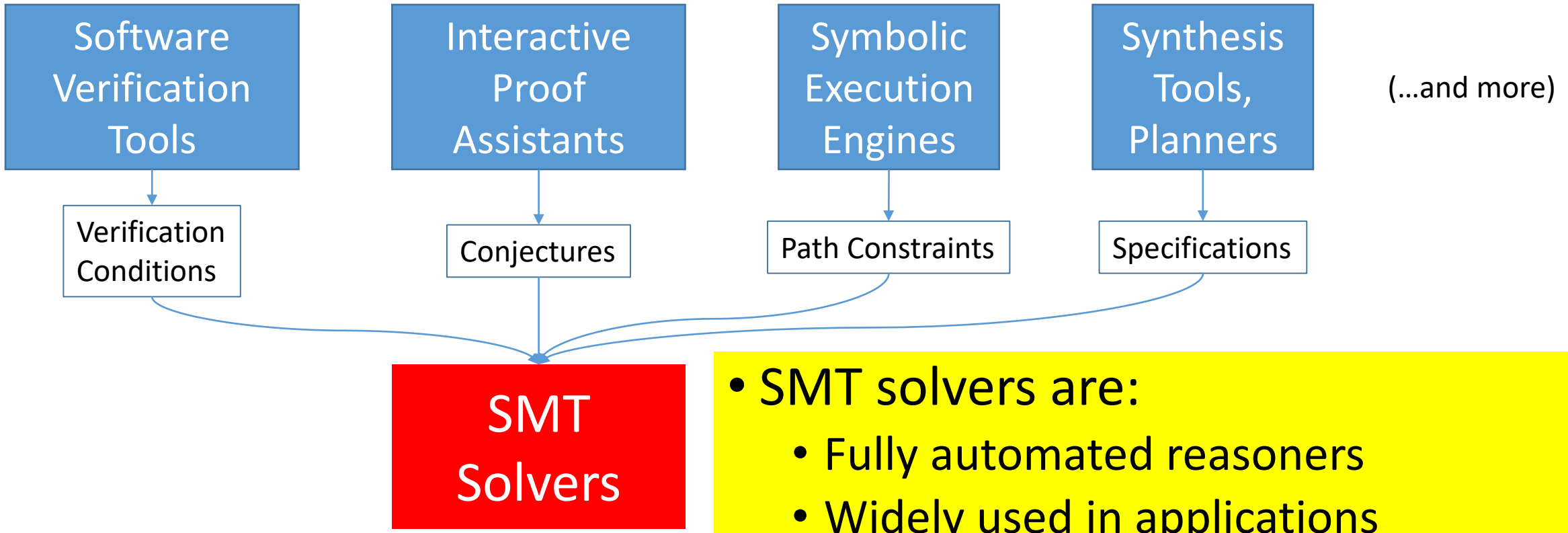
An Introduction to SMT Solvers and Their Applications (Part 2)

