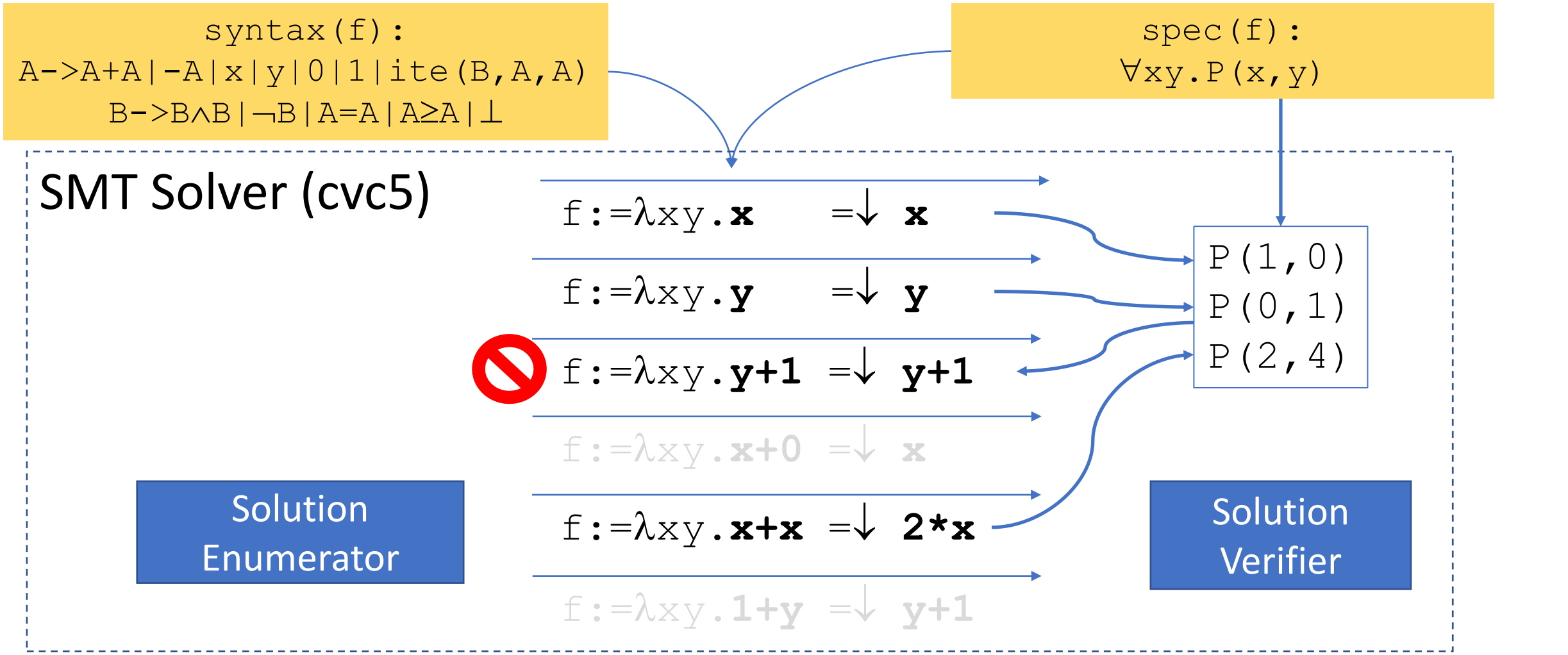


⇒ Leverage aggressive rewriting to avoid redundant candidates

SyGuS inside an SMT solver



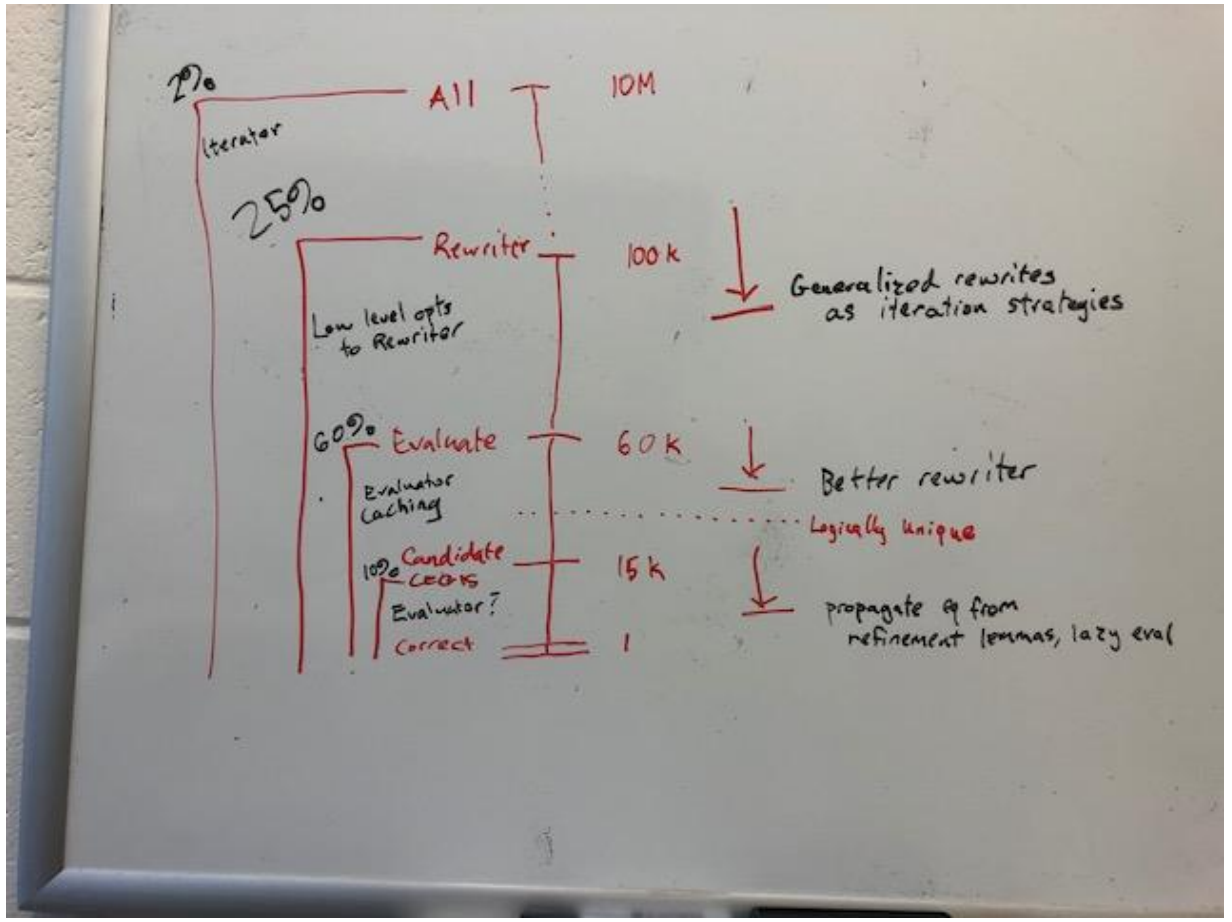
SyGuS inside an SMT solver



⇒ Cache counterexamples to spec and use fast evaluation

DEMO

Profiling Enumerative SyGuS



- Rewriting and evaluating terms on concrete examples is the bottleneck (~85% of runtime)

Using an SMT Solver (cvc5) to *Improve Itself*

1. Rewrite Rule Synthesis

$$\exists t_1 t_2 . \forall x . t_1[x] = t_2[x],\ t_1 \not\to t_2$$

[Noetzli et al SAT 2019]

2. Semantic-guided test case generation

$$\exists Q . \exists_{\leq n} x \in S .\ Q(x)$$

3. Solving Quantified Bit-Vectors (and Floating Points) Using Invertibility Conditions

$$\exists IC . \forall st . IC(s,t) \Leftrightarrow \exists x . x \oplus s \sim t$$

[Niemetz et al CAV 2018,
Brain et al CAV 2019]

…can tackle these problems using syntax-guided synthesis

Application #1:
Rewrite Rule Synthesis

Rewriting is Important for SMT Solving

- SMT solvers use a "rewriter" to put constraints in some normal form
  - E.g. `x+0`$\rightarrow$`x`, `x-y`$\rightarrow$`x+(-1*y)`, `x=x-2`$\rightarrow\bot$

- Having a good rewriter is highly critical to performance
  - In particular, theory of bit-vectors, strings, floating points
    - Single rewrite may make problem go from hard $\rightarrow$ trivial
  - A good rewriter makes the SyGuS enumeration faster

Rewrite Rules are *Difficult to Implement*

- Hard to find commonly applicable rewrites
  - Analyze problem instances, solver runs
- What rewrites have I not already implemented?
- Time consuming, many lines of code
  - cvc5's BV rewriter ~3500 LOC
  - cvc5's string rewriter >5000 LOC

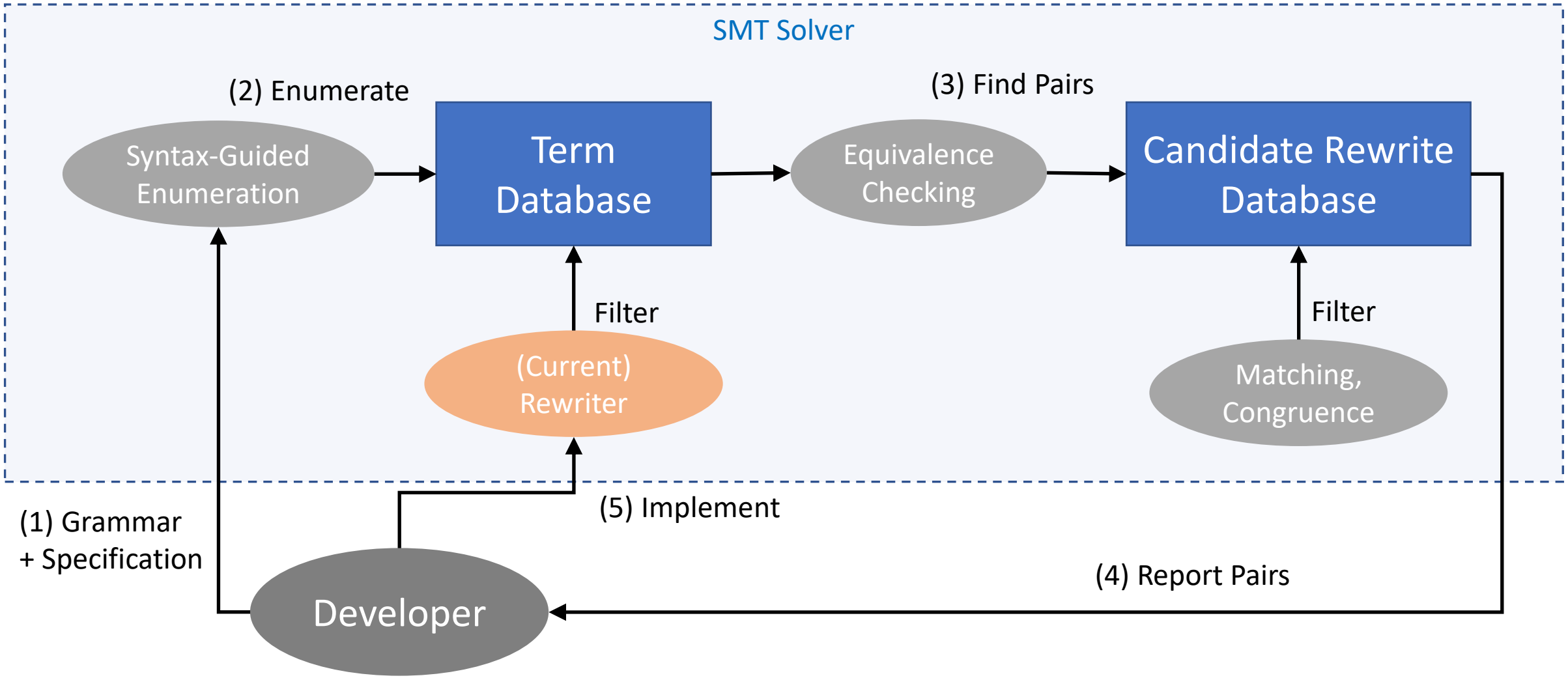
Rewrite Rule Synthesis

- Use the syntax-guided enumeration to assist the developer to implement the solver's rewriter

