Rewrites for SMT Solvers Using Syntax-Guided Enumeration

Andrew Reynolds

Haniel Barbosa

Cesare Tinelli

THE UNIVERSITY OF IOWA

Aina Niemetz

Andres Noetzli

Mathias Preiner

Clark Barrett

STANFORD

Rewrite Rules are Important for SMT Solving

- To develop an SMT theory solver for T, one must implement:
  1. A set of inference rules that decide if a set of constraints is T-sat/T-unsat
     - E.g. `x=y,y=z |= x=z, x=y |= f(x)=f(y), x≠x |= ⊥`
  2. A "rewriter" to put constraints in some normal form
     - E.g. `x+0→x, x-y→x+(-1*y), (x>x+y)→(0>y), x=x-2→⊥`
     - Can be seen as a set of "rewrite rules"

⇒Development of the latter is the focus of this talk

Rewrite Rules are Important for SMT Solving

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

- Powerful rewriter $\Leftrightarrow$ fast enumeration for syntax-guided synthesis

[Reynolds et al CAV 2015]

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
  - CVC4's BV rewriter ~3500 LOC
  - CVC4's string rewriter ~2800 LOC
  - CVC4's floating point rewriter ??? LOC
- Many special and subtle cases
  - Often need to see many examples to see proper generalization

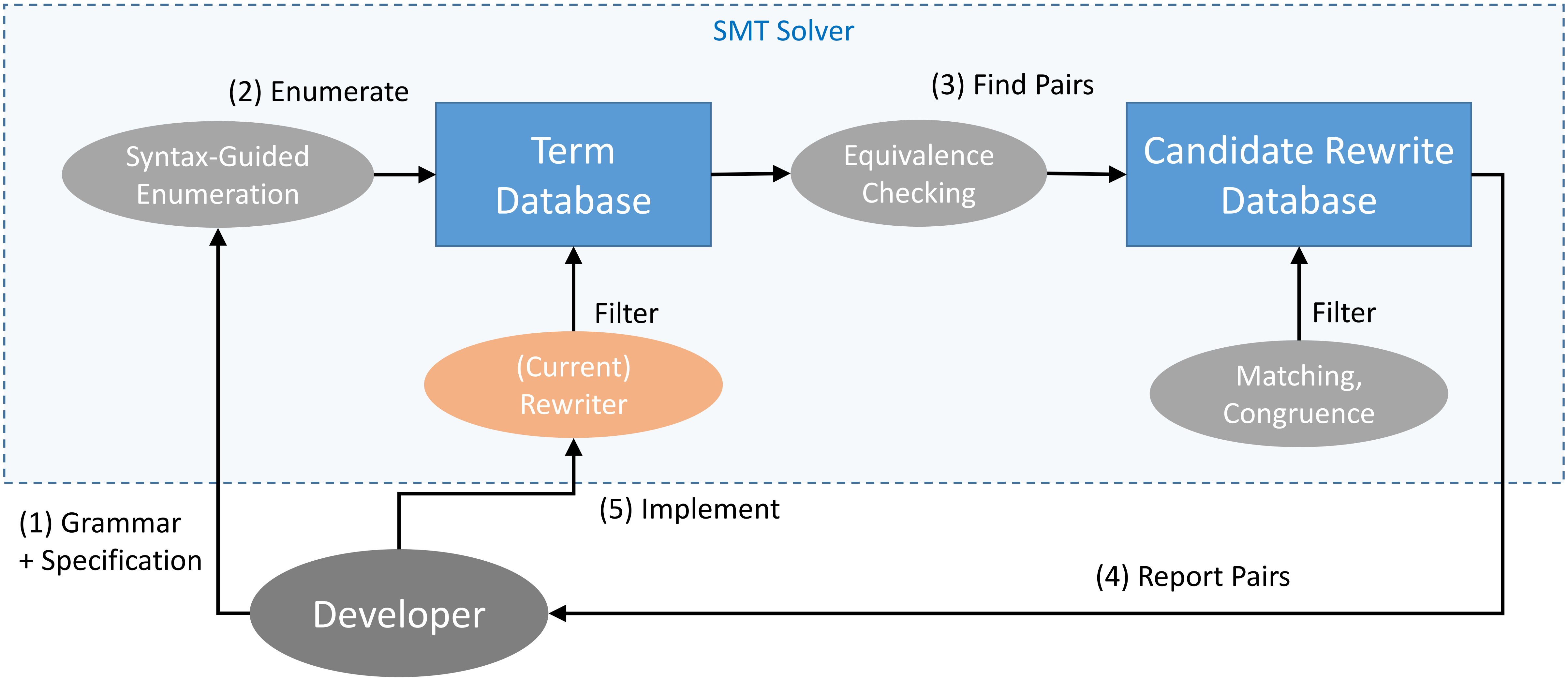
Goal of this Paper

- Use the SMT solver itself to assist the developer to implement the solver's rewriter

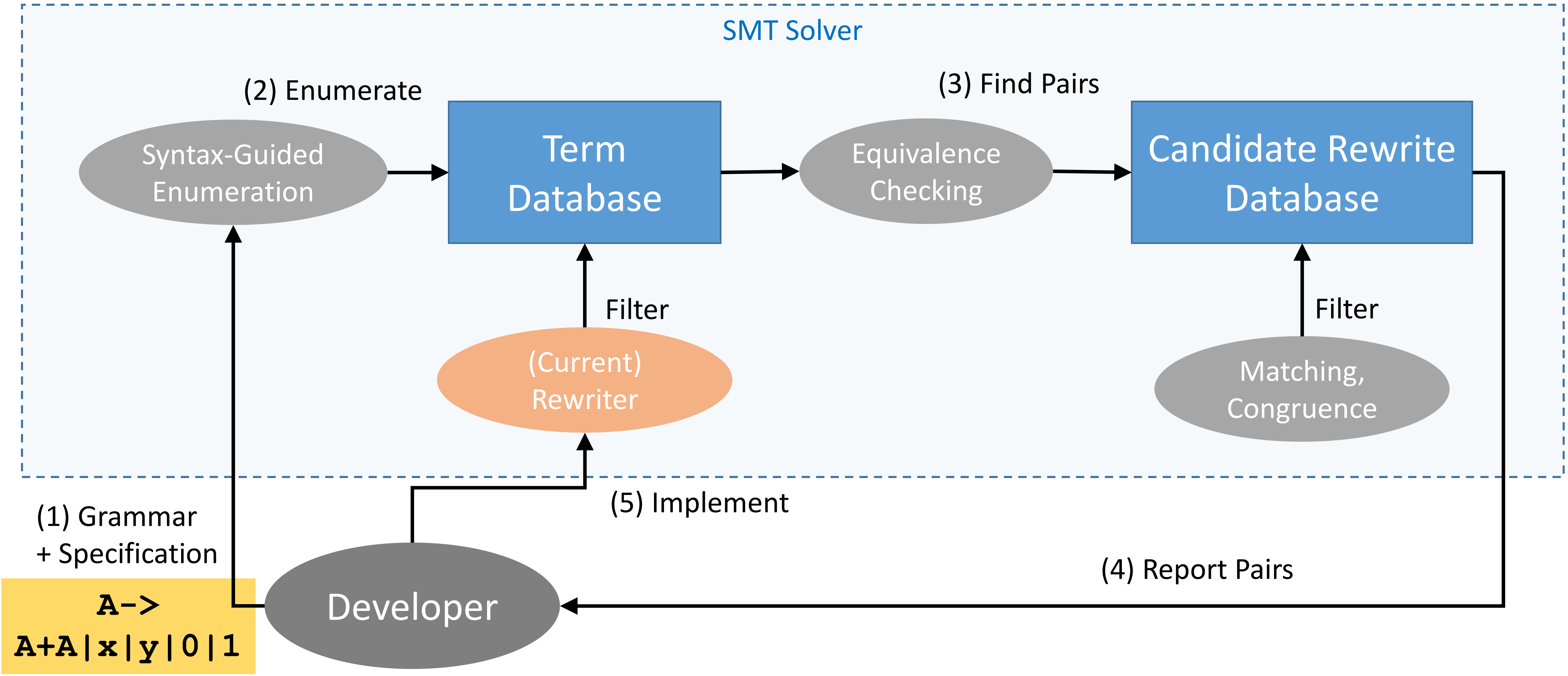
$\Rightarrow$ Increase confidence in the correctness of the rewriter