Andrew Reynolds

University of Iowa

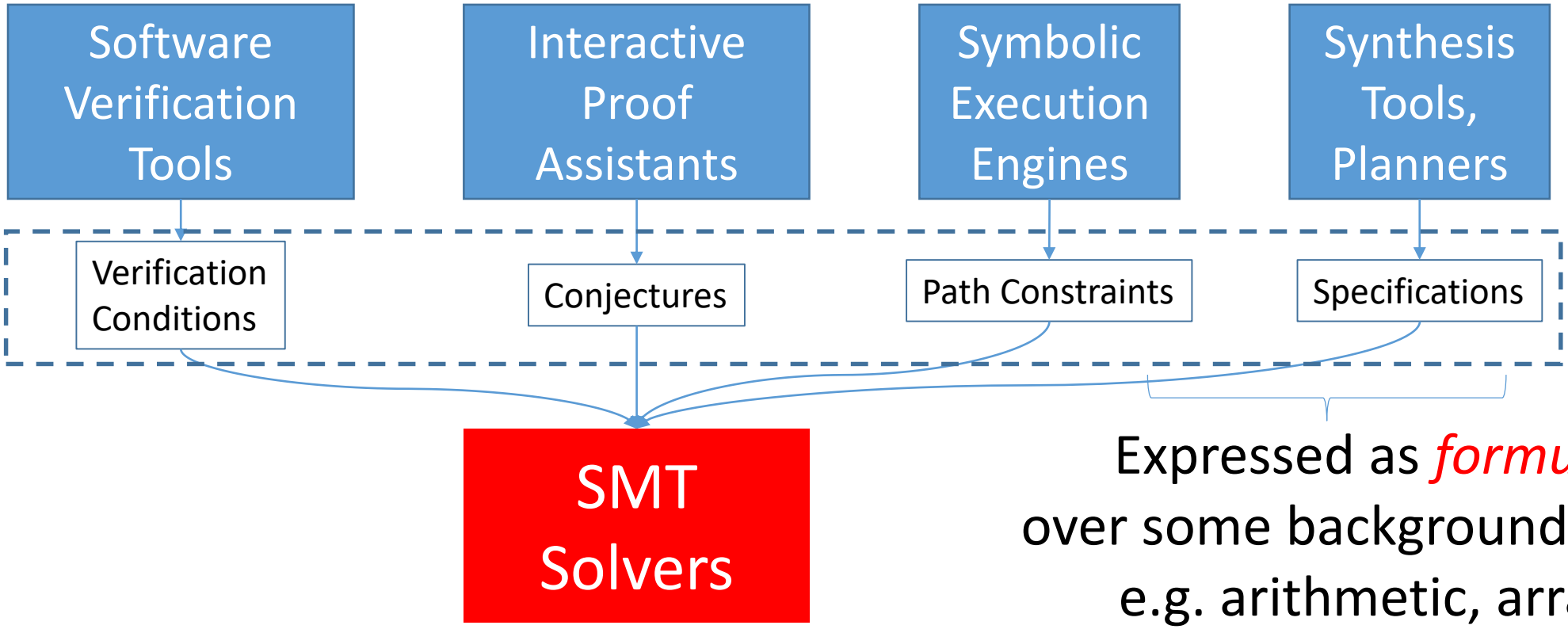
October 20, 2017

Satisfiability Modulo Theories (SMT) Solvers

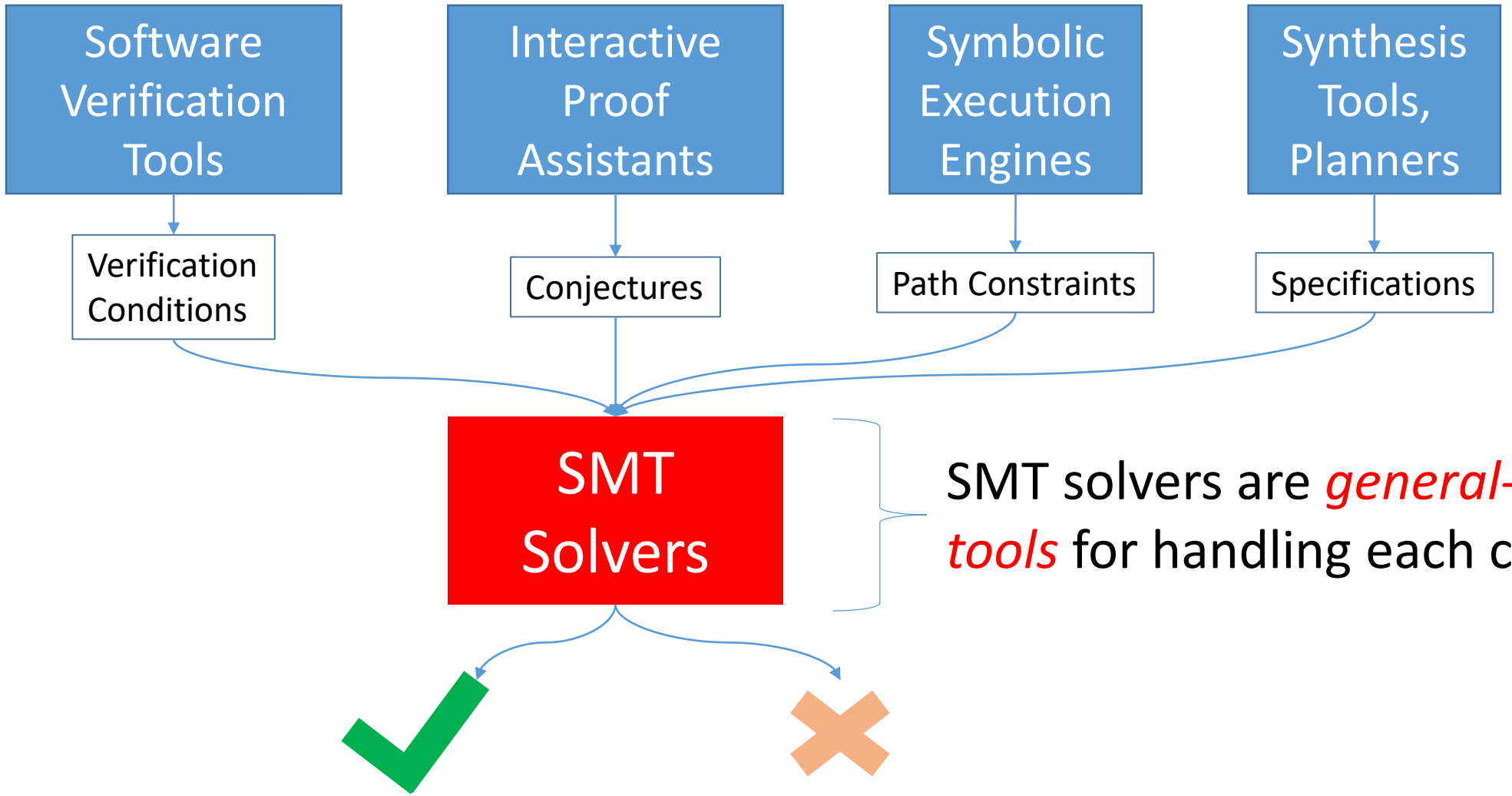


- SMT solvers are:
  - Fully automated reasoners
  - Widely used in applications

Satisfiability Modulo Theories (SMT) Solvers



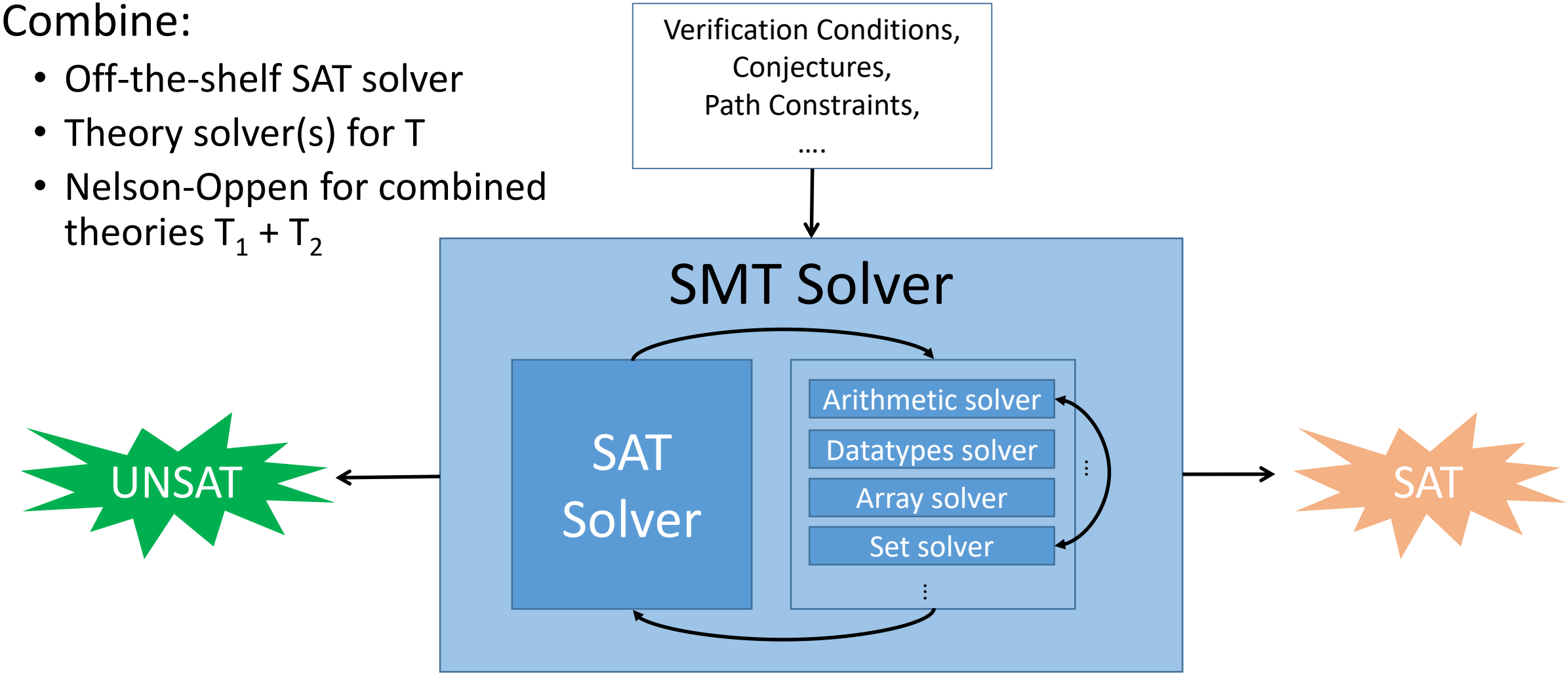
Satisfiability Modulo Theories (SMT) Solvers