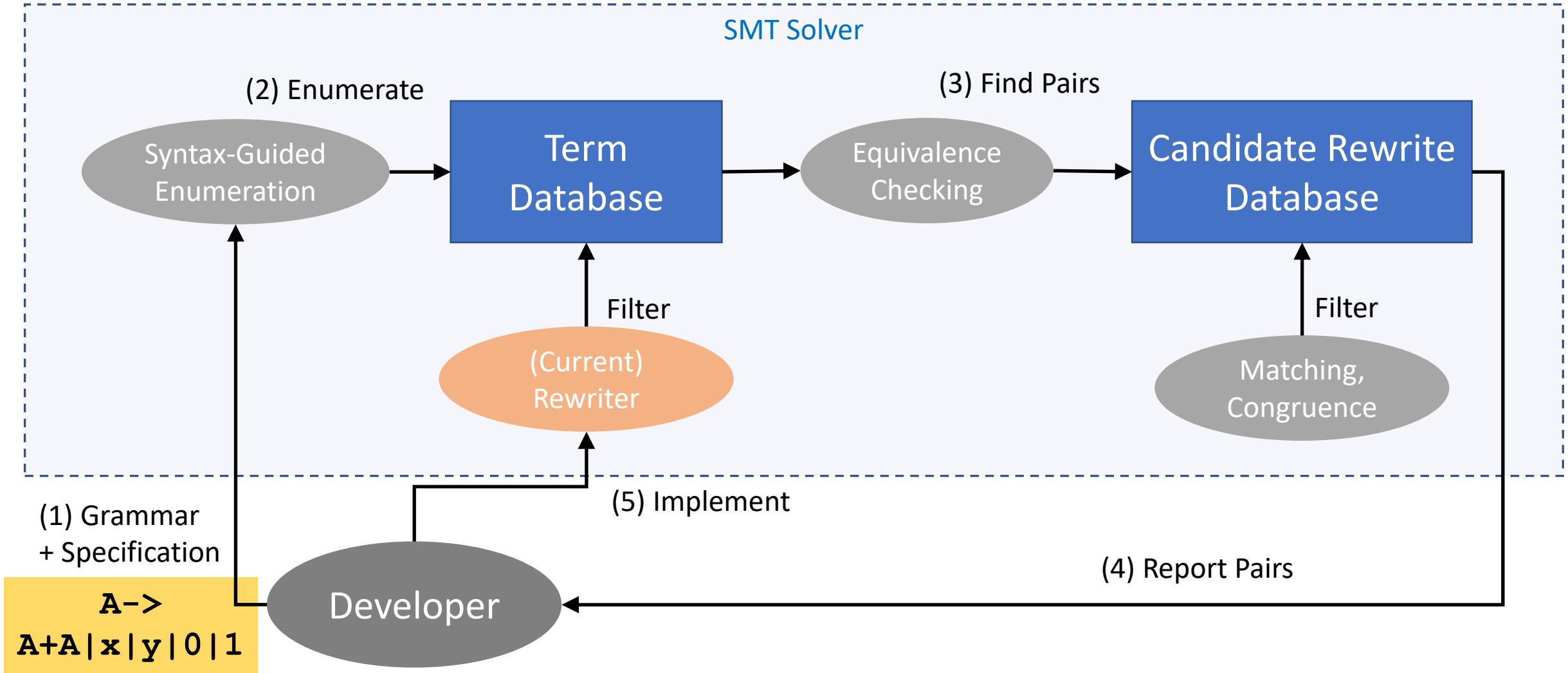
$\Rightarrow$Increase productivity of the developer

$\Rightarrow$Increase confidence in the correctness of the rewriter

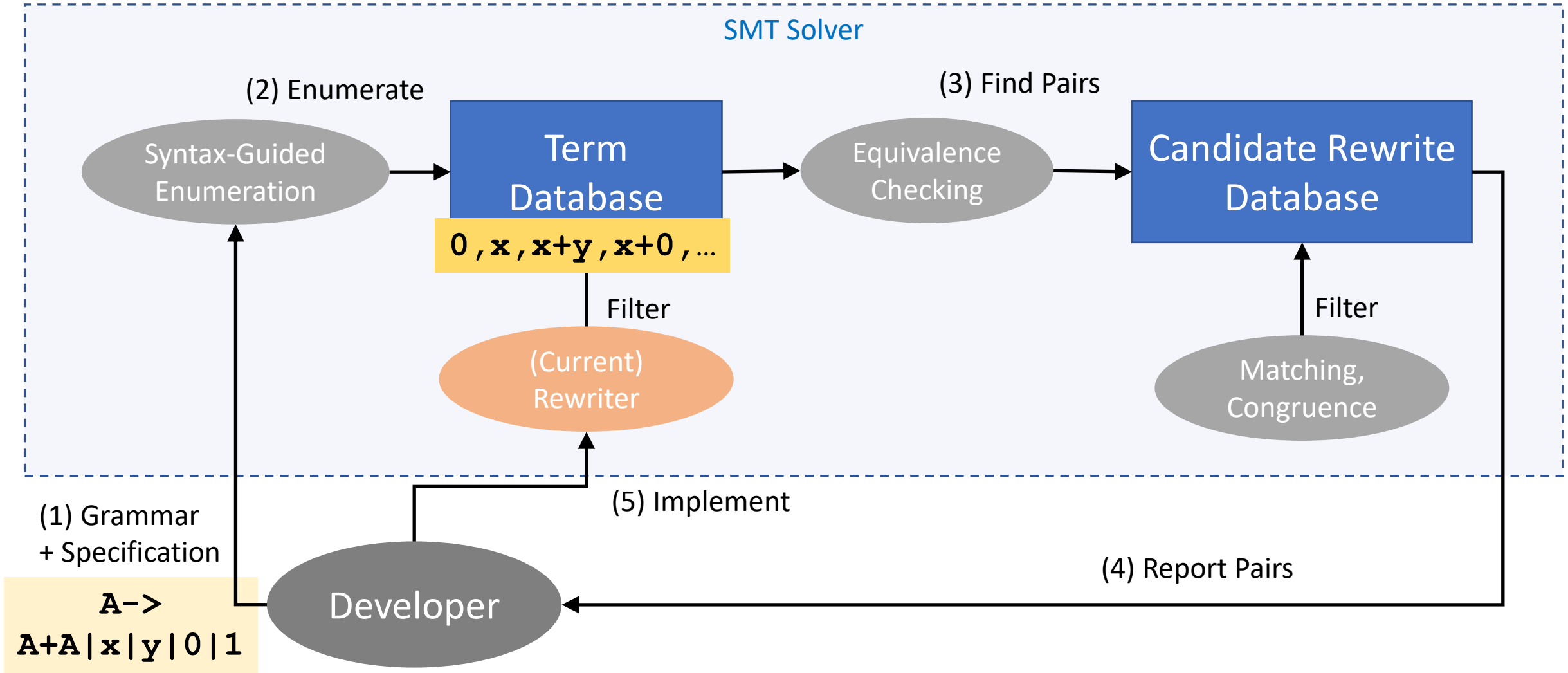
(Partially) Automated Rewrite Rule Generation



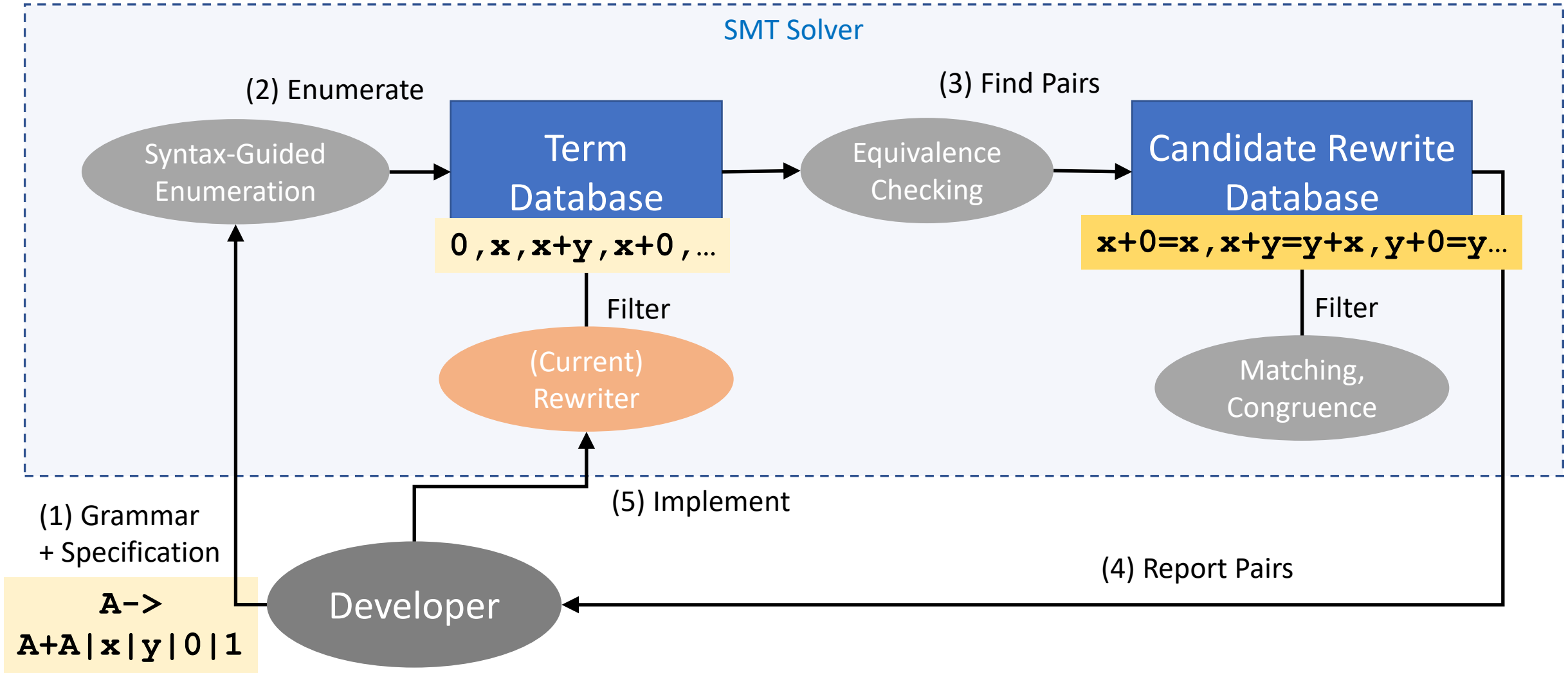
(Partially) Automated Rewrite Rule Generation