$\Rightarrow$ Increase productivity of the developer

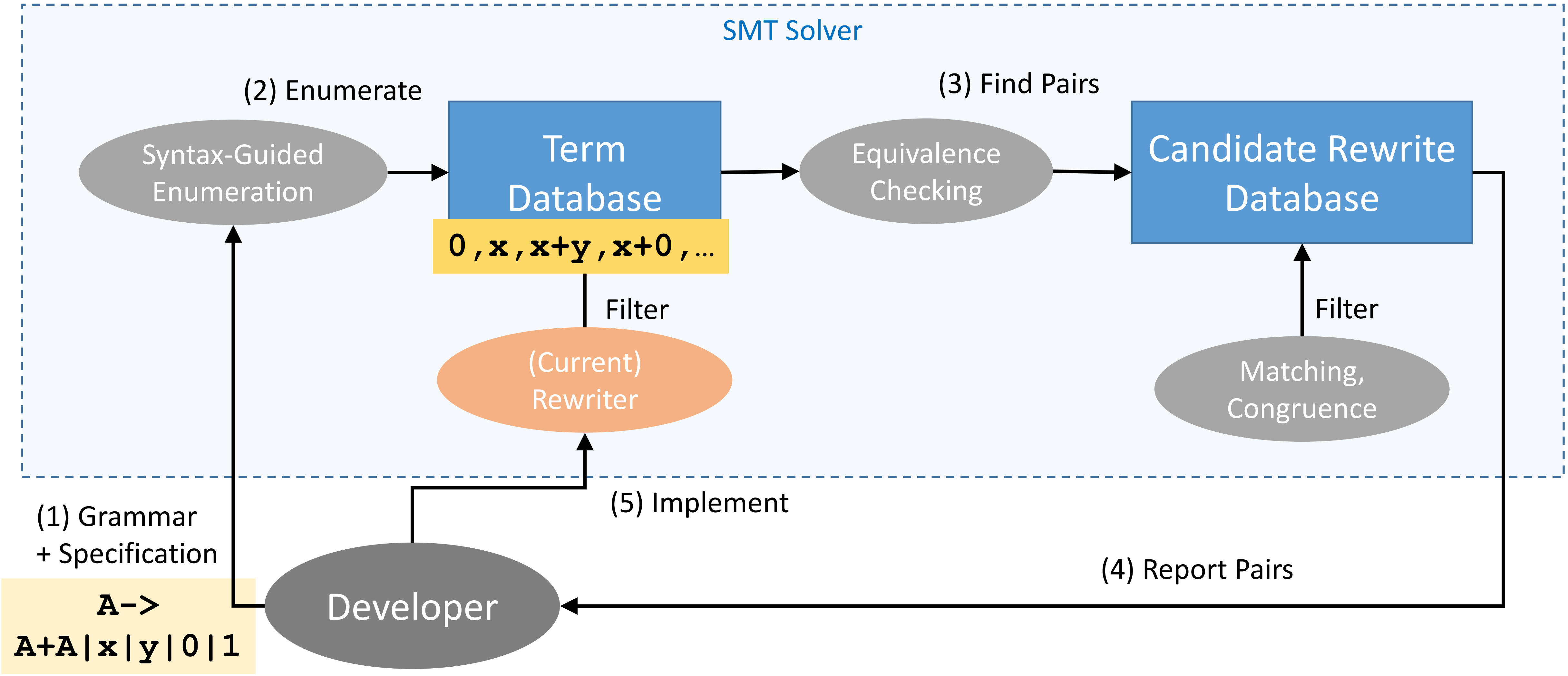
Workflow / Outline



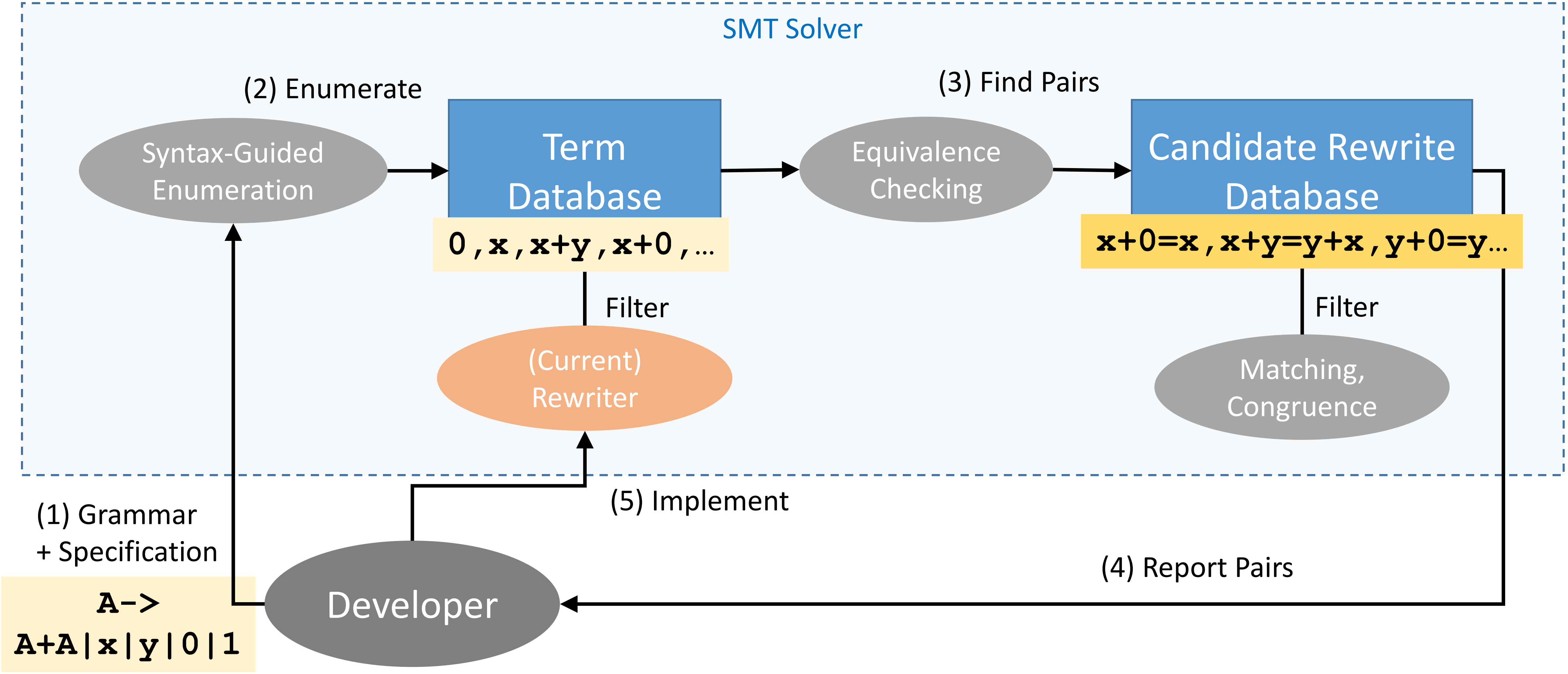
Workflow / Outline



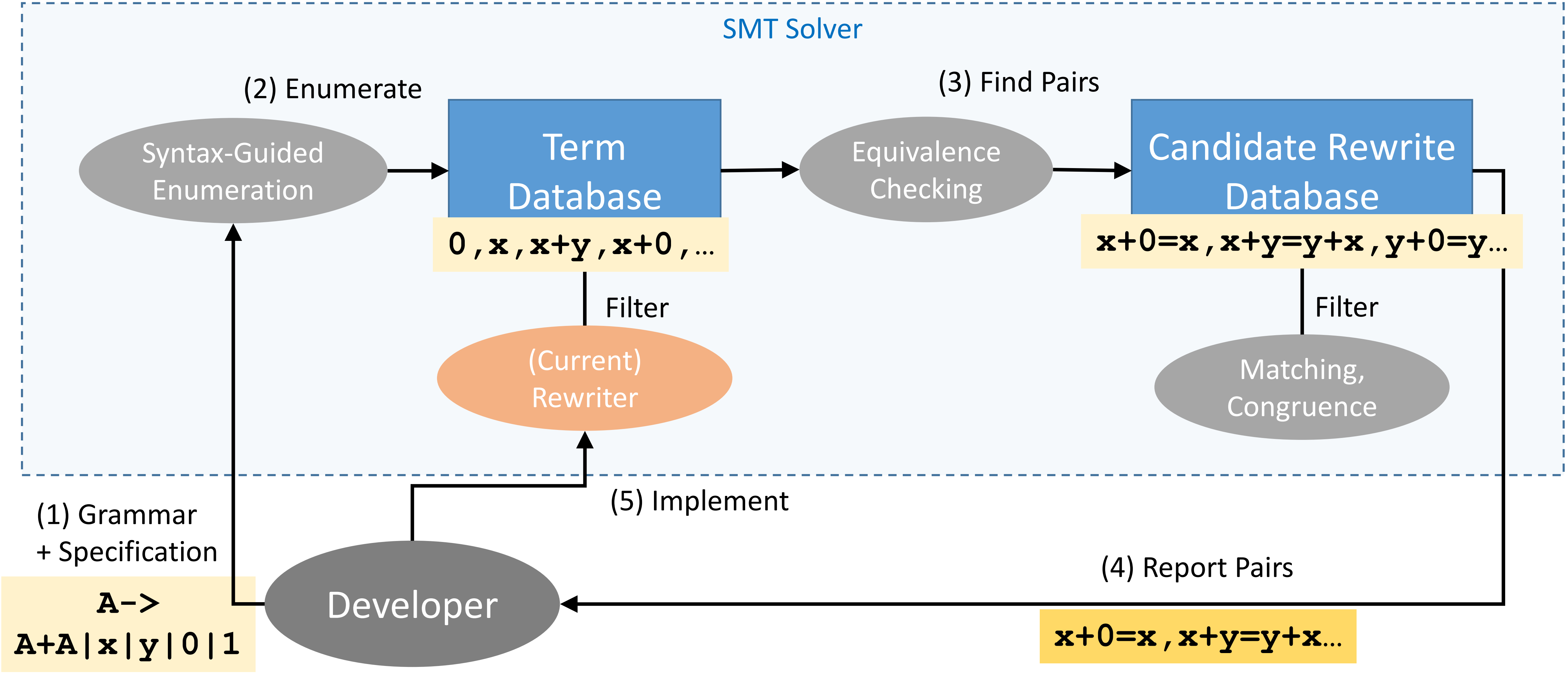
Workflow / Outline



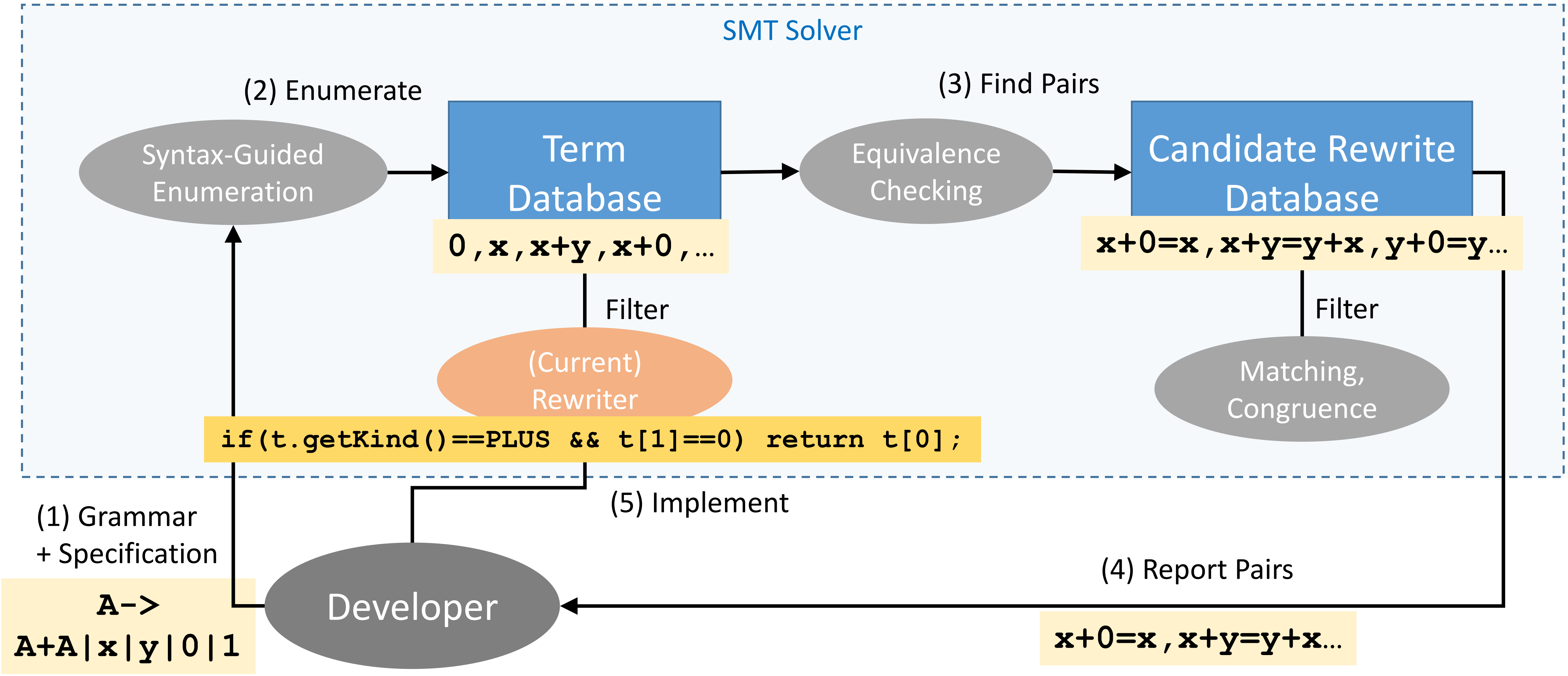
Workflow / Outline



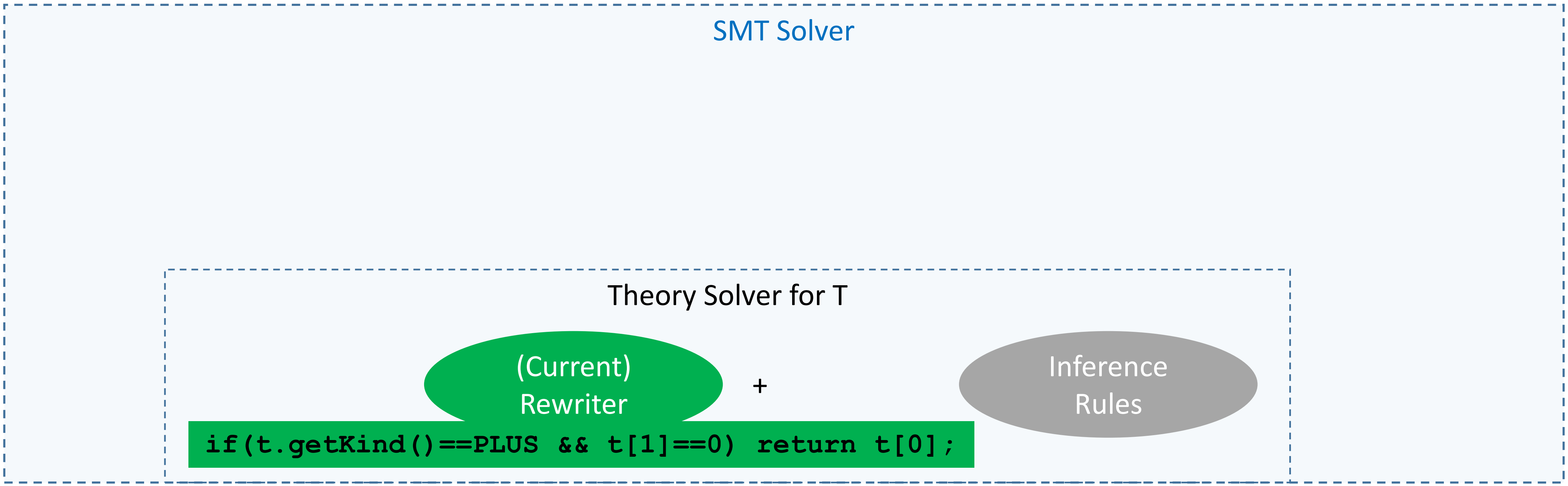
Workflow / Outline



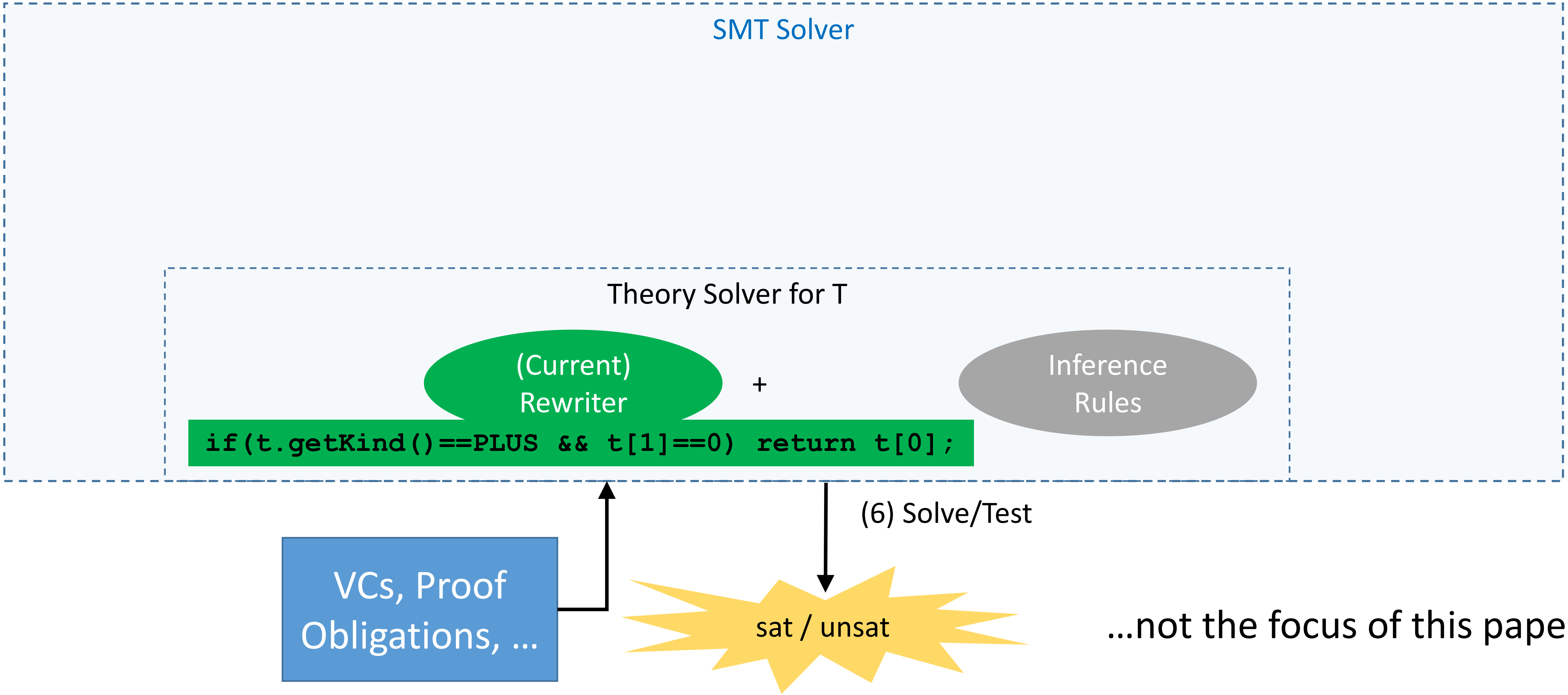
Workflow / Outline



Workflow / Outline



Workflow / Outline