SMT solvers are *general-purpose tools* for handling each case

In Part 1: DPLL(T)-Based SMT Solvers

- Combine:
  - Off-the-shelf SAT solver
  - Theory solver(s) for T
  - Nelson-Oppen for combined theories $T_1 + T_2$



In Part 2 : Decision Procedures

- *Decision procedures* in SMT solvers
  1. What is a decision procedure?
  2. What decision procedures are supported in SMT solvers?

  Focus on: theory of *unbounded strings + length*
     - Solving constraints in this theory
     - Example problems

Constraints Supported by SMT Solvers

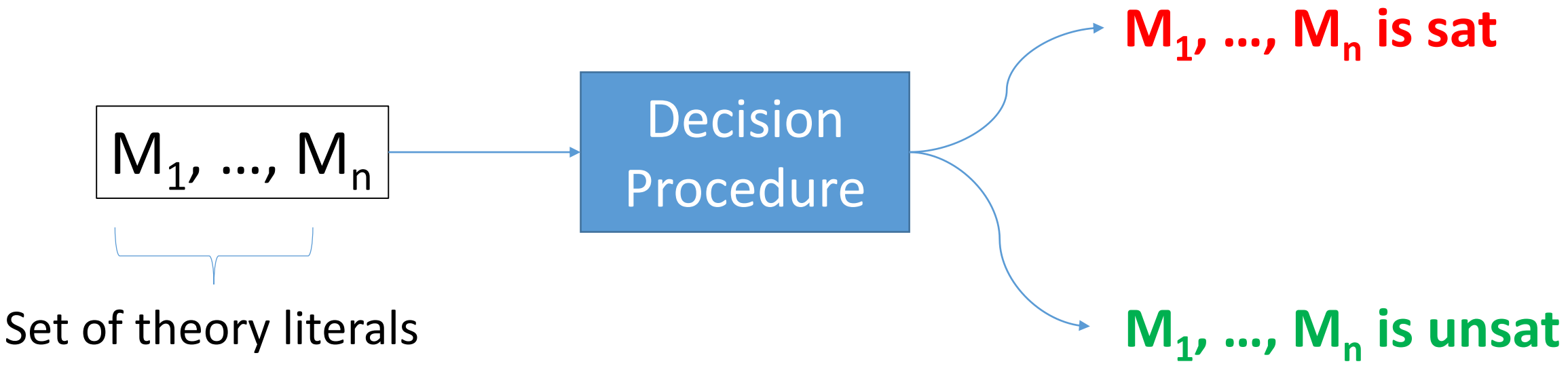
- SMT solvers support:
  - Arbitrary Boolean combinations of theory constraints
  - Examples of supported *theories*:
    - Uninterpreted functions: `f(a)=g(b,c)`
    - Linear real/integer arithmetic: `a≥b+2*c+3`
    - Arrays: `select(A,i)=select(store(A,i+1,3),i)`
    - Bit-vectors: `bvule(x,#xFF)`
    - Algebraic Datatypes: `x,y:List; tail(x)=cons(0,y)`
    - Unbounded Strings: `x,y:String; y=substr(x,0,len(x)-1)`
    - …

Constraints Supported by SMT Solvers

- SMT solvers support:
  - Arbitrary Boolean combinations of theory constraints
  - Examples of supported theories ⇒ decision procedures
    - Uninterpreted functions: ⇒ Congruence Closure **[Nieuwenhuis/Oliveras 2005]**
    - Linear real/integer arithmetic: ⇒ Simplex **[deMoura/Dutertre 2006]**
    - Arrays: ⇒ **[deMoura/Bjorner 2009]**
    - Bit-vectors: ⇒ Bitblasting, lazy approaches **[Bruttomesso et al 2007,Hadarean et al 2014]**
    - Algebraic Datatypes: ⇒ **[Barrett et al 2007,Reynolds/Blanchette 2015]**
    - Unbounded Strings: ⇒ **[Zheng et al 2013,Liang et al 2014, Abdulla et al 2014]**
    - …

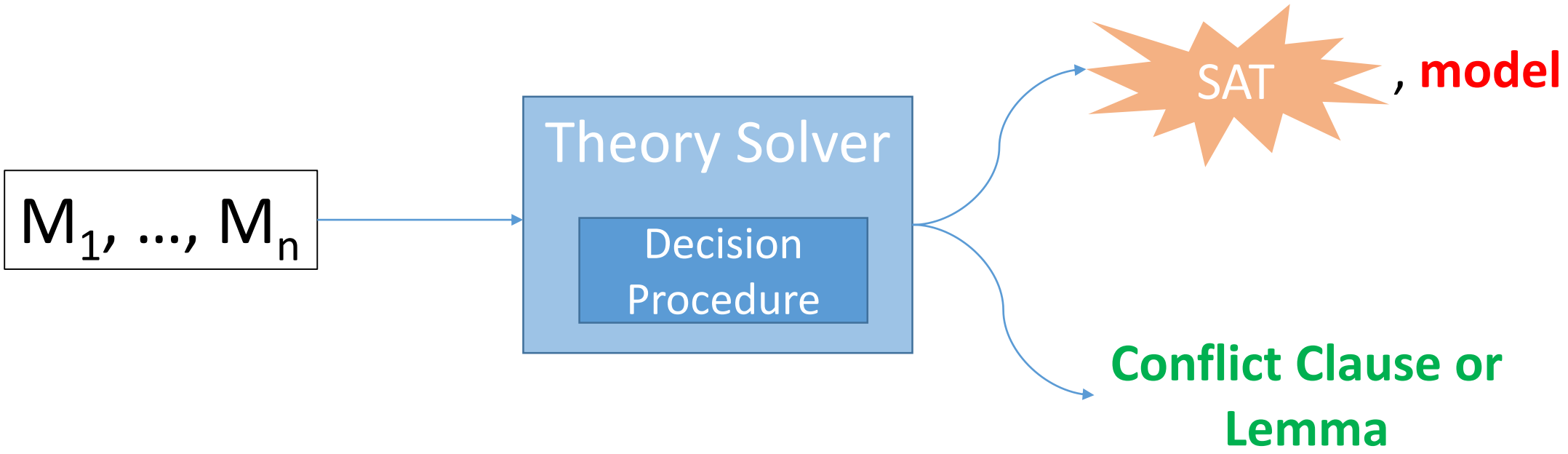
Decision Procedures in DPLL(T)