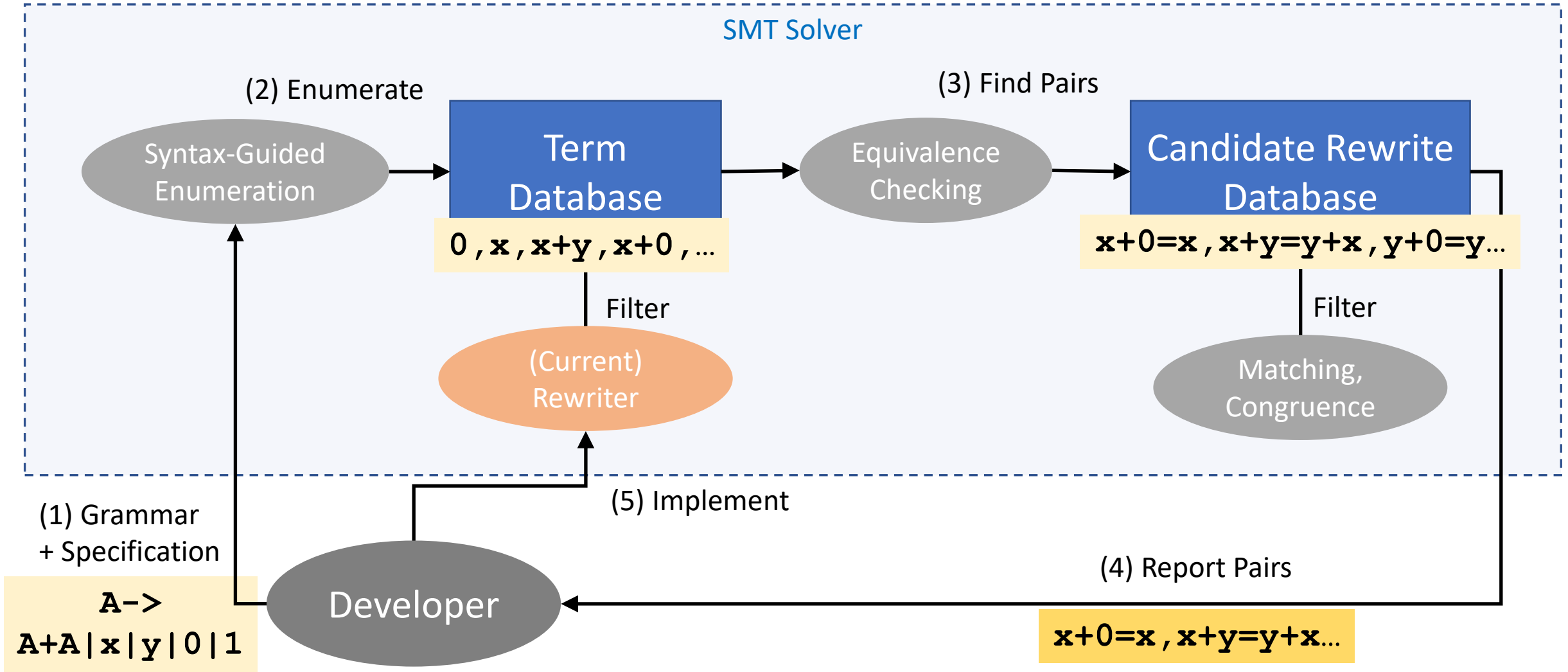
(Partially) Automated Rewrite Rule Generation



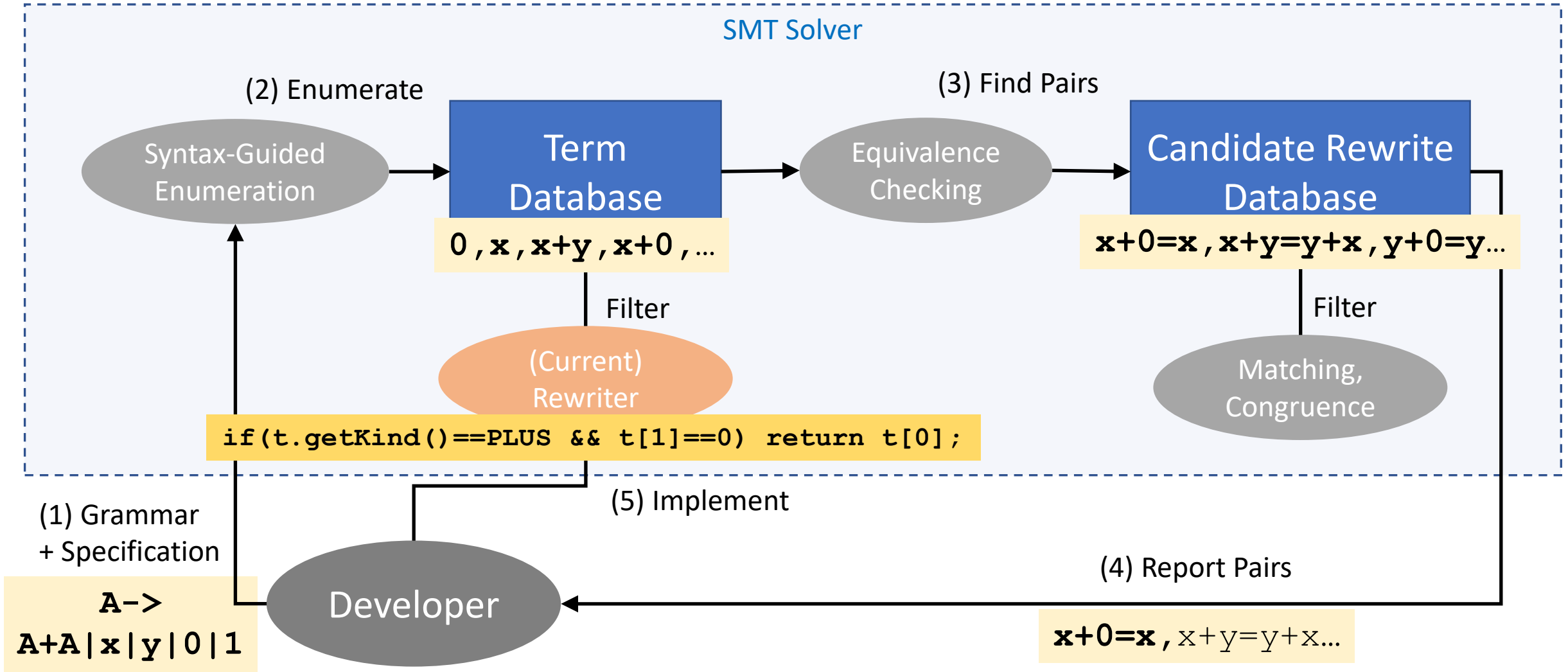
(Partially) Automated Rewrite Rule Generation



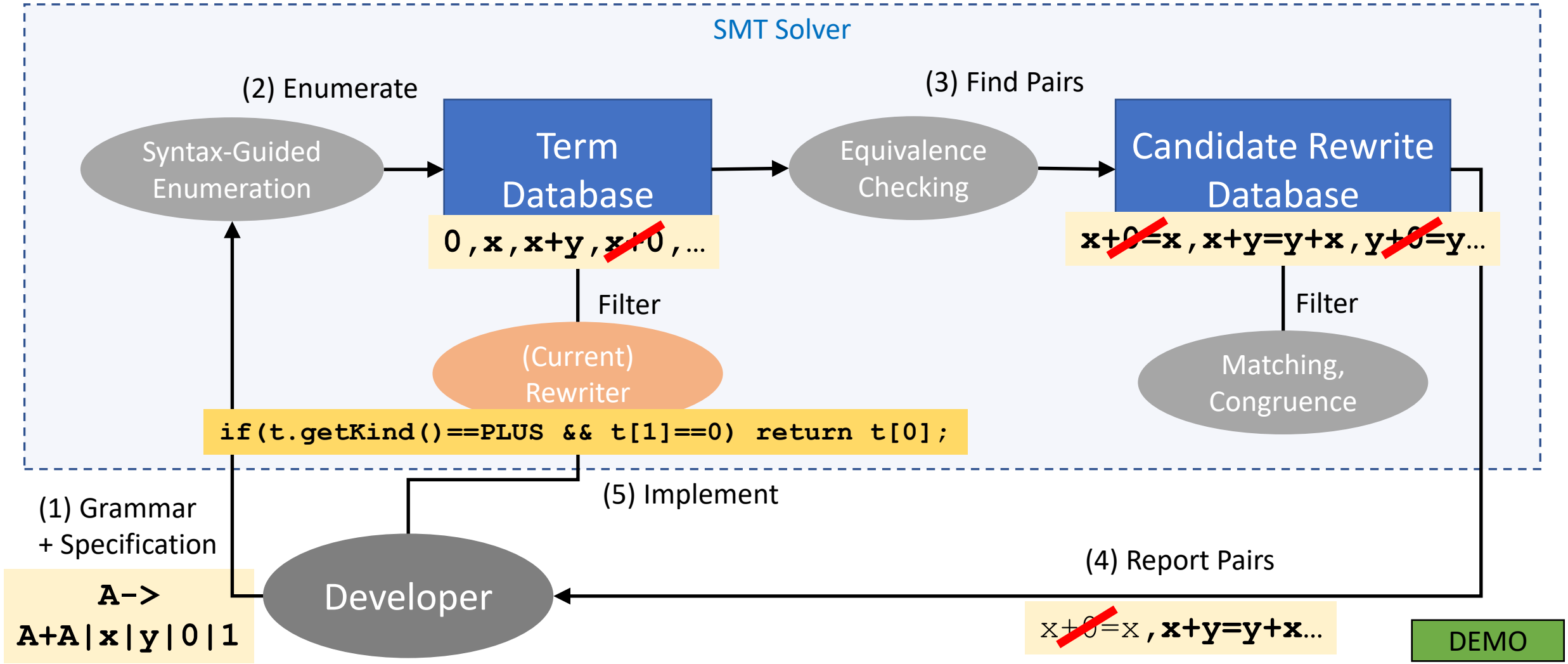
(Partially) Automated Rewrite Rule Generation