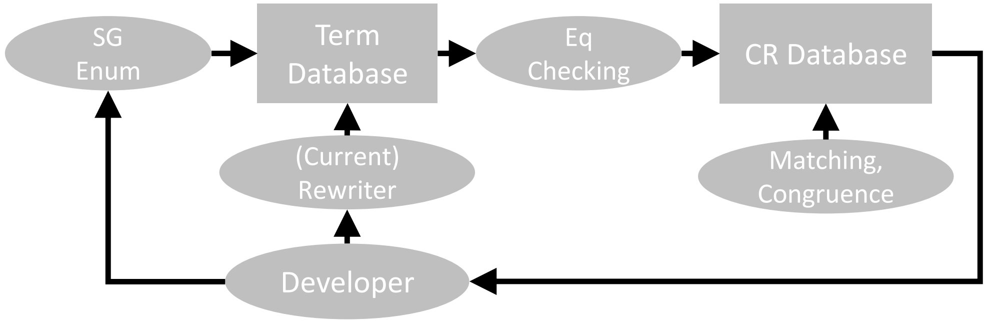
…not the focus of this paper

(Not) Goals of this Paper

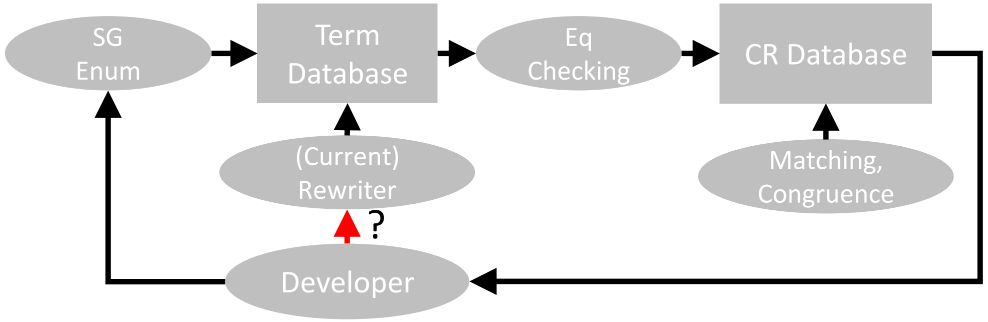
- We do not focus on automating:
  - Code generation of the implementation itself
  - Assessing good vs bad rewrites

$\Rightarrow$ For these, we rely on the creativity and ingenuity of the developer

…although these could be future work

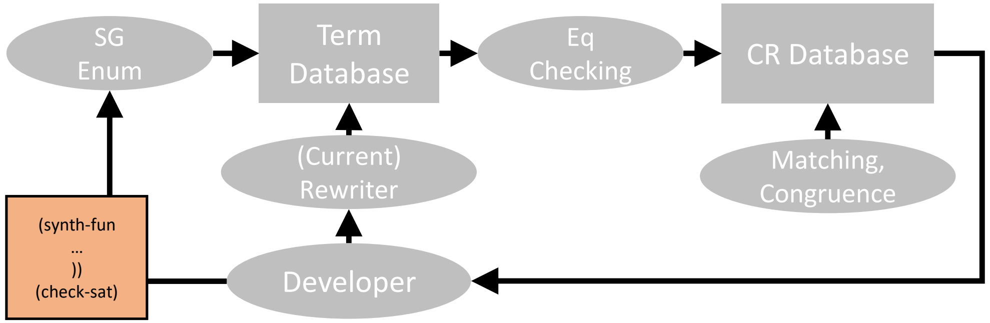

SG
Enum
Term
Database
Eq
Checking
CR Database
(Current)
Rewriter
Matching,
Congruence
Developer