- A *decision procedure* has the following interface:



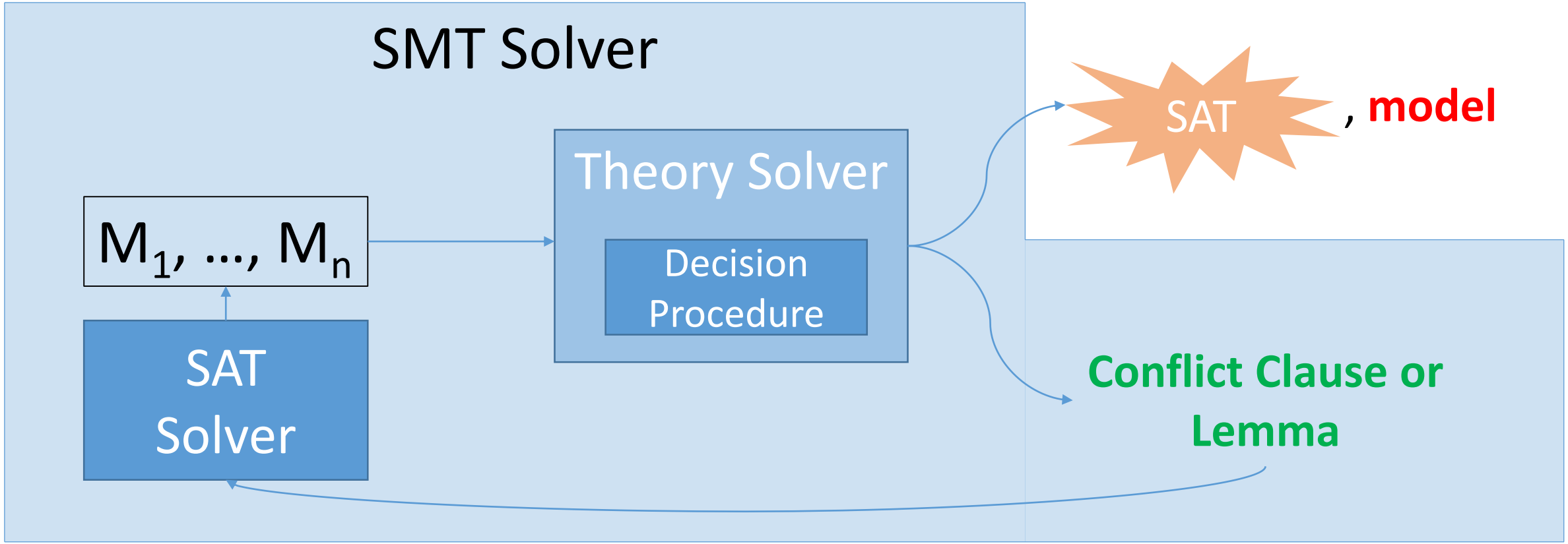
Decision Procedures in DPLL(T)

- SMT solvers use *theory solvers* that implement decision procedures:



Decision Procedures in DPLL(T)

- Theory solvers are integrated in the *DPLL(T)* loop:



DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
  2. $\{ l_1, \ldots, l_n \} \subseteq$ M is T-unsatisfiable
  3. Don’t know

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { x $\rightarrow$ 2, y $\rightarrow$ 3, z $\rightarrow$ -3} for { x<y, y+z=0 }

     $\Rightarrow$ Should be *solution-sound*
     - Answers “M is T-satisfiable” only if M is T-satisfiable
  2. { l_1, …, l_n } $\subseteq$ M is T-unsatisfiable
  3. Don’t know

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { $x \rightarrow 2$, $y \rightarrow 3$, $z \rightarrow -3$} for { $x<y$, $y+z=0$ }

     $\Rightarrow$ Should be *solution-sound*
       - Answers "M is T-satisfiable" only if M is T-satisfiable
  2. $\{ l_1, \ldots, l_n \} \subseteq$ M is T-unsatisfiable
     - Return *conflict* clause, e.g. ($x<0 \vee \neg x=3$) for { $\neg x<0$, $x=3$ }

     $\Rightarrow$ Should be *refutation-sound*
       - Answers "$\{ l_1, \ldots, l_n \}$ is T-unsatisfiable" only if $\{ l_1, \ldots, l_n \}$ is T-unsatisfiable
  3. Don't know

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { x $\rightarrow$ 2, y $\rightarrow$ 3, z $\rightarrow$ -3} for { x<y, y+z=0 }

     $\Rightarrow$ Should be *solution-sound*
       - Answers "M is T-satisfiable" only if M is T-satisfiable
  2. { l_1, …, l_n } $\subseteq$ M is T-unsatisfiable
     - Return *conflict* clause, e.g. (x<0 $\vee \neg$ x=3) for { $\neg$x<0, x=3 }

     $\Rightarrow$ Should be *refutation-sound*
       - Answers "{ l_1, …, l_n } is T-unsatisfiable" only if { l_1, …, l_n } is T-unsatisfiable
  3. Don't know
     - Return lemma, e.g. split on (x=y $\vee \neg$ x=y)

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { x $\rightarrow$ 2, y $\rightarrow$ 3, z $\rightarrow$ -3} for { x<y, y+z=0 }

     $\Rightarrow$ Should be *solution-sound*
       - Answers "M is T-satisfiable" only if M is T-satisfiable
  2. { l_1, …, l_n } $\subseteq$ M is T-unsatisfiable
     - Return *conflict* clause, e.g. (x<0 $\vee \neg$ x=3) for { $\neg$x<0, x=3 }

     $\Rightarrow$ Should be *refutation-sound*
       - Answers "{ l_1, …, l_n } is T-unsatisfiable" only if { l_1, …, l_n } is T-unsatisfiable
  3. Don't know
     - Return lemma, e.g. split on (x=y $\vee \neg$ x=y)

  $\Rightarrow$If solver is solution-sound, refutation-sound, and *terminating*,
     - Then it implements a *decision procedure* for T