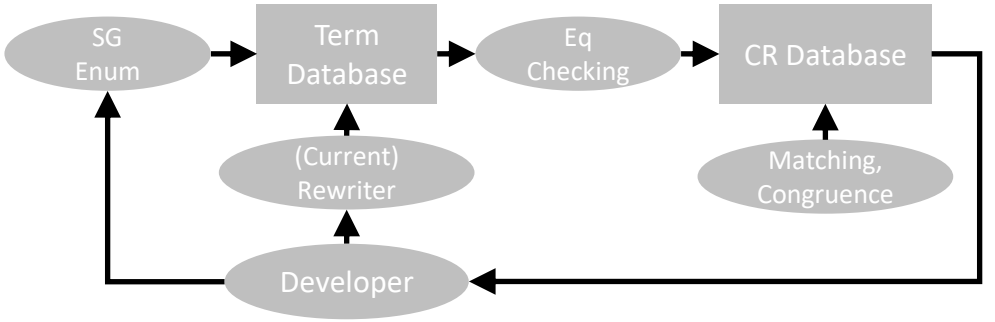
(Partially) Automated Rewrite Rule Generation



(Partially) Automated Rewrite Rule Generation



Experience

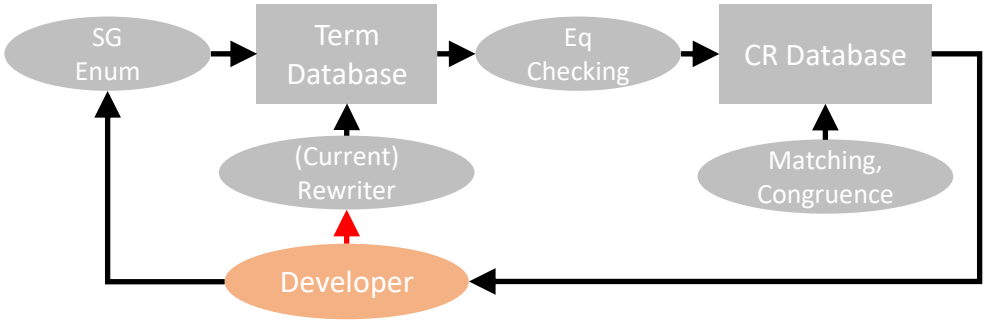



```
(synth-fun f
 ((x String) (y String) (z Int))
 String (
 (Start String (
  x y "A" "B" ""
  (str.++ Start Start)
  (str.replace Start Start Start)
  (str.at Start ie)
  (int.to.str ie)
  (str.substr Start ie ie)))
 (ie Int (
  0 1 z
  (+ ie ie)
  (- ie ie)
  (str.len Start)
  (str.to.int Start)
  (str.indexof Start Start ie)))))
```

```
(synth-fun f ((s (BitVec 4))
              (t (BitVec 4)))
 (BitVec 4) (
 (Start (BitVec 4) (
  s t #x0
  (bvneg  Start)
  (bvnot  Start)
  (bvadd  Start Start)
  (bvmul  Start Start)
  (bvand  Start Start)
  (bvlshr Start Start)
  (bvor   Start Start)
  (bvshl  Start Start)))))
```

```
(synth-fun f
 ((x Bool) (y Bool)
  (z Bool) (w Bool))
 Bool (
 (Start Bool (
  (and d1 d1) (not d1)
  (or d1 d1) (xor d1 d1)))
 (d1 Bool (
  x (and d2 d2) (not d2)
  (or d2 d2) (xor d2 d2)))
 (d2 Bool (
  w (and d3 d3) (not d3)
  (or d3 d3) (xor d3 d3)))
 (d3 Bool (
  y (and d4 d4) (not d4)
  (or d4 d4) (xor d4 d4)))
 (d4 Bool (z))))
```

Examples of Rewrites



- Bit-Vectors

  `bvlshr(x,x)` → `#x0000` `x-(x&y)` → `x&~y` `concat(#x1,x)=concat(#x0,y)` → ⊥

  `x+1` → `~(-x)` `(x&y)+(x|y)` → `x+y` `bvxor(x,x&y)` → `~y&x`

- Strings

  `x++"A"="B"++x` → ⊥ `indexof("ABCDE",x,3)` → `indexof("AAADE",x,3)`

  `contains(x,x++"A")` → ⊥ `replace(x,x++y,y)` → `replace(x,x++y,"")`

  `"A"++x="A"++y` → `x=y` `contains("ABCD","C"++x++"B")` → ⊥

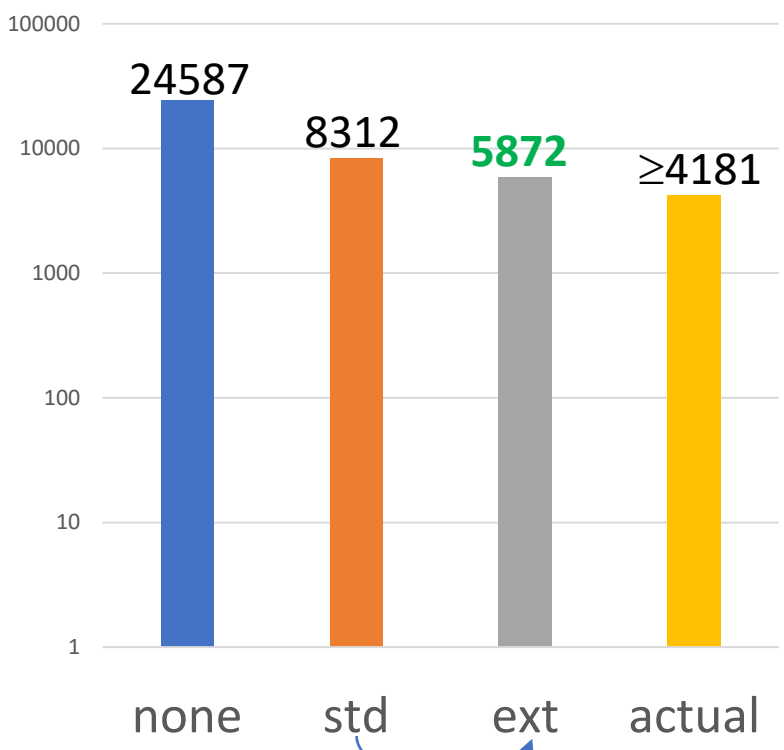
- Booleans

  A∧(A∨B) → A∧B (A∨C)∧(A∨B) → A∧(C∨B)

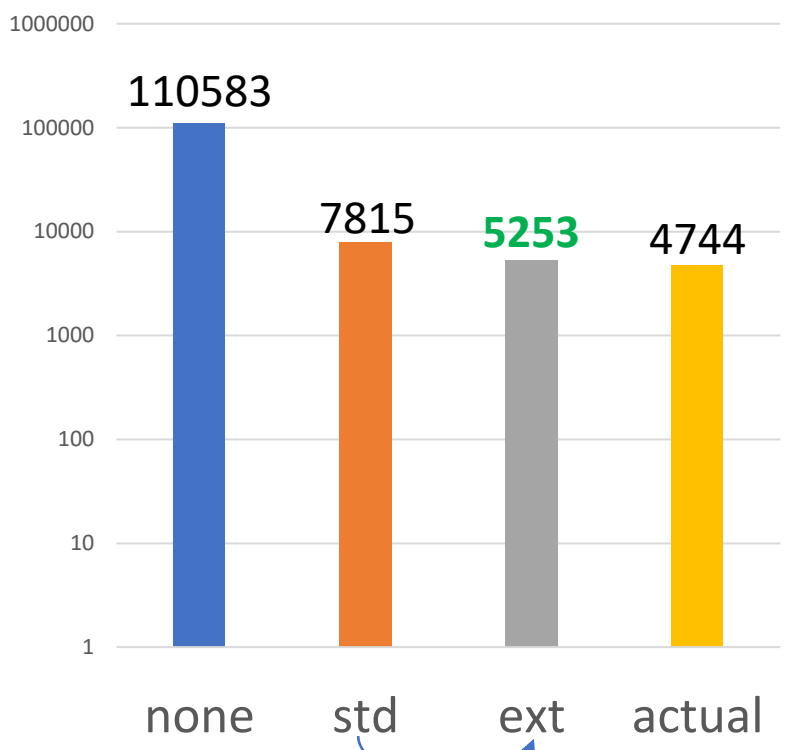
  A=A&B → ¬A∨B (A∨B)=(A∨B∨C) → A∨B∨¬C

Statistics: cvc5's Rewriter (circa 2019):

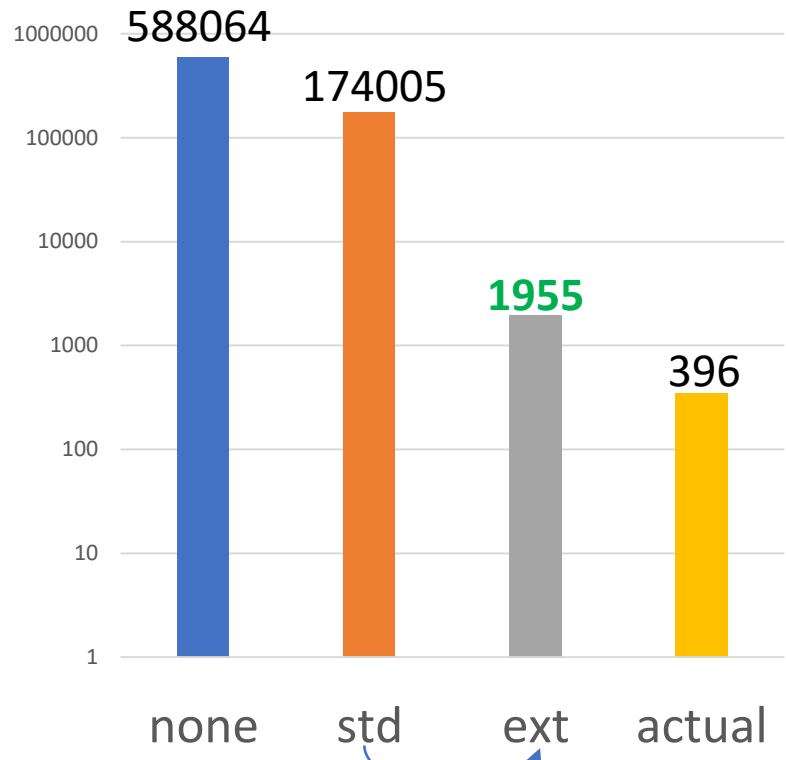
string-term, depth 2



bv-term, depth 3



bool-crci, depth 7



	string-term, depth 2		bv-term, depth 3		bool-crci, depth 7	
%redundant:	49.7%	**28.8%**	39.3%	**9.7%**	99.8%	**82.2%**
time to enumerate:	90.0	**60.8**	96.4	**55.4**	19128.8	**60.6**

[Noetzli/Reynolds/Barbosa/Niemetz/Preiner/Barrett/Tinelli SAT 19]