Development of SMT Rewriter



- Developer has some idea of the set of terms that they are interested in developing new rewrites for:
  - “set of string terms built from concat, replace, and at most 2 variables”
  - “set of bit-vector terms with top-symbol multiplication”
  - “set of floating point predicates that include common interval abstractions”

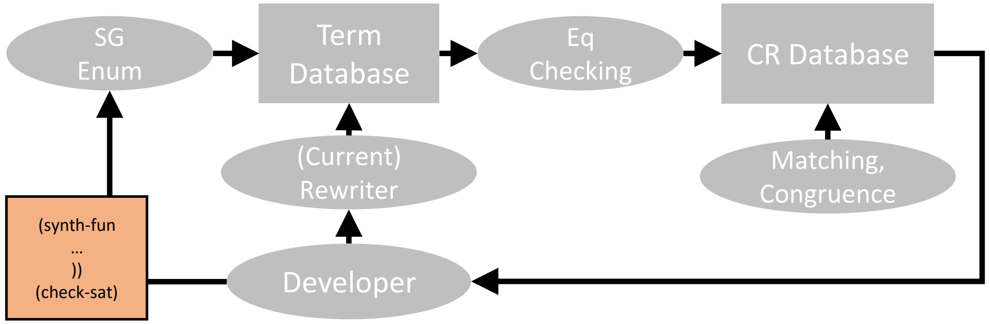
Grammar + Specification



- Use syntax-guided synthesis format *.sy for specify a class of terms:

```
(synth-fun f ((x Int) (y Int)) Int (
   (Start Int (A))
   (A Int ((+ A A) x y 0 1 (ite B A A))
   (B Int ((= A A) (>= A A) (not B) (and B B))
))
(constraint (= (f x y) (f y x)))
(check-synth)
```

Grammar + Specification



- Use syntax-guided synthesis format *.sy for specify a class of terms:

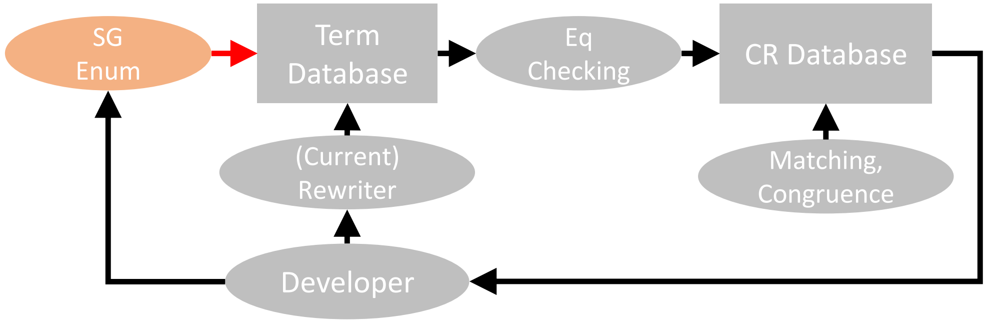
```
(synth-fun f ((x Int) (y Int)) Int (
   (Start Int (A))
   (A Int ((+ A A) x y 0 1 (ite B A A))
   (B Int ((= A A) (>= A A) (not B) (and B B))
))
(constraint (= (f x y) (f y x)))
(check-synth)
```

(1) Body of `f` is built from the **grammar**:

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

(2) `f` satisfies the **specification**:

$$\forall xy.f(x,y)=f(y,x)$$

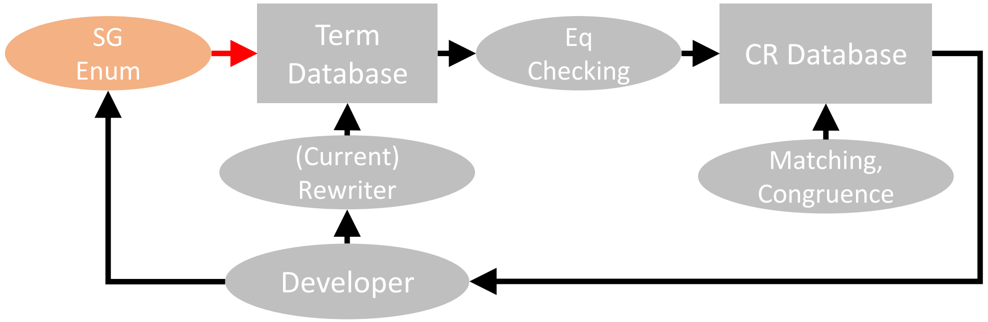


Syntax-Guided Enumeration

```
(synth-fun f ((x Int) (y Int)) Int (
  (Start Int (A))
  (A Int ((+ A A) x y 0 1 (ite B A A))
  (B Int ((= A A) (>= A A) (not B) (and B B))
))
(constraint (= (f x y) (f y x)))
(check-synth)
```

- Use enumerative syntax-guided search to generate ***multiple*** solutions to this conjecture
  - E.g. `0, 1, (+ x y), (+ y x), (+ 1 1),` …

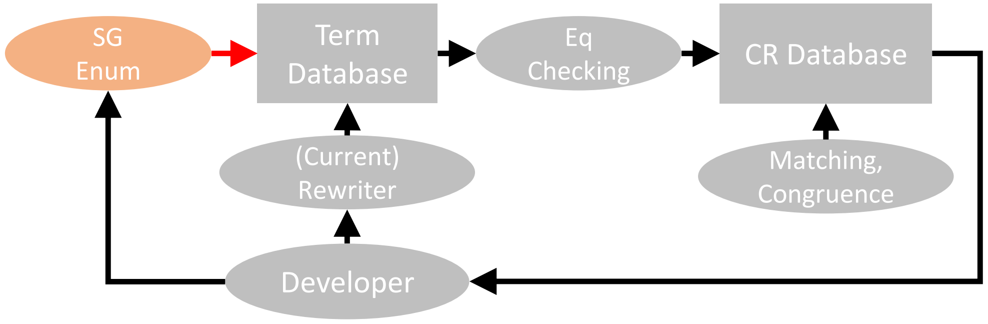
Syntax-Guided Enumeration




```
(synth-fun f ((x Int) (y Int)) Int (
  (Start Int (A))
  (A Int ((+ A A) x y 0 1 (ite B A A))
  (B Int ((= A A) (>= A A) (not B) (and B B))
))
(constraint (= (f x y) (f y x)))
(check-synth)
```

- Use enumerative syntax-guided search to generate ***multiple*** solutions to this conjecture
  - E.g. `0, 1, (+ x y), (+ y x), (+ 1 1), …`

⇒Number of **arguments** determines (maximum) variables per rewrite



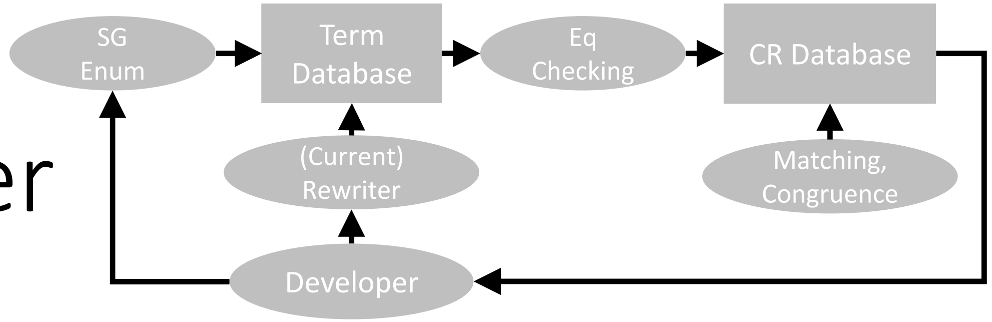
Syntax-Guided Enumeration

```
(synth-fun f ((x Int) (y Int)) Int (
  (Start Int (A))
  (A Int ((+ A A) x y 0 1 (ite B A A))
  (B Int ((= A A) (>= A A) (not B) (and B B))
))
(constraint (= (f x y) (f y x)))
(check-synth)
```

- Use enumerative syntax-guided search to generate ***multiple*** solutions to this conjecture
  - E.g. `0, 1, (+ x y), (+ y x), (+ 1 1), …`

⇒Number of arguments determines (maximum) variables per rewrite

⇒Specification can be used to filter out classes of terms

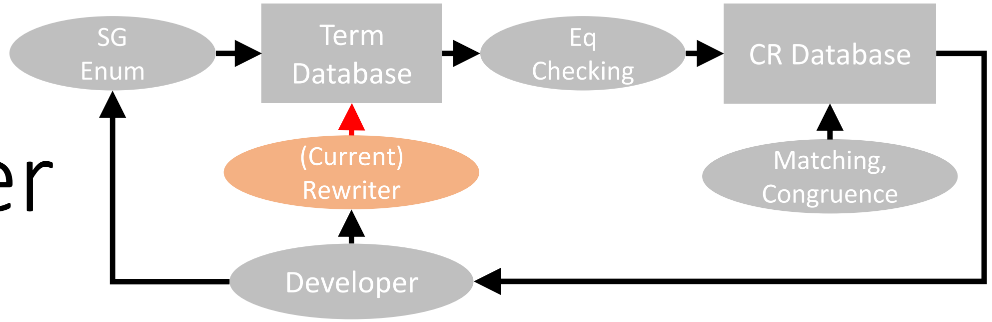


Filtering via the Current Rewriter

- When enumerating:

`0, 1, x, y, x+0, x+y, y+x, y+0, x+x, 1+1, …`

Filtering via the Current Rewriter

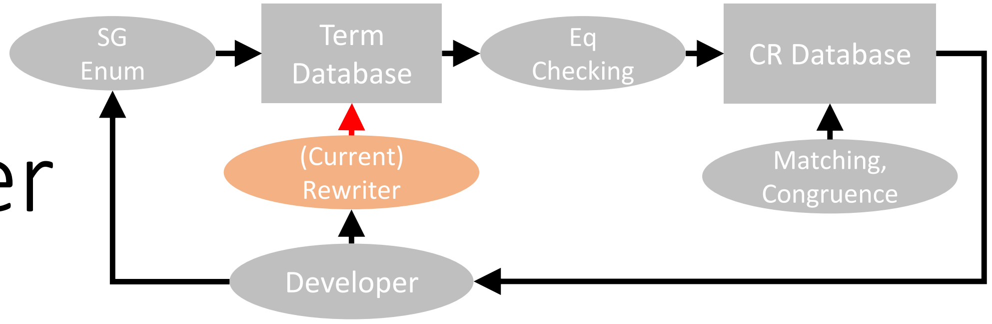


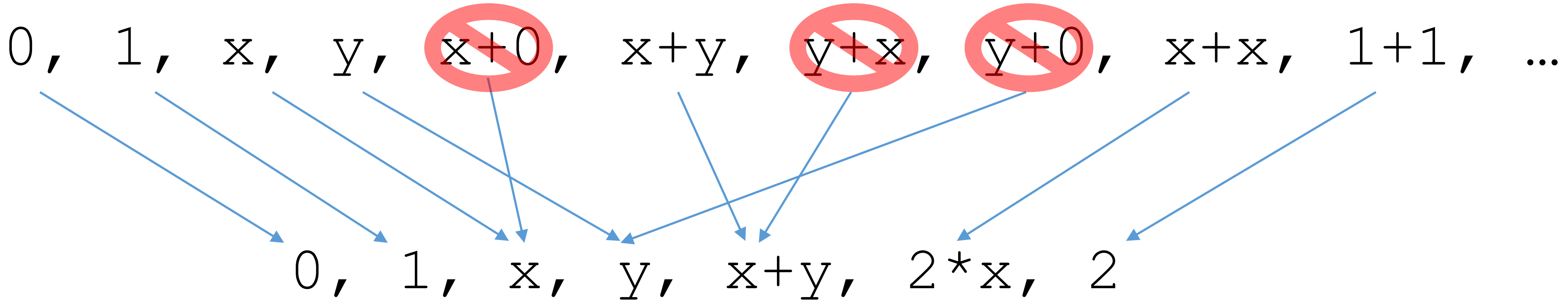
- When enumerating, map terms to their rewritten form, based on the current rewriter:

```
0, 1, x, y, x+0, x+y, y+x, y+0, x+x, 1+1, …
```

```
0, 1, x, y, x+y, 2*x, 2
```

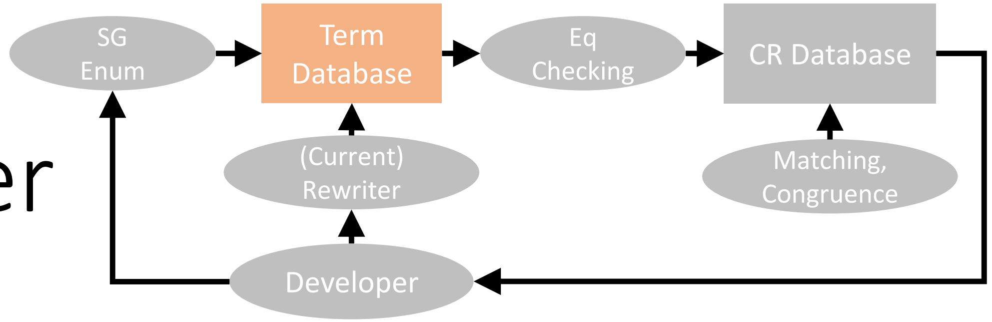
Filtering via the Current Rewriter





- Can discard all but one term for each set of terms that have the same rewritten form

  ⇒*This is what makes syntax-guided enumeration fast in practice*

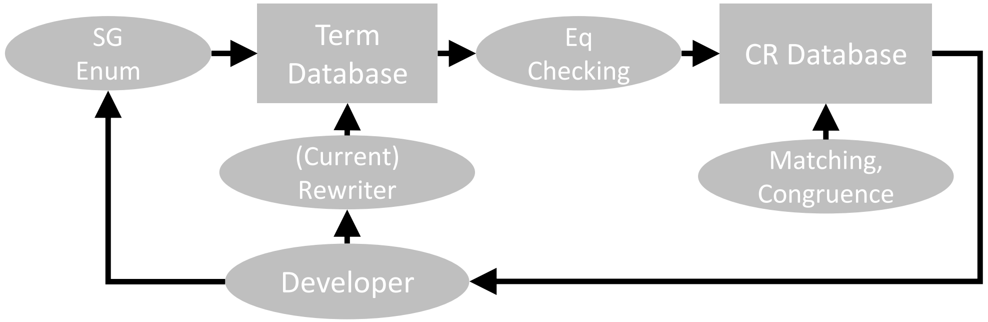


Filtering via the Current Rewriter

- Gives us a stream of terms that are unique up to the current rewriter:

`0, 1, x, y, x+y, x+x, 1+1, …`

“Term Database”

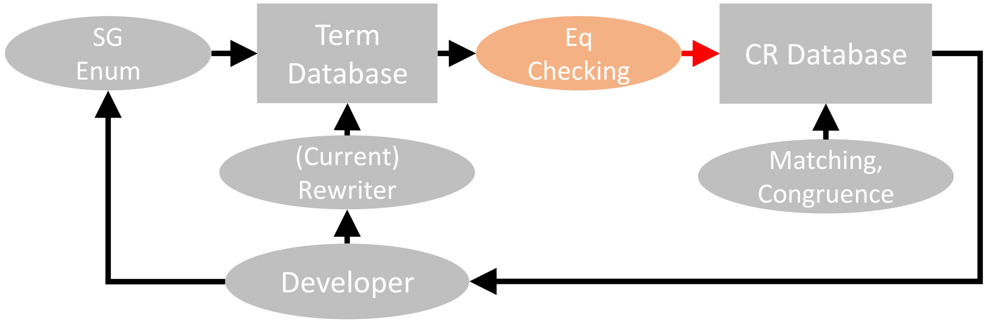


Equivalence Checking

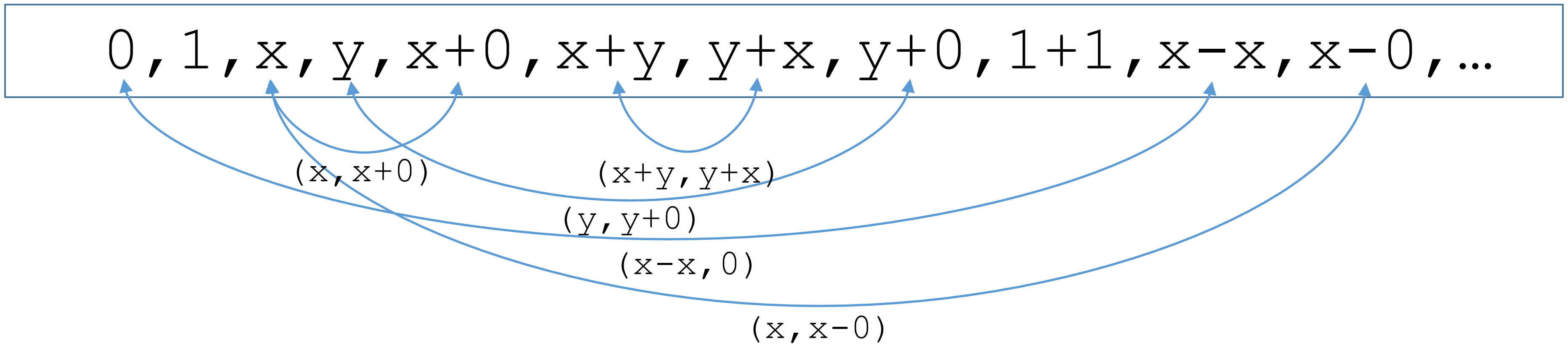
- **Given**: a set of terms, unique up to rewriting
- **Compute**: pairs of terms (s,t) such that s and t are (likely) T-equivalent

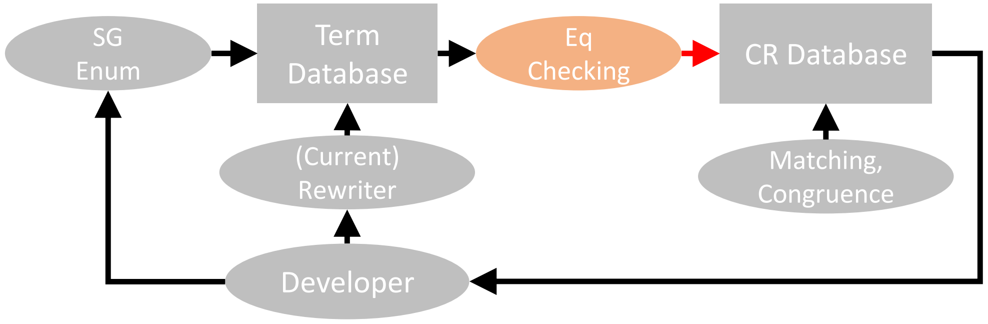
```
0,1,x,y,x+0,x+y,y+x,y+0,1+1,x-x,x-0,…
```

Equivalence Checking



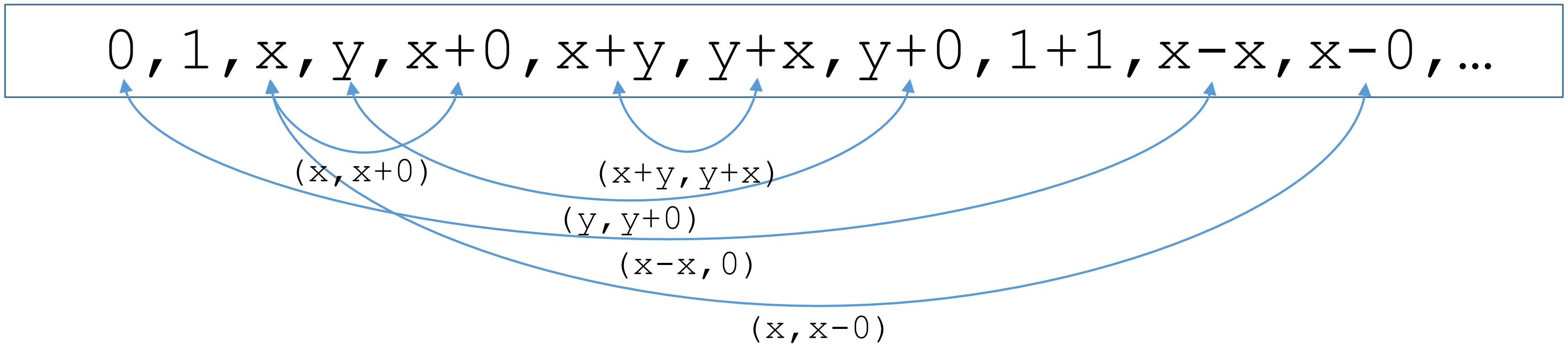
- **Given**: a set of terms, unique up to rewriting
- **Compute**: pairs of terms (s,t) such that s and t are (likely) T-equivalent