Theory Solvers: Linear Arithmetic

Linear Arithmetic

- Quantifier-free linear real and integer arithmetic
  QF_LRA, QF_LIA, QF_LIRA

- Given the linear inequalities

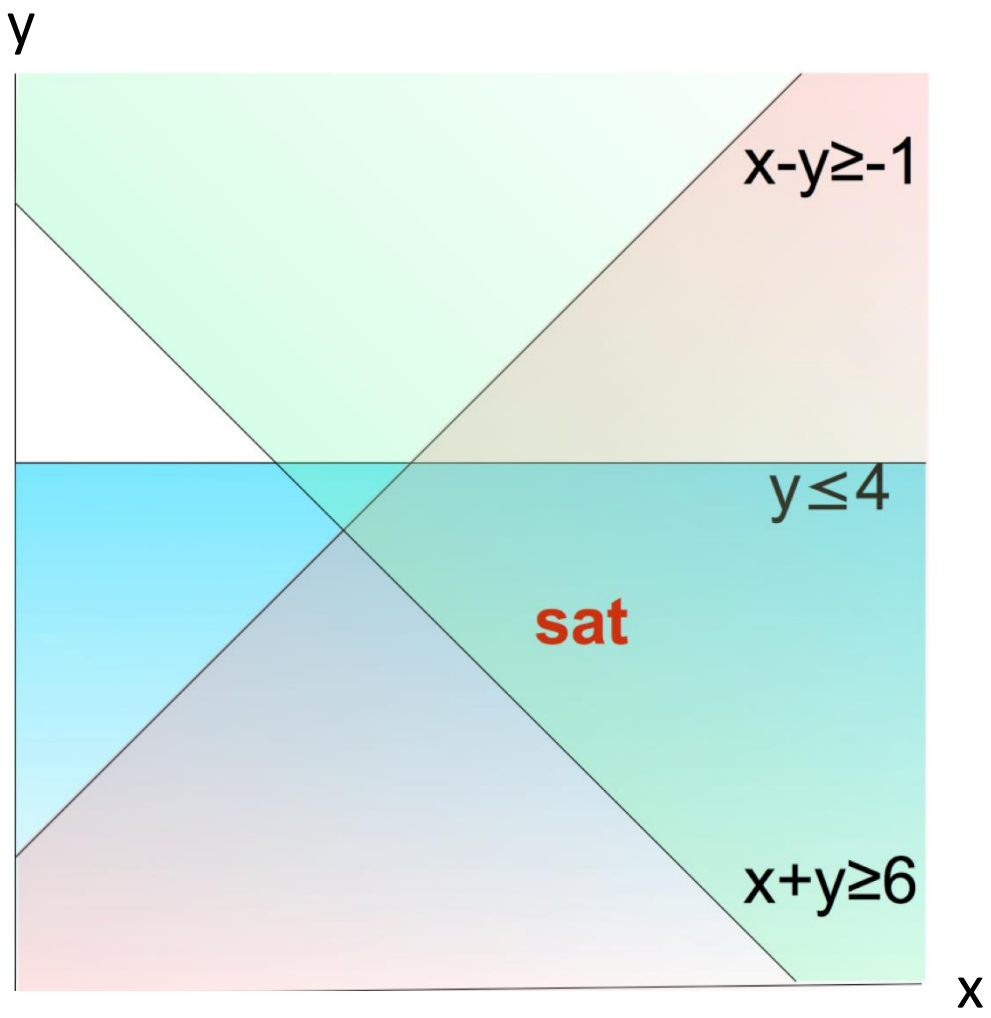
$$x : \text{Real}; \; y : \text{Real};$$

$$x - y \geq -1, \;\; y \leq 4, \;\; x + y \geq 6$$

  is there an assignment to x and y that makes all of them true?

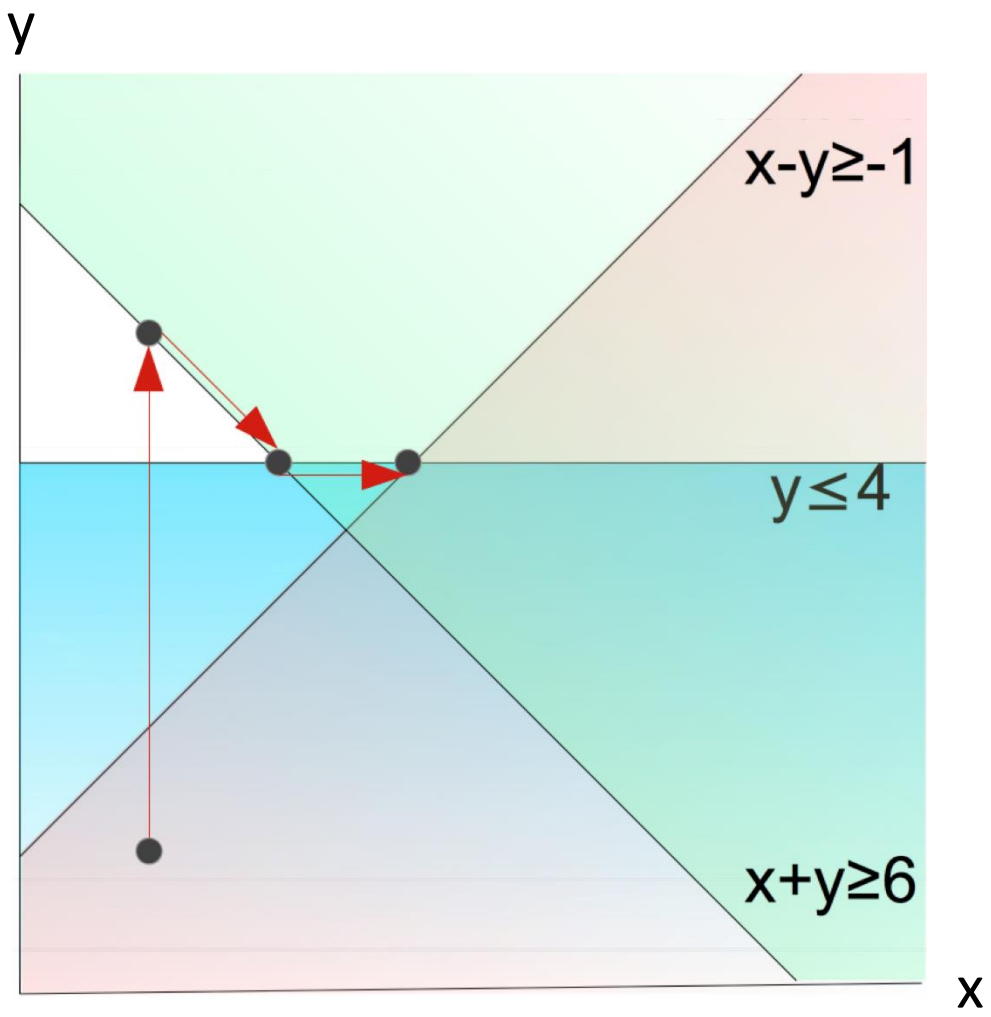
- Solve using simplex-based approaches **[Dutertre/de Moura 2006]**