Impact of Aggressive Simplification *for Strings*

Set		all	-arith	-contain	-msets	Z3	OSTRICH
CMU	sat	7947	7746	**7948**	7946	4585	
	unsat	**66**	31	**66**	**66**	52	
	×	173	409	172	174	3549	
TERMEQ	sat	**10**	**10**	**10**	**10**	1	
	unsat	**49**	36	27	**49**	36	
	×	22	35	44	22	44	
SLOG	sat	**1302**	**1302**	**1302**	**1302**	1100	1289
	unsat	**2082**	**2082**	**2082**	**2082**	2075	**2082**
	×	7	7	7	7	216	20
APLAS	sat	**132**	**132**	**132**	**132**	10	
	unsat	**292**	291	171	171	94	
	×	159	160	280	280	479	
Total	sat	9391	9190	**9392**	9390	5696	1289
	unsat	**2489**	2440	2346	2368	2257	2082
	×	361	611	503	483	4288	8870

[Reynolds/Noetzli/Tinelli/Barrett CAV 19]

-arith: w/o arithmetic simplifications
-contain: w/o contain-based simplifications
-mset: w/o multiset-based simplifications

- cvc5 uses >5000 lines of C++ for simplification rules (and growing)
  - Important aspect of modern string solving

Application #2: Improving Confidence in the SMT Solver

Improving Confidence: Rewriting

- *Does my SMT solver implement any unsound rewrites?*

Improving Confidence: Rewriting

Grammar of interest

```
(synth-fun f ((s BV8) (t BV8)) BV8
   ((Start BV8
      (s t #x00 #x01
       ~Start
       -Start
       bvlshr(Start Start)
       Start & Start
       Start + Start
   )))
```

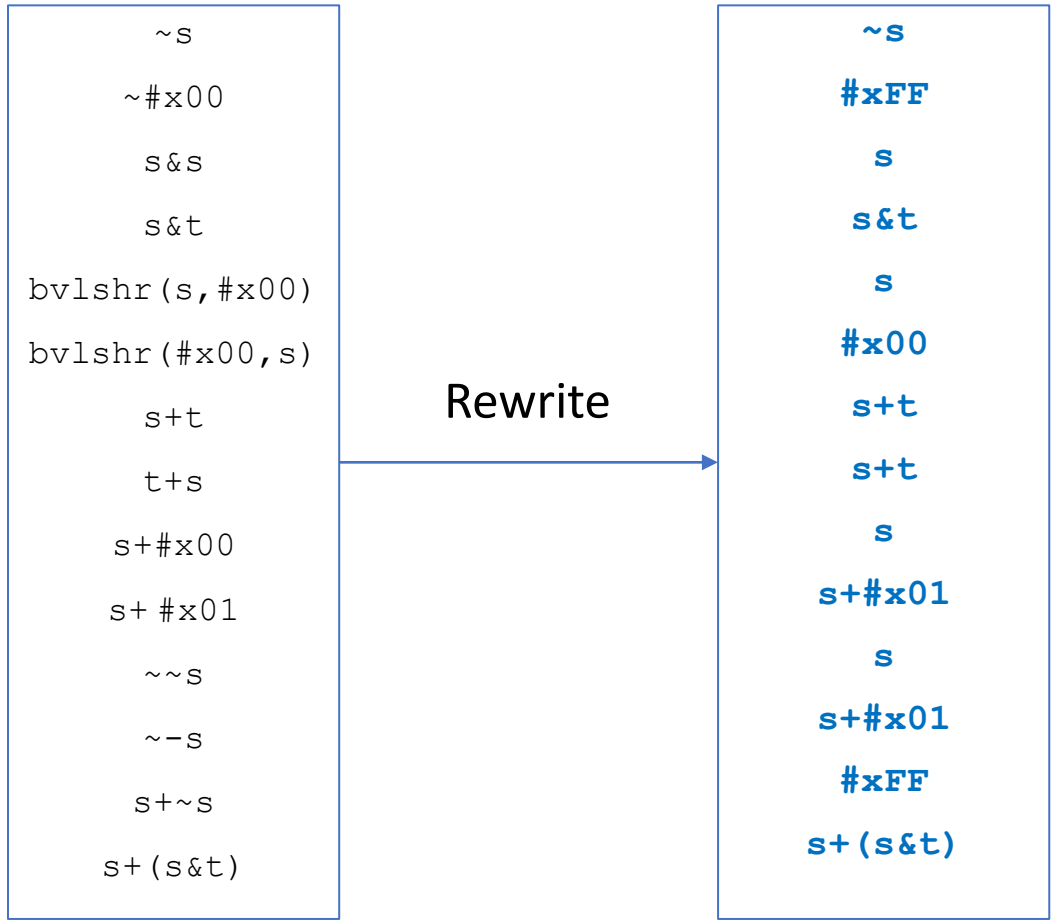
Improving Confidence: Rewriting

```
~s
~#x00
s&s
s&t
bvlshr(s,#x00)
bvlshr(#x00,s)
s+t
t+s
s+#x00
s+#x01
~~s
~-s
s+~s
s+(s&t)
```

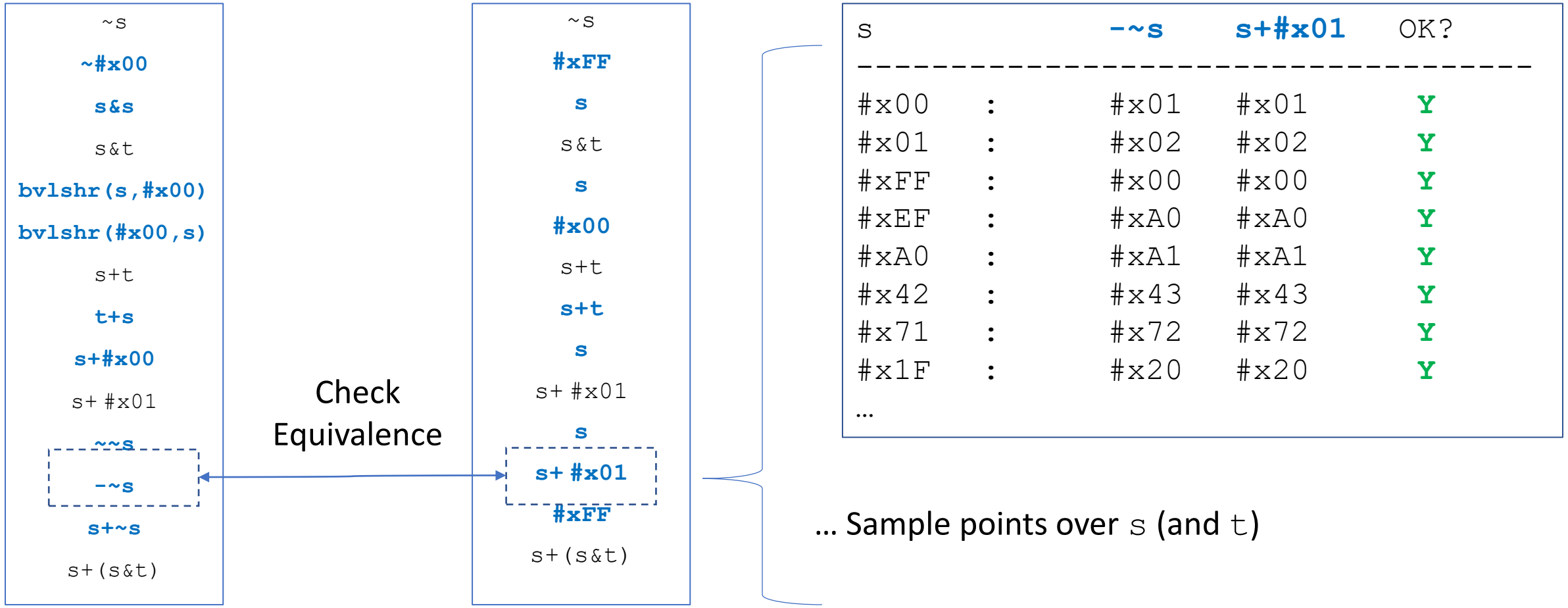
Syntax-Guided Term Enumeration

```
(synth-fun f ((s BV8) (t BV8)) BV8
   ((Start BV8
      (s t #x00 #x01
       ~Start
       -Start
       bvlshr(Start Start)
       Start & Start
       Start + Start
   )))
```