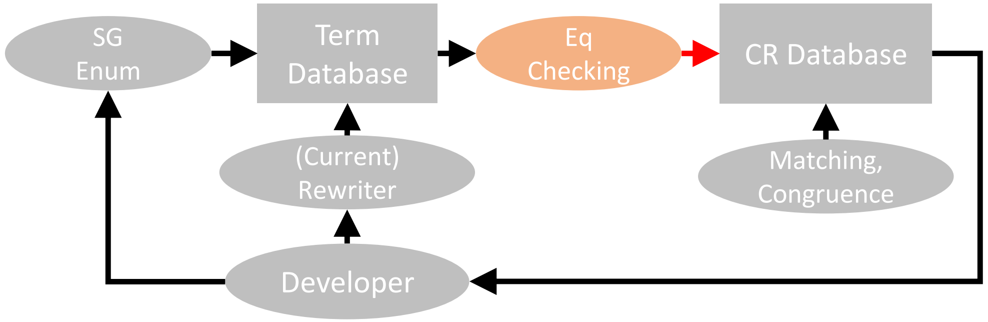
Equivalence Checking

- **Given**: a set of terms, unique up to rewriting
- **Compute**: pairs of terms (s,t) such that s and t are (likely) T-equivalent



- This gives us pairs of terms (`s`,`t`) such that:
  - `s` could be rewritten to `t` (or vice versa)
  - But our current rewriter does not already know this rewrite

Equivalence Checking



- To compute pairs (s,t), we check equivalence of s and t:
  - Via **Sampling**

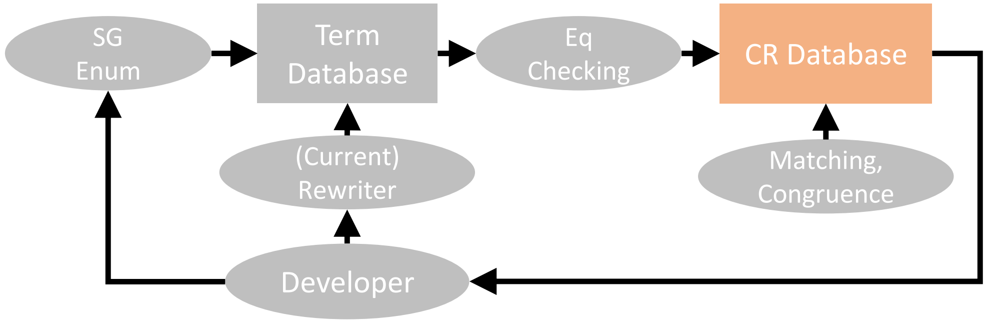
    s and t are equivalent if they evaluate to the same thing on N fixed sample points
    - Pro: can be very fast
    - Pro: feasible even if background theory (e.g. strings) is undecidable
    - Con: procedures false positives (s,t) where s and t are T-disequivalent

      $\Rightarrow$ …but can be made fairly precise using "grammar-based" sampling to find interesting points
  - Via **Exact Equivalence Checking**

    s and t are equivalent if the SMT solver says "unsat" for query $\exists x . s \neq t$
    - Pro: exact, i.e. (s,t) is a pair only if s and t are indeed T-equivalent
    - Con: not feasible and slower for some theories

      $\Rightarrow$ …but can be made efficient by caching counterexample points to failed queries

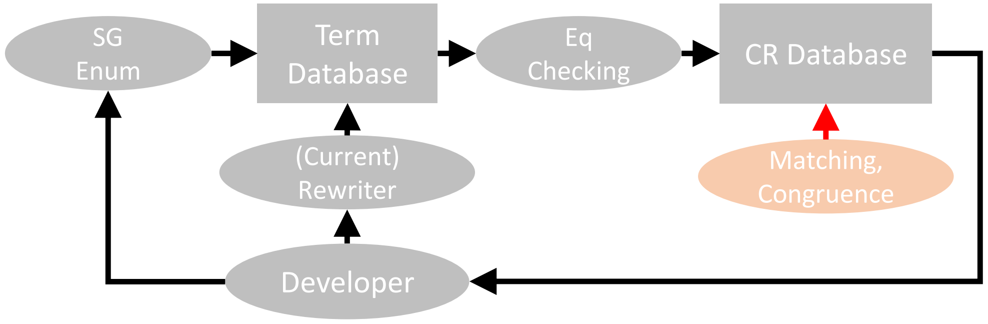
Rewrite Rule Filtering



- **Given:** set of rewrite pairs

```
(x,x+0),(x+y,y+x),(y,y+0),(x+0,0+x),(x,x−0),(x+y,(x+0)+y),…
```

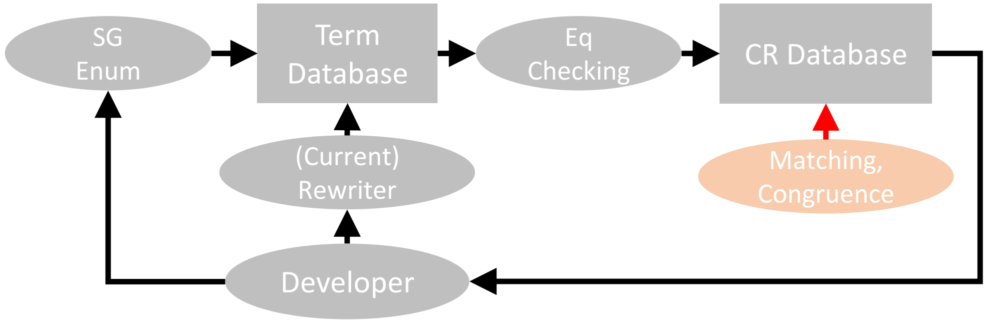
"Candidate Rewrite Database"



Rewrite Rule Filtering

- **Given:** set of rewrite pairs
- **Compute:** set of rewrite pairs that are not useful to the user

`(x,x+0), (x+y,y+x), (y,y+0), (x+0,0+x), (x,x-0), (x+y,(x+0)+y),…`

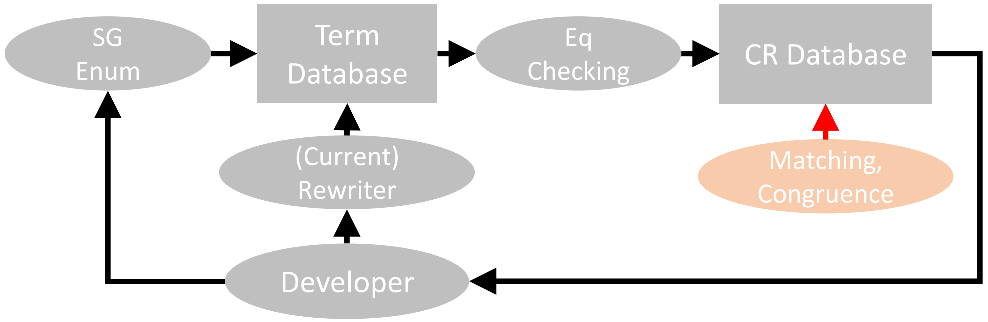


Rewrite Rule Filtering

- **Given:** set of rewrite pairs
- **Compute:** set of rewrite pairs that are not useful to the user

```
(x,x+0),(x+y,y+x),(y,y+0),(x+0,0+x),(x,x−0),(x+y,(x+0)+y),…
```

Rewrite Rule Filtering



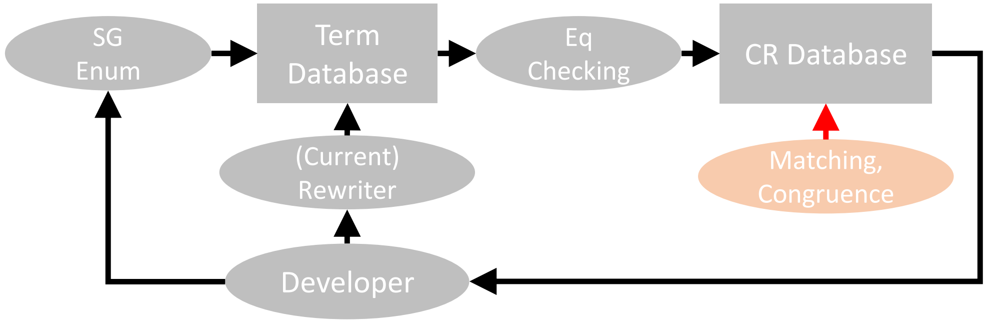
- **Given:** set of rewrite pairs
- **Compute:** set of rewrite pairs that are not useful to the user

`(x,x+0)`, `(x+y,y+x)`, ~~`(y,y+0)`~~, `(x+0,0+x)`, `(x,x−0)`, `(x+y,(x+0)+y)`, …

Alpha-equivalent

$\Rightarrow$ Can be efficiently enforced by fixing a variable ordering

Rewrite Rule Filtering

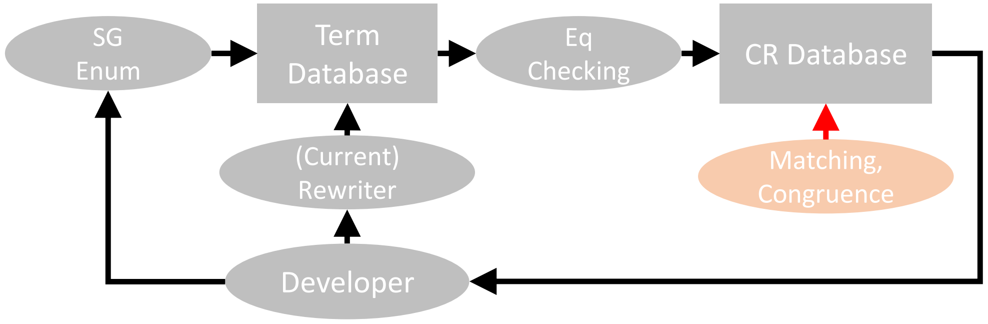


- **Given:** set of rewrite pairs
- **Compute:** set of rewrite pairs that are not useful to the user