Simplex Search



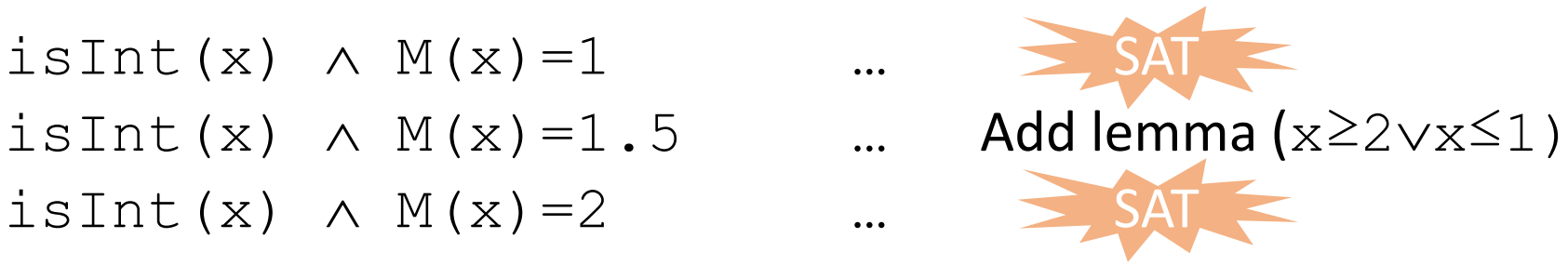
Is an intersection of half planes empty?

Simplex Search



From Reals to (Mixed) Integers

- Add `isInt(x)` constraints
- First solve real relaxation
  - Ignore `isInt(x)` constraints
- If real relaxation is sat:
  - Check if current assignment `M(x)` satisfies `isInt(x)` constraints
  - If not, refine by branching **[Dillig 2006, Griggio 2012, Jovanovic/de Moura 2013]**



Theory Solvers: Equality + Uninterpreted Functions (EUF)

Theory of Uninterpreted Functions (UF)

- Equalities and disequalities between terms built from UF, e.g.

$$a=b,\ f(a,f(b,g(a)))=d,\ g(b)=c,\ f(a,c)=c,\ f(a,c)\neq d$$

…where signature is:

"uninterpreted sort" U

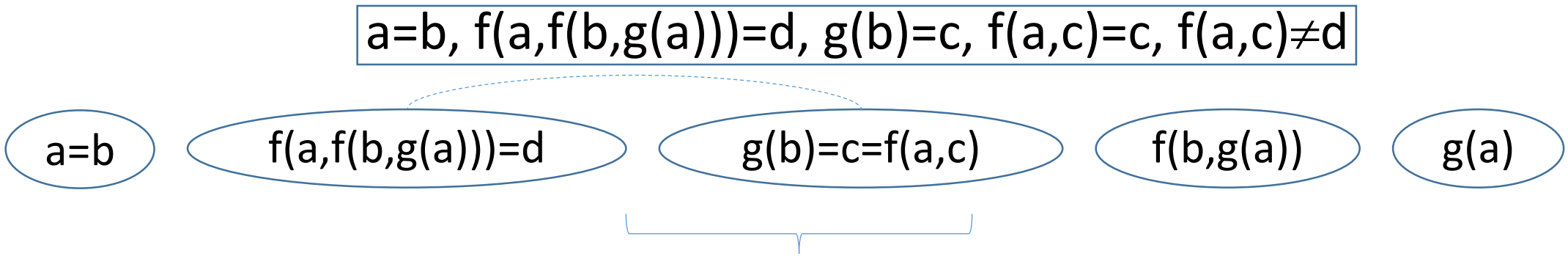
$a,b,c,d : U$

$g : U \rightarrow U$

$f : U \times U \rightarrow U$

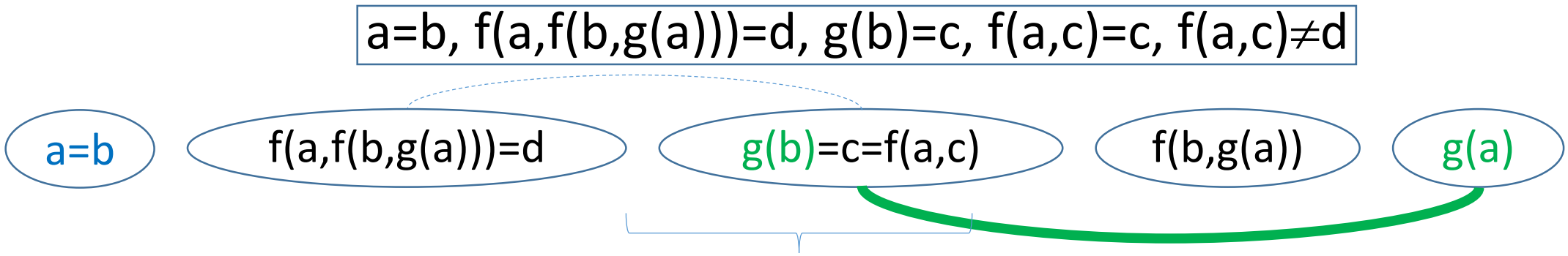
$\Rightarrow$ UF are useful for abstracting processes, other symbols not natively supported by solver