Improving Confidence: Rewriting



Improving Confidence: Rewriting

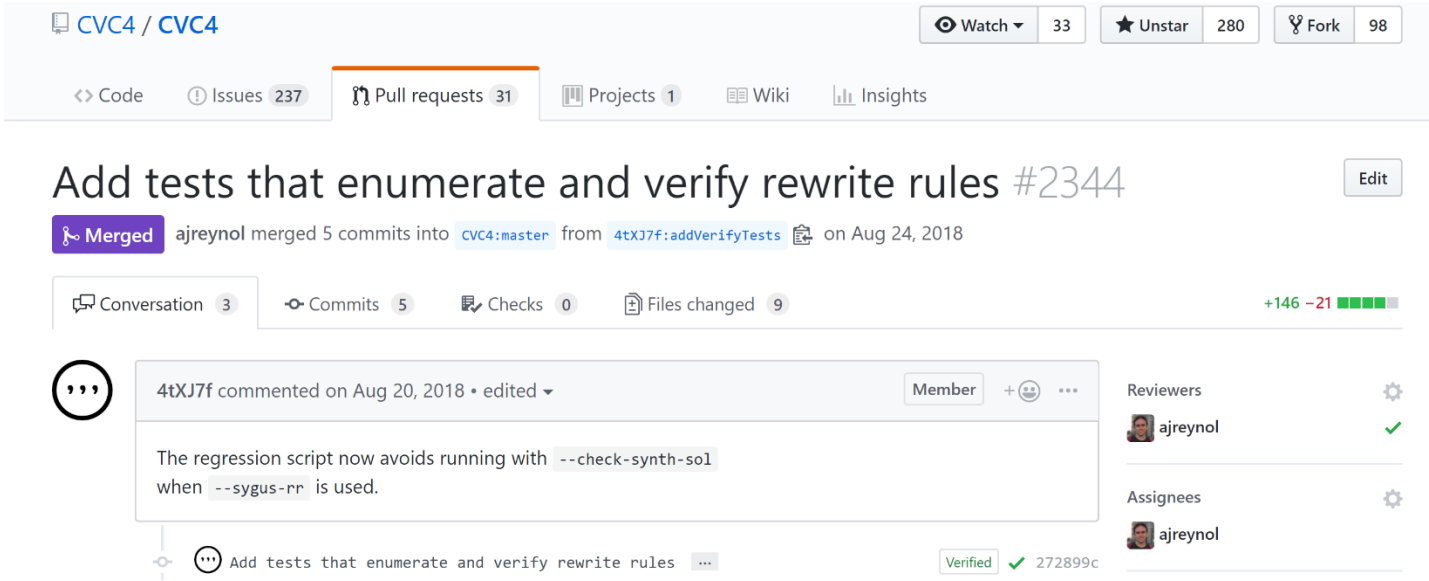


Improving Confidence: Rewriting

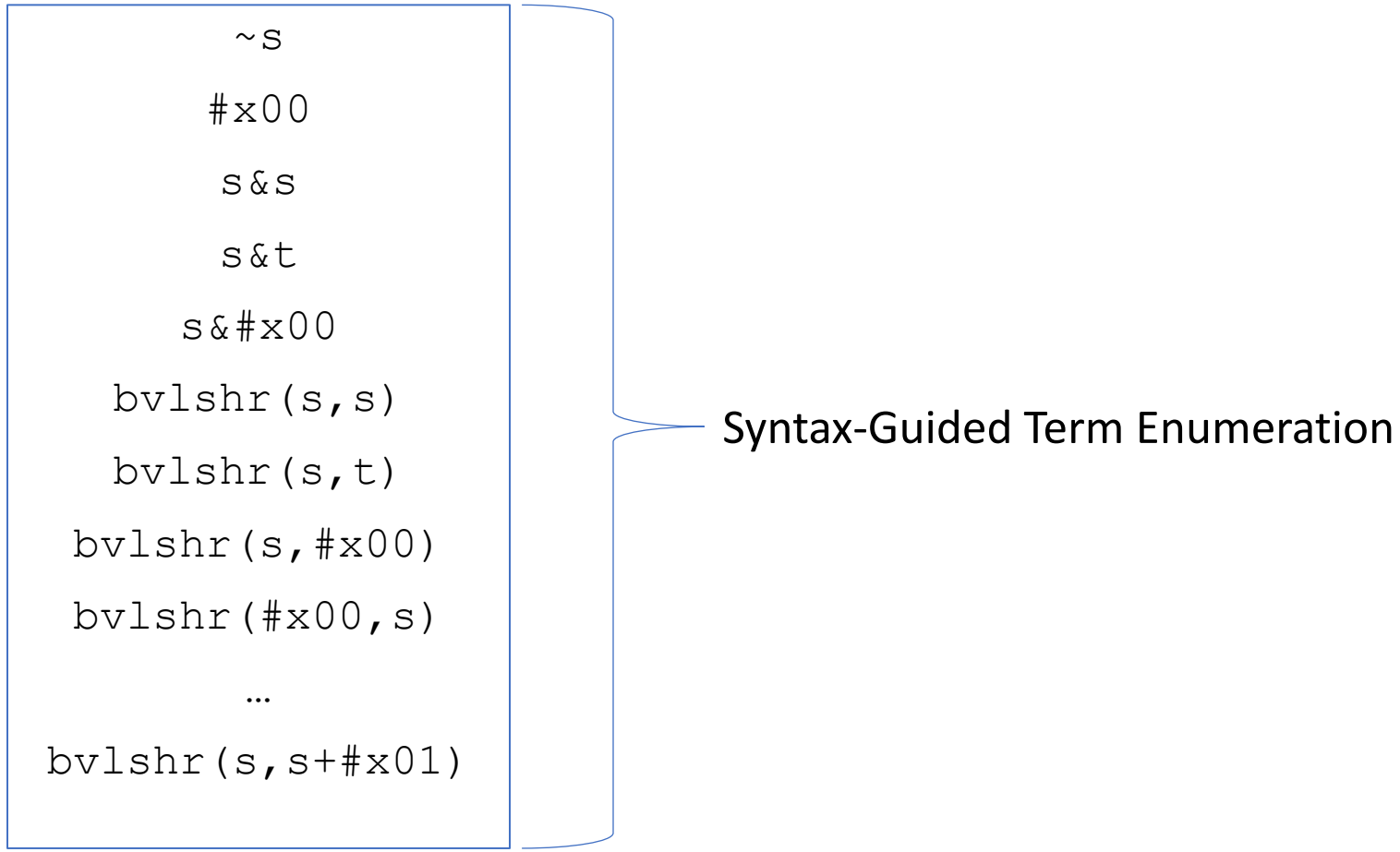
⇒Has been critical for *finding bugs* in newly written rewriter code

```
(unsound-rewrite (bvuge (bvadd x #x0001) x) true)
; --sygus-rr-verify detected unsoundness in the rewriter!
; Terms have the same rewritten form but are not equivalent
; for x=#xFFFF, where they evaluate to:
; (bvuge (bvadd x #x0001) x) = false
; true = true
```

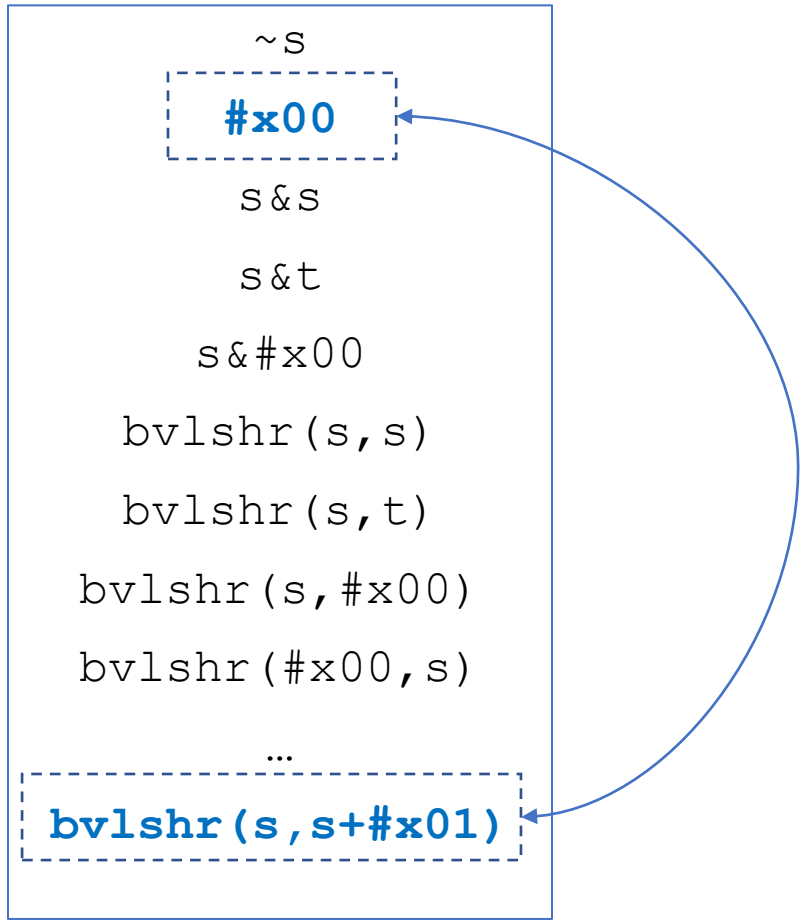
⇒Run as part of cvc5's regression tests:



Improving Confidence: Query Generation



Improving Confidence: Query Generation



Check Equivalence

s		#x00	bvlshr(s,s+#x01)	=?
#x00	:	#x00	#x01	Y
#x01	:	#x00	#x00	Y
#xFF	:	#x00	#xFF	N
#xEF	:	#x00	#x00	Y
#xA0	:	#x00	#x00	Y
#x42	:	#x00	#x00	Y
#x71	:	#x00	#x00	Y
#x1F	:	#x00	#x00	Y

…

- **Intuition:** terms that are “almost equivalent” => inputs that are “almost unsatisfiable”

Application #3: Solving Quantified Inputs via Invertibility Conditions

Solving Quantified Bit-Vectors: Example

- Consider quantifier elimination problem for bit-vectors:

$$\exists x{:}BV_8.\, s*x=t \;\Leftrightarrow\; ???$$

$\Rightarrow$ Applications in proving safety properties, compiler optimzations

- Common solving technique is model-based instantiation **[Wintersteiger et al 2013]**
  - Based on *values*: above is equivalent $s*\#x00=t \vee s*\#x01=t \vee s*\#x02=t \vee \ldots$
  - Scalability issues for larger bit-widths

- **Idea:** Use SyGuS to find *invertibility conditions* for bit-vector equations

  $\Rightarrow$ Compact form for quantifier elimination

What is an Invertibility Condition?

- Some equations are “invertible”, e.g.:
  - `s+x=t` … always has solution **`x=t-s`**
- Some equations are not, e.g.:
  - `s*x=t` … `x` has *no solution* when, e.g. `s=2`, `t=3`

[Niemetz/Preiner/Reynolds/Barrett/Tinelli CAV 2018]

What is an Invertibility Condition?

- `s+x=t` … always has solution **`x=t-s`**
- `s*x=t` … `x` has *no solution* when, e.g. `s=2, t=3`

- **Challenge:** it is possible to characterize exactly when `x` has a solution?
  - Find a predicate `IC(s,t)` such that
    $\exists$`x.(x`$\oplus$`s~t)` is satisfiabile iff `IC(s,t)` is satisfiable
  - We call `IC` an "invertibility condition"
  - Invertibility Conditions $\Rightarrow$ QE procedure for "linear" formulas

[Niemetz/Preiner/Reynolds/Barrett/Tinelli CAV 2018]

Finding Invertibility Conditions for BV

- Applied to theory of Bit-Vectors **[Niemetz et al CAV 2018]**
- Signature of BV has 15 operators, 4 relations (=/≠, signed/unsigned inequality)
  $\Rightarrow$ Total of 162 invertibility conditions to find
- Some took several hours to find by hand $\therefore$ *use SyGuS*
  - Using SyGuS, found **118** of **162** conditions
    - Many simpler than hand-crafted ones
  - When combined with hand-crafted ICs, found all **162** conditions

Invertibility Conditions for BV

$\ell[x]$	$\approx$	$\not\approx$
$x \cdot s \bowtie t$	$(-s \mid s)\ \&\ t \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \bmod s \bowtie t$	$\sim(-s) \geq_u t$	$s \not\approx 1 \vee t \not\approx 0$
$s \bmod x \bowtie t$	$(t + t - s)\ \&\ s \geq_u t$	$s \not\approx 0 \vee t \not\approx 0$
$x \div s \bowtie t$	$(s \cdot t) \div s \approx t$	$s \not\approx 0 \vee t \not\approx \sim 0$
$s \div x \bowtie t$	$s \div (s \div t) \approx t$	$\begin{cases} s\ \&\ t \approx 0 & \text{for } \kappa(s) = 1 \\ \top & \text{otherwise} \end{cases}$
$x\ \&\ s \bowtie t$	$t\ \&\ s \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \mid s \bowtie t$	$t \mid s \approx t$	$s \not\approx \sim 0 \vee t \not\approx \sim 0$
$x \gg s \bowtie t$	$(t \ll s) \gg s \approx t$	$t \not\approx 0 \vee s <_u \kappa(s)$
$s \gg x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \gg i \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \gg_a s \bowtie t$	$(s <_u \kappa(s) \Rightarrow (t \ll s) \gg_a s \approx t) \wedge$ $(s \geq_u \kappa(s) \Rightarrow (t \approx \sim 0 \vee t \approx 0))$	$\top$
$s \gg_a x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \gg_a i \approx t$	$(t \not\approx 0 \vee s \not\approx 0) \wedge$ $(t \not\approx \sim 0 \vee s \not\approx \sim 0)$
$x \ll s \bowtie t$	$(t \gg s) \ll s \approx t$	$t \not\approx 0 \vee s <_u \kappa(s)$
$s \ll x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \ll i \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \circ s \bowtie t$	$s \approx t[\kappa(s) - 1 : 0]$	$\top$
$s \circ x \bowtie t$	$s \approx t[\kappa(t) - 1 : \kappa(t) - \kappa(s)]$	$\top$

Table 2. Conditions for the invertibility of bit-vector operators over (dis)equality. Those for $\cdot$, $\&$ and $\mid$ are given modulo commutativity of those operators.