(x,x+0), (x+y,y+x), ~~(x+0, 0+x)~~, (x,x−0), (x+y,(x+0)+y),…

Matchable

⇒ Use discrimination tree indexing



Rewrite Rule Filtering

- **Given:** set of rewrite pairs
- **Compute:** set of rewrite pairs that are not useful to the user

(x,x+0),(x+y,y+x), (x,x−0),(x+y,(x+0)+y),…

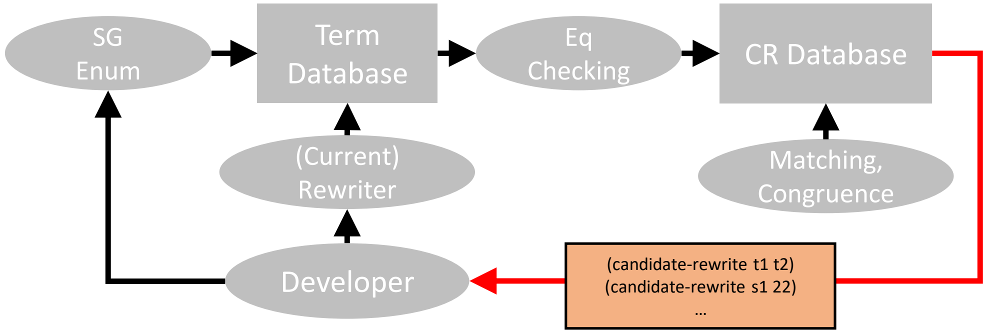
Consequence of Equality Reasoning

⇒ Maintain a congruence closure over pairs of terms

x=x+0 |= **x**+y=(**x+0**)+y

- Typically 30-40% rewrites are filtered, some grammars 60+%

Rewrite Rule Filtering

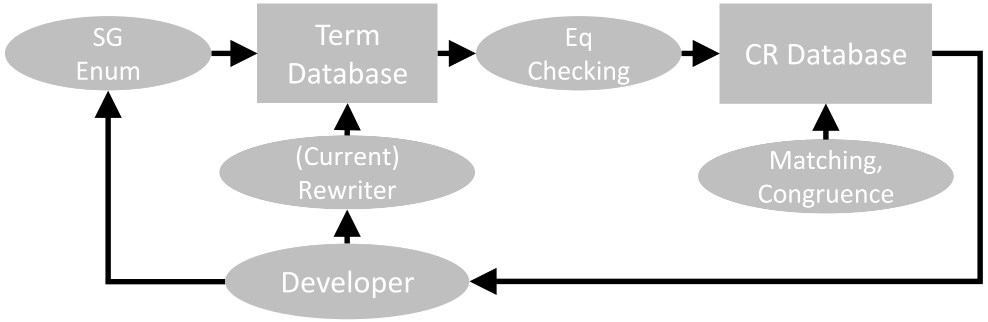


(x,x+0),(x+y,y+x), (x,x-0), ...

- This set of pairs is reported back to the user:

```
(candidate-rewrite (+ x 0) x)
(candidate-rewrite (+ x y) (+ y x))
(candidate-rewrite (- x 0) x)
...
```

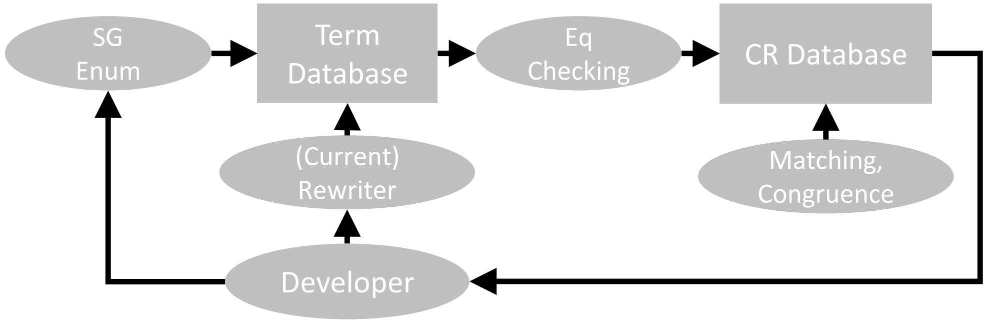
Preliminary Experience



- Implemented these features in the CVC4 SMT solver
  - Run on *.sy inputs using command line option `--sygus-rr-synth`
  - Many variants of this option are available

- Used workflow to generate rewrites for:
  - Strings
  - Bit-Vectors
  - Booleans
  - …Floating Points?