Congruence Closure



Compute *equivalence classes* of all (sub)terms

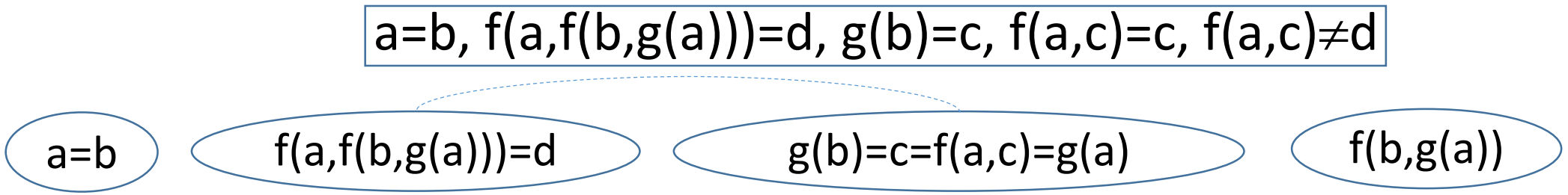
Congruence Closure



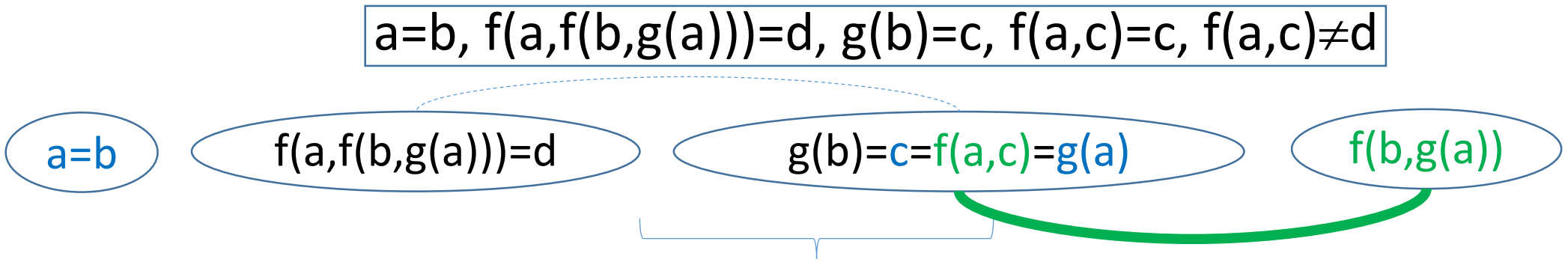
…merge congruent terms

since a=b, we know g(a)=g(b)

Congruence Closure



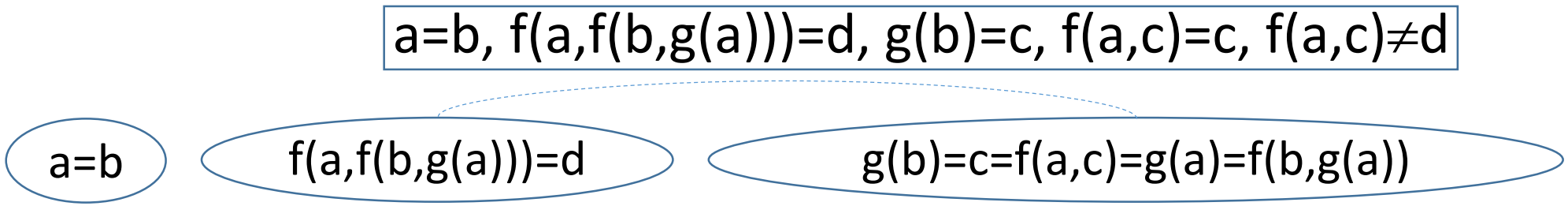
Congruence Closure



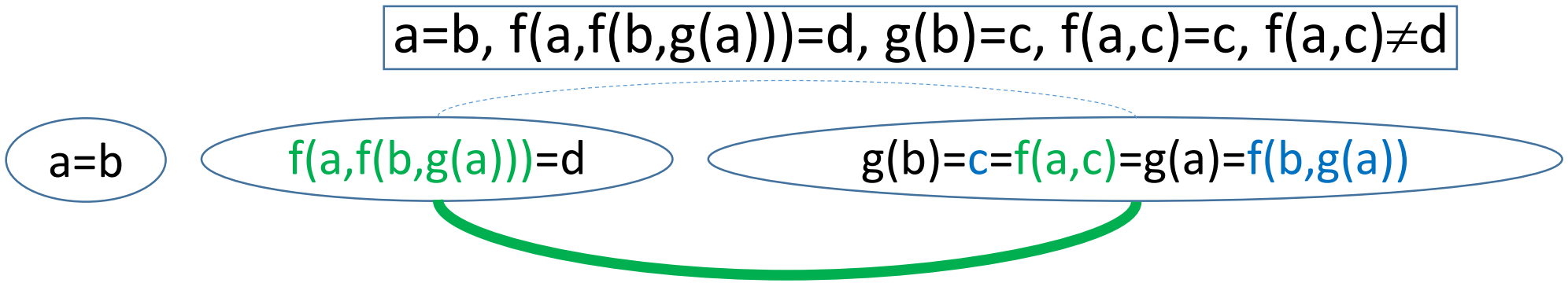
…merge congruent terms

since a=b and c=g(a), we know f(a,c)=f(b,g(a))

Congruence Closure



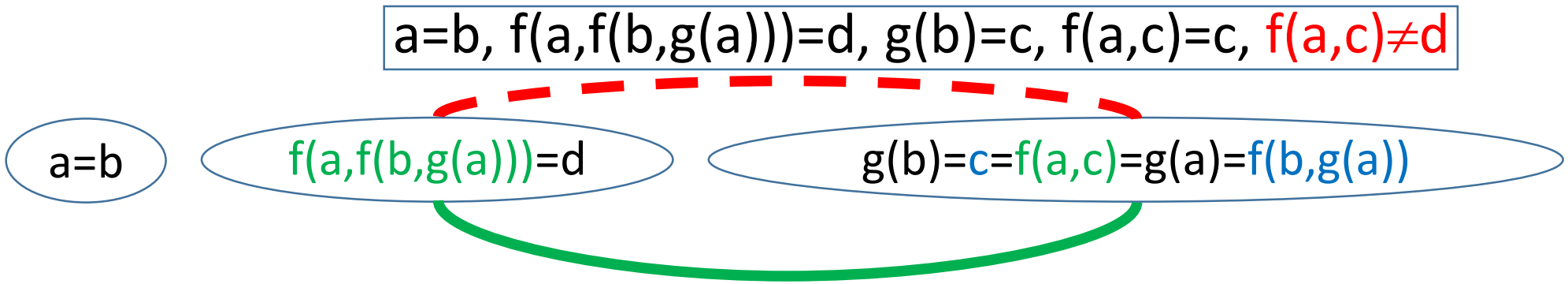
Congruence Closure



…merge congruent terms

since f(b,g(a))=c, we know f(a,f(b,g(a)))=f(a,c)