ICs for =,≠

…not pictured:
ICs for `bvuge`, `bvugt`, `bvsge`, `bvsgt`

[Niemetz/Preiner/Reynolds/Barrett/Tinelli CAV 2018]

Invertibility Conditions for BV

$\ell[x]$	$\approx$	$\not\approx$
$x \cdot s \bowtie t$	$(-s \mid s)\ \&\ t \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \bmod s \bowtie t$	$\sim(-s) \geq_u t$	$s \not\approx 1 \vee t \not\approx 0$
$s \bmod x \bowtie t$	$(t + t - s)\ \&\ s \geq_u t$	$s \not\approx 0 \vee t \not\approx 0$
$x \div s \bowtie t$	$(s \cdot t) \div s \approx t$	$s \not\approx 0 \vee t \not\approx \sim 0$
$s \div x \bowtie t$	$s \div (s \div t) \approx t$	$\begin{cases} s\ \&\ t \approx 0 & \text{for } \kappa(s) = 1 \\ \top & \text{otherwise} \end{cases}$
$x\ \&\ s \bowtie t$	$t\ \&\ s \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \mid s \bowtie t$	$t \mid s \approx t$	$s \not\approx \sim 0 \vee t \not\approx \sim 0$
$x \gg s \bowtie t$	$(t \ll s) \gg s \approx t$	$t \not\approx 0 \vee s <_u \kappa(s)$
$s \gg x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \gg i \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \gg_a s \bowtie t$	$(s <_u \kappa(s) \Rightarrow (t \ll s) \gg_a s \approx t) \wedge$ $(s \geq_u \kappa(s) \Rightarrow (t \approx \sim 0 \vee t \approx 0))$	$\top$
$s \gg_a x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \gg_a i \approx t$	$(t \not\approx 0 \vee s \not\approx 0) \wedge$ $(t \not\approx \sim 0 \vee s \not\approx \sim 0)$
$x \ll s \bowtie t$	$(t \gg s) \ll s \approx t$	$t \not\approx 0 \vee s <_u \kappa(s)$
$s \ll x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \ll i \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \circ s \bowtie t$	$s \approx t[\kappa(s) - 1 : 0]$	$\top$
$s \circ x \bowtie t$	$s \approx t[\kappa(t) - 1 : \kappa(t) - \kappa(s)]$	$\top$

Table 2. Conditions for the invertibility of bit-vector operators over (dis)equality. Those for $\cdot$, & and | are given modulo commutativity of those operators.

$$\exists x.\ s*x=t$$

$$\Leftrightarrow$$

$$(-s \mid s)\ \&\ t=t$$

- Condition above encodes
  "s has fewer trailing zeroes than t"
- Conditions like this one are concise, subtle
  $\Rightarrow$ Excellent target for SyGuS

DEMO

Experimental Results

- Quantifier Instantiation based on Invertibility Conditions in CVC4
- Won quantified bit-vector (BV) category of SMT-COMP 2018

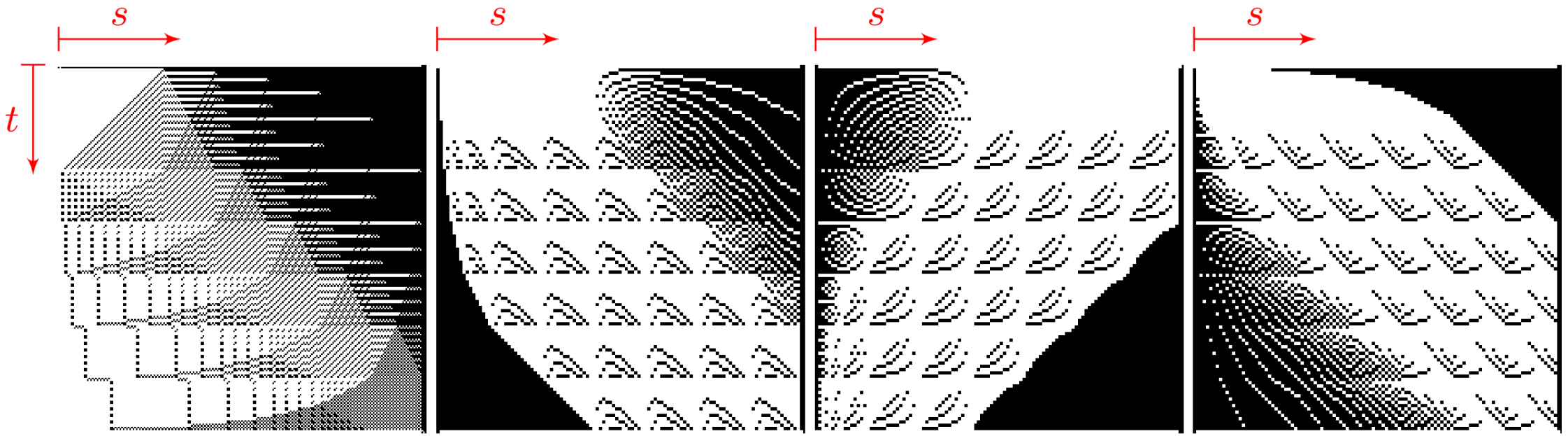
unsat	Boolector	CVC4	Q3B	Z3	cegqi$_m$	cegqi$_k$	cegqi$_s$	cegqi$_b$
h-uauto	14	12	93	24	10	103	105	**106**
keymaera	3917	3790	3781	**3923**	3803	3798	3888	3918
psyco	**62**	**62**	49	**62**	**62**	39	**62**	61
scholl	57	36	13	**67**	36	27	36	35
tptp	55	52	**56**	**56**	**56**	**56**	**56**	**56**
uauto	**137**	72	131	**137**	72	72	135	**137**
ws-fixpoint	74	71	**75**	74	**75**	74	**75**	**75**
ws-ranking	16	8	18	**19**	15	11	12	11
Total unsat	4332	4103	4216	4362	4129	4180	4369	**4399**

sat	Boolector	CVC4	Q3B	Z3	cegqi$_m$	cegqi$_k$	cegqi$_s$	cegqi$_b$
h-uauto	15	10	**17**	13	16	**17**	16	**17**
keymaera	**108**	21	24	**108**	20	13	36	75
psyco	131	**132**	50	131	**132**	60	**132**	129
scholl	**232**	160	201	204	203	188	208	211
tptp	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**
uauto	14	14	15	**16**	14	14	14	14
ws-fixpoint	45	49	**54**	36	45	51	49	50
ws-ranking	19	15	**37**	33	33	31	31	32
Total sat	**581**	418	415	558	480	391	503	545

Total (5151)	4913	4521	4631	4920	4609	4571	4872	**4944**

Table 4. Results on satisfiable and unsatisfiable benchmarks with a 300 second timeout.

…Extended to Floating Points [Brain et al CAV 19]



(a) $x + s \approx t$ (b) $x \cdot s \approx t$ (c) $s \div x \approx t$ (d) $x \div s \approx t$

$$t \approx (t \overset{\text{RTP}}{-} s) \overset{R}{+} s \vee t \approx (t \overset{\text{RTN}}{-} s) \overset{R}{+} s \vee s \approx t$$

$$t \approx (t \overset{\text{RTP}}{\div} s) \overset{R}{\cdot} s \vee t \approx (t \overset{\text{RTN}}{\div} s) \overset{R}{\cdot} s \vee (s \approx \pm\infty \wedge t \approx \pm\infty) \vee (s \approx \pm 0 \wedge t \approx \pm 0)$$

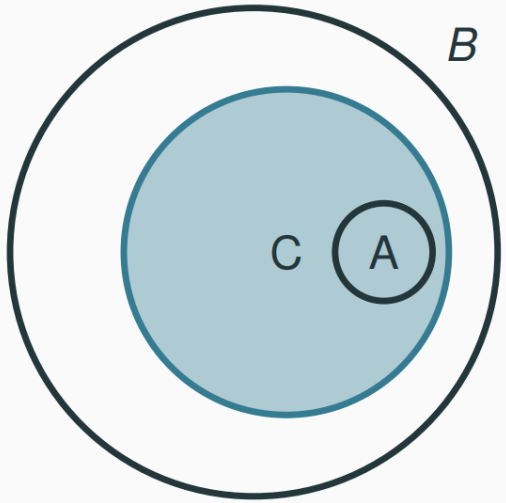
$$t \approx (s \overset{\text{RTP}}{\cdot} t) \overset{R}{\div} s \vee t \approx (s \overset{\text{RTN}}{\cdot} t) \overset{R}{\div} s \vee (s \approx \pm\infty \wedge t \approx \pm 0) \vee (t \approx \pm\infty \wedge s \approx \pm 0)$$

$$t \approx s \overset{R}{\div} (s \overset{\text{RTP}}{\div} t) \vee t \approx s \overset{R}{\div} (s \overset{\text{RTN}}{\div} t) \vee (s \approx \pm\infty \wedge t \approx \pm\infty) \vee (s \approx \pm 0 \wedge t \approx \pm 0)$$

Ongoing Applications

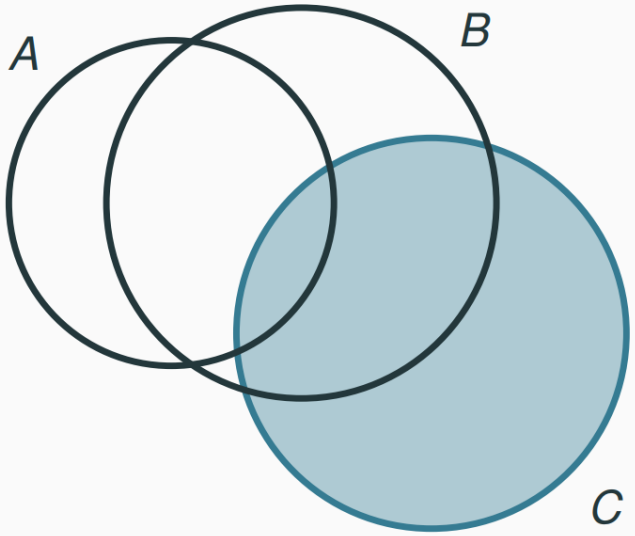
Interpolation and Abduction via SyGuS

Interpolation



Find a term C such that $A \models C$ and $C \models B$. Free symbols in C are from set of shared symbols between A and B.

Abduction



Find a term C such that $A \wedge C$ is satisfiable and $A \wedge C \models B$.