Preliminary Experience

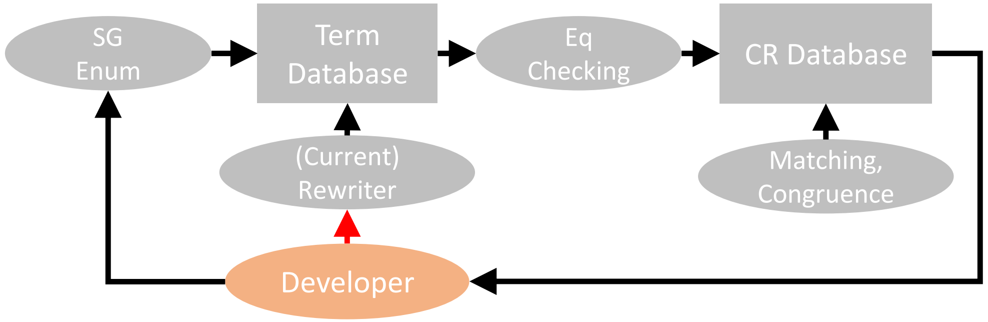



```
(synth-fun f
 ((x String) (y String) (z Int))
 String (
 (Start String (
  x y "A" "B" ""
  (str.++ Start Start)
  (str.replace Start Start Start)
  (str.at Start ie)
  (int.to.str ie)
  (str.substr Start ie ie)))
 (ie Int (
  0 1 z
  (+ ie ie)
  (- ie ie)
  (str.len Start)
  (str.to.int Start)
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

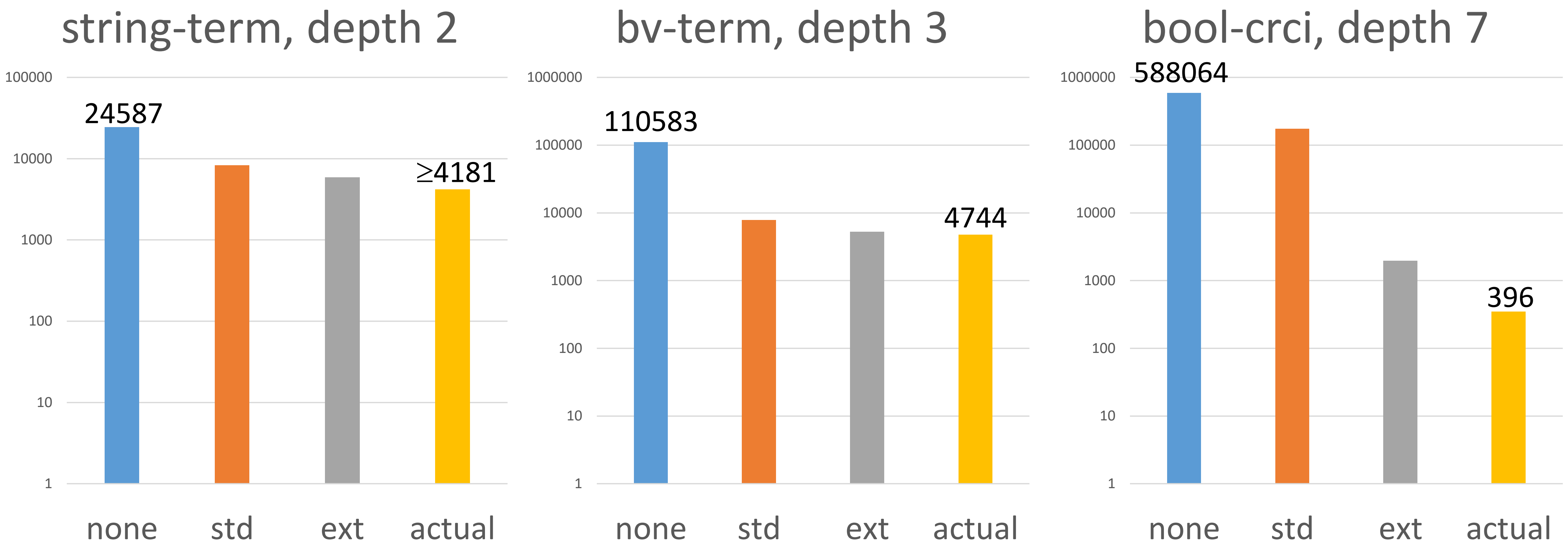
  `contains(x,x++"A")` → ⊥   `replace(x,x++y,y)` → `replace(x,x++y,"")`

- **Booleans**

  $A\wedge(A\vee B) \rightarrow A\wedge B$ $(A\vee C)\wedge(A\vee B) \rightarrow A\wedge(C\vee B)$

  $A=A\&B \rightarrow \neg A\vee B$ $(A\vee B)=(A\vee B\vee C) \rightarrow A\vee B\vee\neg C$

Statistics: CVC4’s Current Rewriter(s)



none: # terms from the grammar at given depth

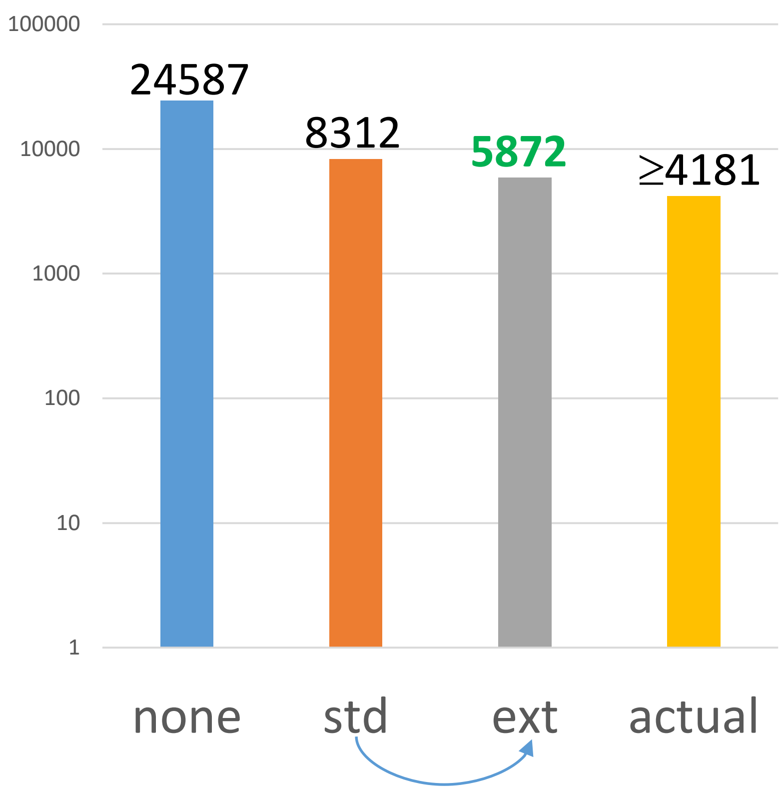
actual: # T-unique terms from grammar at given depth

std: CVC4 version 1.5’s rewriter (before this paper)

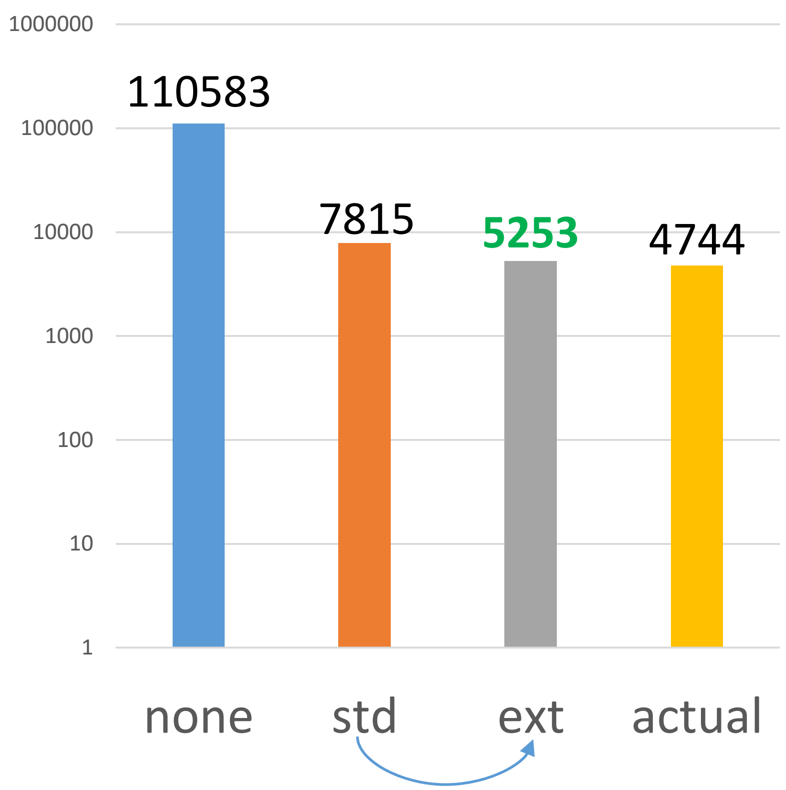
ext: CVC4’s aggressive rewriter (after this paper)

Statistics: CVC4’s Current Rewriter(s)

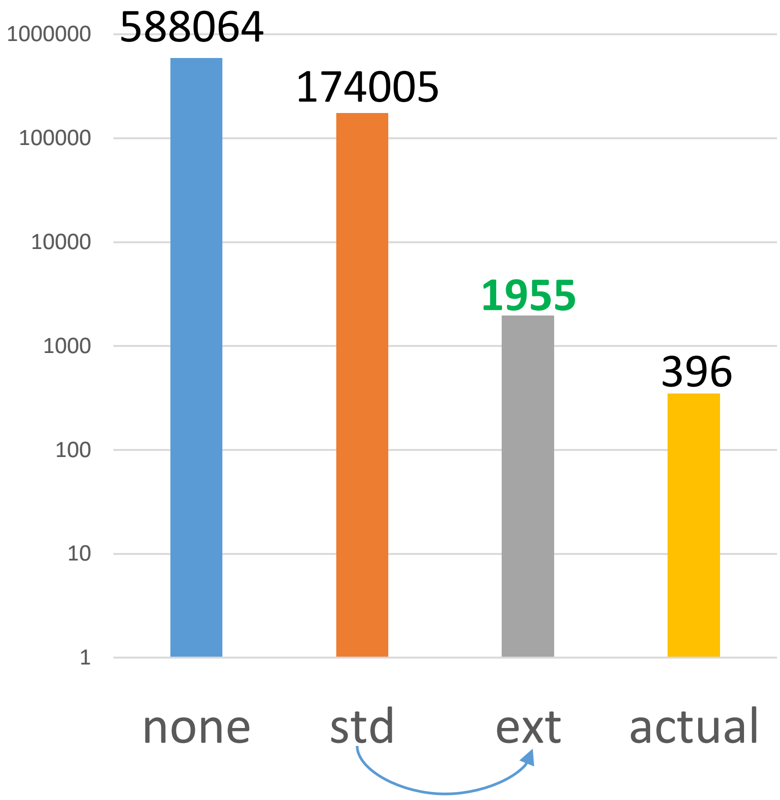
string-term, depth 2



bv-term, depth 3



bool-crci, depth 7

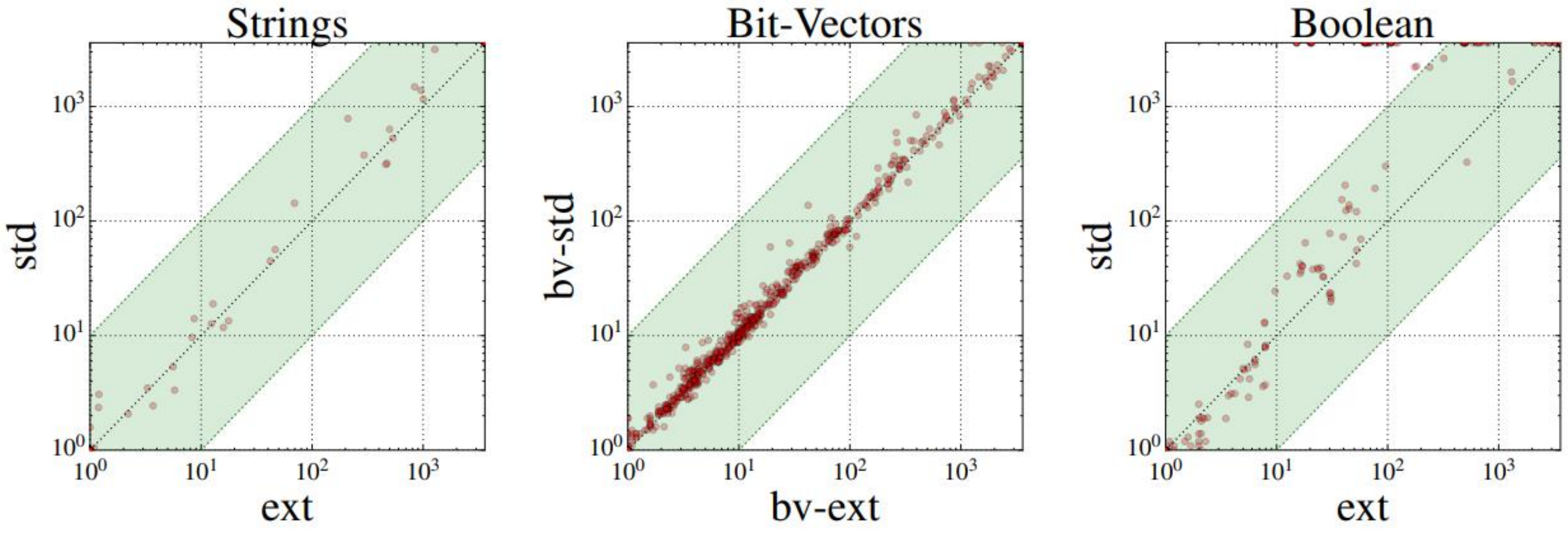


%redundant:	49.7%	**28.8%**	39.3%	**9.7%**	99.8%	**82.2%**
time to enumerate:	90.0	**60.8**	96.4	**55.4**	19128.8	**60.6**

Impact on Solving: SyGuS Conjectures

- For syntax-guided synthesis (sygus) queries, all rewrites are useful

  $\Rightarrow$ Speeds up enumeration times



Impact on Solving: SMT queries

- For general *.smt2 queries, some rewrites are good, some are bad
- Mixed performance using new rewrites (ext) vs original (def):
  - SMTLIB, QF_BV: good for unsat (+232,-158), bad for sat (+143,-236)
  - Quantified BV: overall improvement (+42,-15)
  - Strings (PyEx): good for unsat (+12,-1), bad for sat (+13,-94)

Improving Confidence in the Rewriter

- Can use sampling techniques to detect unsoundness in the rewriter
  - Run on *.sy inputs using command line option `--sygus-rr-verify`

```
(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
```

- Approximately 3.5x overhead

$\Rightarrow$ Has been critical for finding bugs in newly written rewriter code

Conclusions

- Infrastructure in CVC4 to increase productivity of rewrite rule developer
  - Used for past ~6 months to develop ~3000 LOC of rewrites
    - Strings, Bit-vectors, Booleans
  - Feedback loop:
    - More rewrites implemented → faster enumeration → more interesting rewrites found

- Has had impact on solving:
  - Significant improvements in syntax-guided synthesis *.sy problems
  - Mixed impact on *.smt2 problems

Future Work

- Further implementation on rewriters
  - Strings, bit-vectors, Booleans, …*floating points?*
- Optimizations to enumeration, equivalence checking
- Ways to infer grammars and interesting terms from *.smt2 inputs
  - Give me the rewrites that will help benchmark X
- Automate configurations of rewrite rules
  - Is this rewrite X good or bad (in context Y)?
- Interfaces to external users?
  - Users who want new rewrites in CVC4?
  - Developers of other rewriters?

Thanks for Listening!

- SMT Solver CVC4
  - Open source
  - Available at : http://cvc4.cs.stanford.edu/web/
- New options
  - `--sygus-rr-synth`: synthesize new rewrite rules from *.sy
  - `--sygus-rr-verify`: check the correctness of the current rewriter on *.sy
  - Configurable term filtering, equivalence checking, rule filtering

Demo?