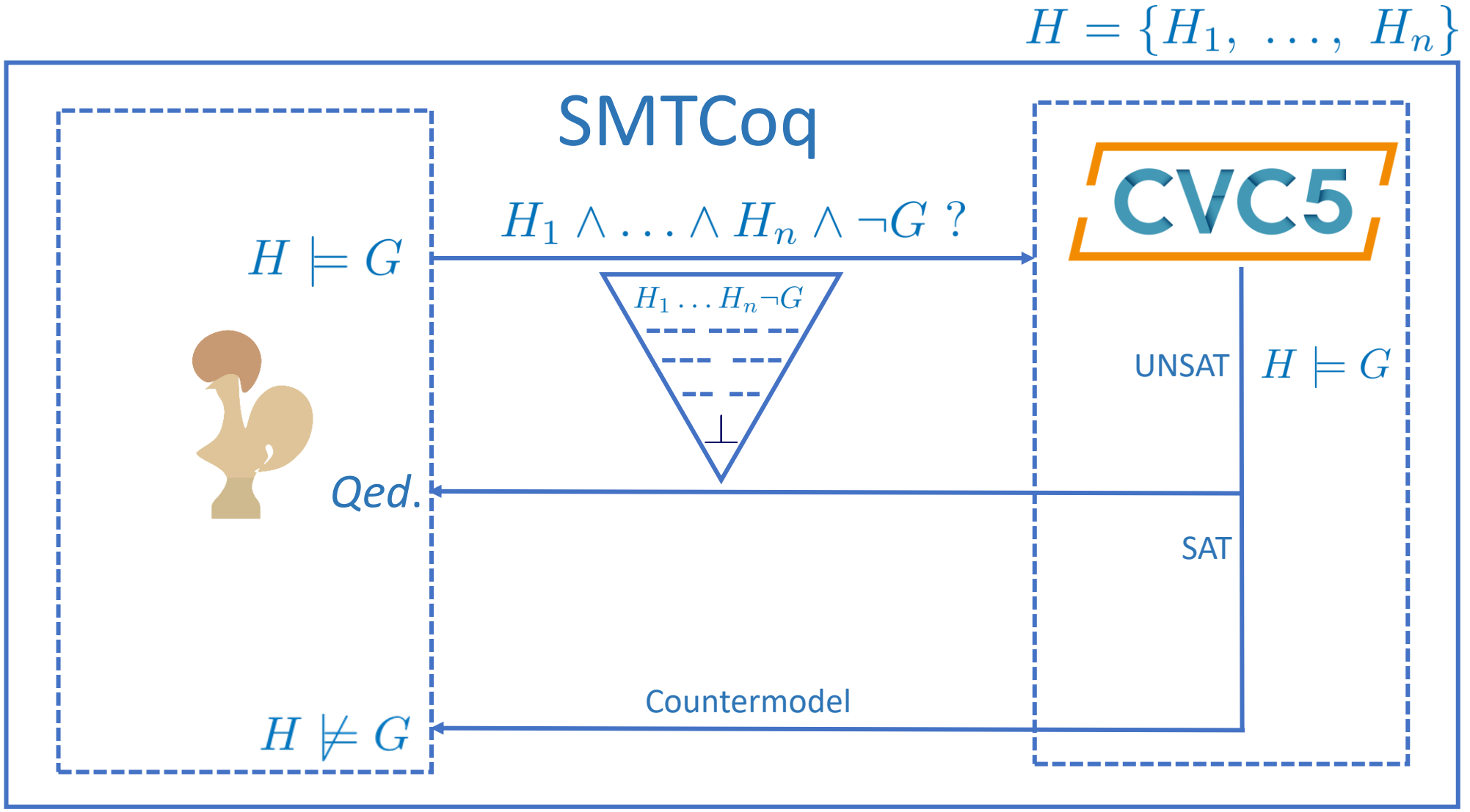
$\Rightarrow$ cvc5 supports these problems (for any background logic) via SyGuS

Synthesizing Conditions for Code Optimization

```
(set-option :produce-abducts true)
(set-logic QF_FP)
(declare-const rm RoundingMode)
(declare-const x Float32)
(get-abduct C (= (fp.add rm x (_ +zero 8 24)) x)
    ; Grammar
    ((B Bool) (F Float32))
    ((B Bool ((not (= F F))))
    (F Float32 (
        x (_ +zero 8 24) (_ -zero 8 24) (_ +oo 8 24) (_ -oo 8 24)
        (_ NaN 8 24)(fp.add rm F F) (fp.sub rm F F)
))))
```

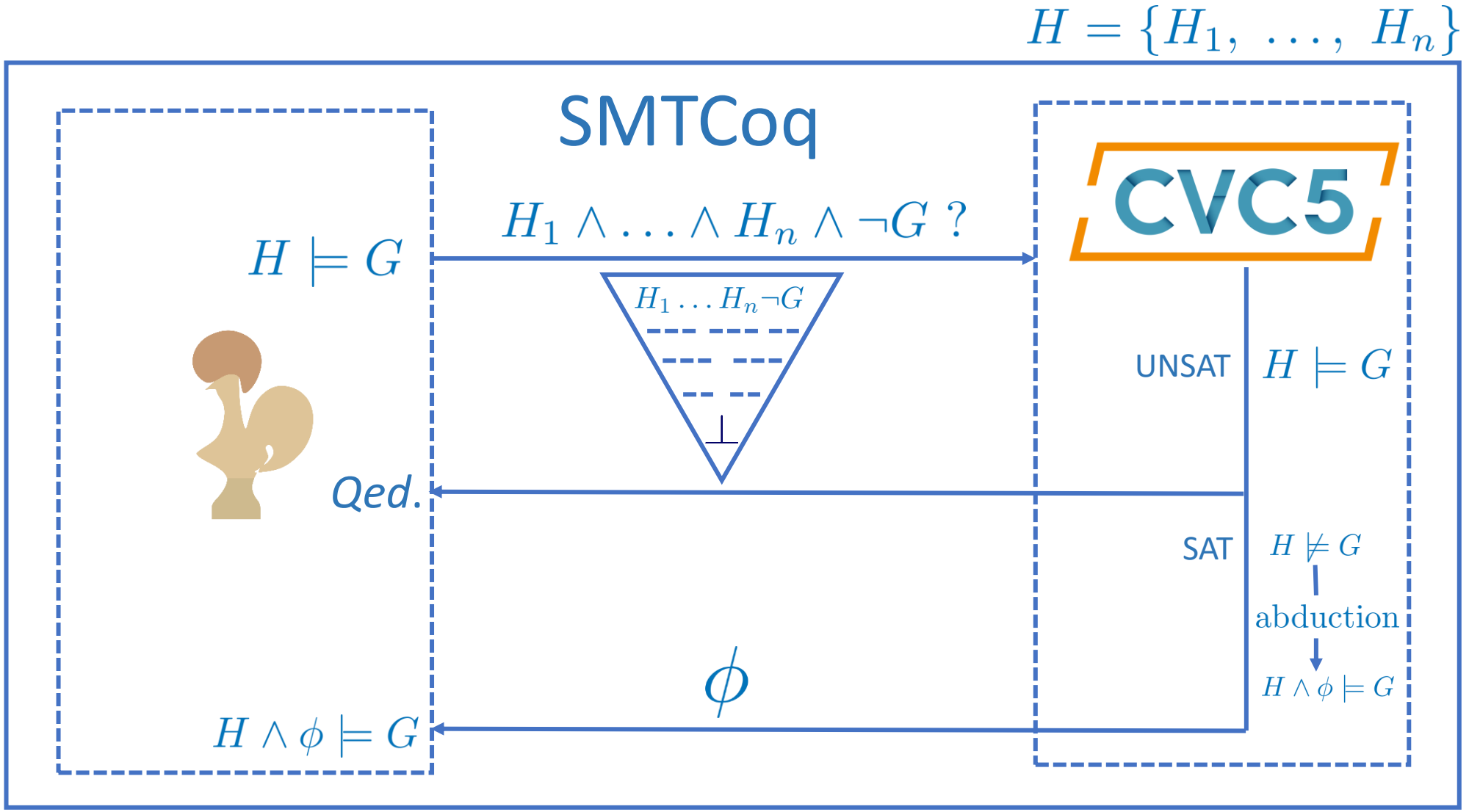
DEMO

Finding Subgoals in ITP via Abduction



Joint work with Arjun Viswanathan

Finding Subgoals in ITP via Abduction



Joint work with Arjun Viswanathan

Enabling Future Applications via **Oracles**

Oracles

- User-provided specification to black-box executables (oracles)

```
(declare-oracle-fun P (Int) Bool isPrime)
```

…where **`isPrime`** is a user-provided executable:

- Parses a string indicating integer value as input
  - txt file or alternatively via command line
- Prints a Bool value `true|false` on standard output

- The above command:
  - Allows `P` to be embedded in any constraint, similar to `declare-fun`
  - Associates the semantics of `P` to the given executable

SyGuS + Oracles

```
(set-logic LIA)
(synth-fun f ((x Int)) Int)
(declare-oracle-fun P (Int) Bool isPrime)
(constraint (and (P (f 6)) (not (P (f 8)))))
(check-synth)
```

isPrime
exe

SyGuS + Oracles

```
(set-logic LIA)
(synth-fun f ((x Int)) Int)
(declare-oracle-fun P (Int) Bool isPrime)
(constraint (and (P (f 6)) (not (P (f 8)))))
(check-synth)
```

```
(
(define-fun f v () Int (+ x 1))
)
```

DEMO

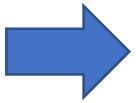
SMT + Oracles

```
(set-logic LIA)
(declare-fun c () Int)
(declare-fun d () Int)
(declare-oracle-fun P (Int) Bool isPrime)
(assert (and (< 1000 c) (< c d)))
(assert (and (P c) (P d)))
(check-sat)
```

- Can be used in combination with SMT queries as well

SMT + Oracles

```
(set-logic LIA)

(declare-fun c () Int)

(declare-fun d () Int)

(declare-oracle-fun P (Int) Bool isPrime)

(assert (and (< 1000 c) (< c d)))

(assert (and (P c) (P d)))

(check-sat)
```



```
(
(define-fun c () Int 1009)
(define-fun d () Int 1013)
)
```

DEMO

Solving Problems with Oracles in cvc5

- “Satisfiability and Synthesis Modulo Oracles” **[Polgreen/Reynolds/Seshia VMCAI 2022]**

- Defined two problems:
  - SMTO (satisfiability modulo theories and oracles)
  - SyMO (synthesis modulo oracles)

- Assumes deterministic (functional) oracles

- For each, defined a basic algorithm for solving them
  - Sound but incomplete

- Implementation is (partially) available in the main branch of cvc5

Potential Applications

- Implement semantics of operators in more expressive logics
  - For example, some temporal logic operators (MTL) require recursive definitions in SMT
  - Use oracle to implement the semantics as compiled code

- Black-box system testing (e.g. parsers)
  - Define a space of relevant inputs to a parser, via combination of datatypes/bitvectors
  - Oracle calls the system-under-test to check for errors

- Network policy synthesis
  - Network policies expressed as simple programs embedded as SMT datatypes
  - Oracle evaluates the effectiveness via random sampling + simulation, returns an approx. score

⇒ **Oracles eliminate the need for formalizing complex semantics in SMT**

⇒ **Oracles can lead to faster reasoning (compiled << interpreted)**

Conclusion

- Syntax-guided synthesis in an SMT solver
  - Highly optimized, actively developed
- Successfully used by cvc5 developers
  - SyGuS is a powerful tool to improve the SMT solver itself
- Many ongoing applications
  - Expressivity of SyGuS enhanced by oracles