Congruence Closure

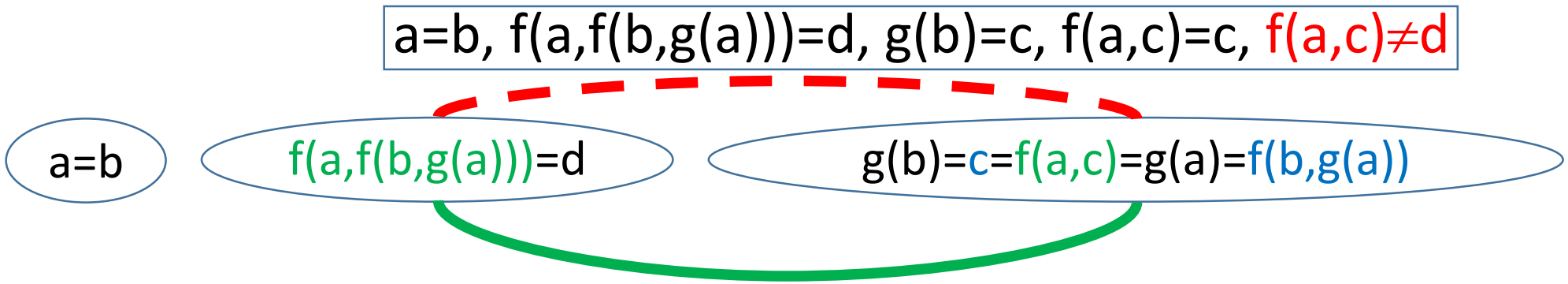


…merge congruent terms

since f(b,g(a))=c, we know f(a,f(b,g(a)))=f(a,c)

$\Rightarrow$ Must merge disequal equivalence classes … conflict!

Congruence Closure



…merge congruent terms

since f(b,g(a))=c, we know f(a,f(b,g(a)))=f(a,c)

$\Rightarrow$ Must merge disequal equivalence classes … conflict!

Congruence closure is important building block for many decision procedures

Theory Solvers: Arrays

Arrays : Signature Σ

Types:

(Array T_1 T_2) : Arrays with index type T_1, element type T_2

Operators:

(store a i v) : Array obtained by writing v at index i in Array a

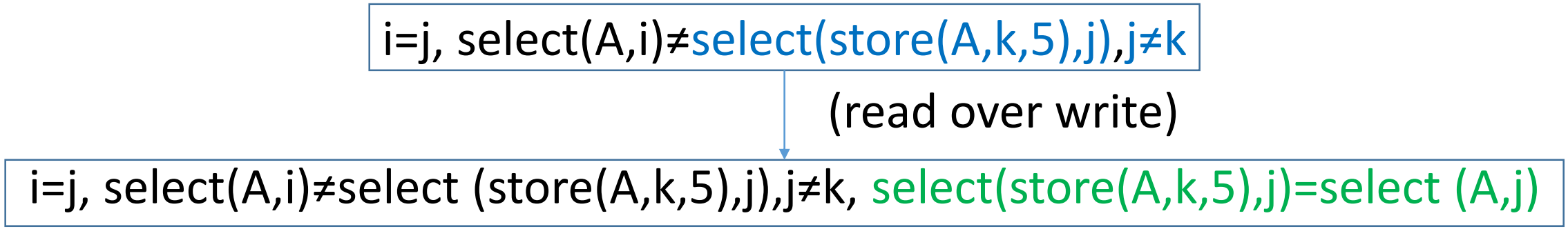
(select a i) : Element at index i of Array a

$\Rightarrow$ Arrays are useful for modelling memory, data structures

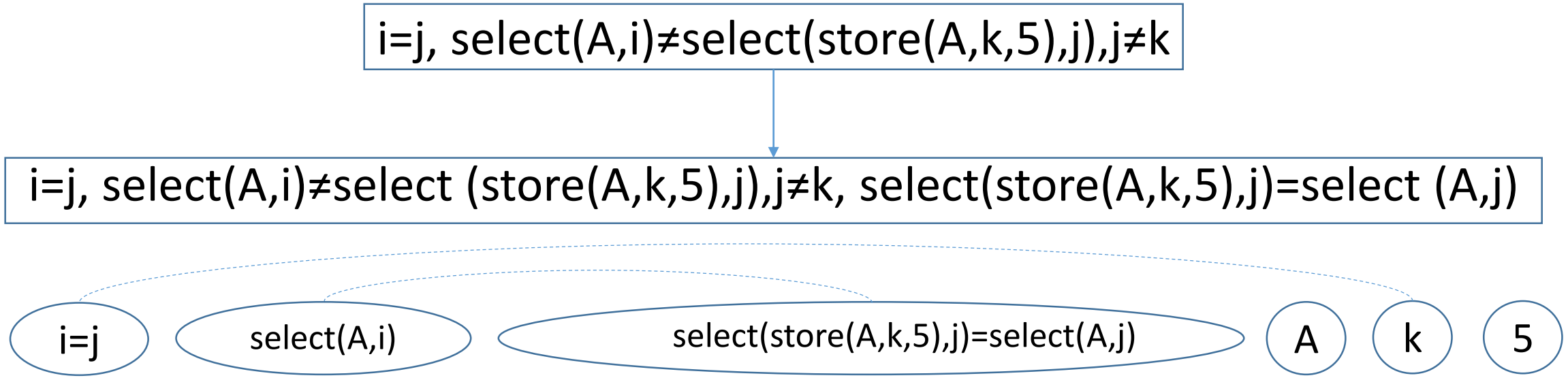
Procedure for Arrays with Extensionality

$i=j,\ select(A,i)\neq select(store(A,k,5),j),j\neq k$

Procedure for Arrays with Extensionality

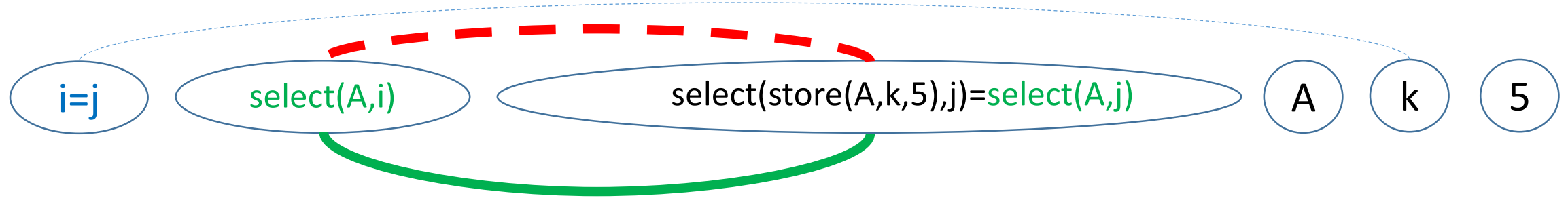


Procedure for Arrays with Extensionality



Procedure for Arrays with Extensionality

i=j, select(A,i)≠select(store(A,k,5),j),j≠k



…merge congruent terms

since i=j, we know select(A,i)=select(A,j)

conflict!

Theory Solvers: Bitvectors

Bit Vectors

- Bit-vectors parameterized by a bit-width

  (_ BitVec 2) : #b00,#b01,#b10,#b11

  (_ BitVec 10) : #b0000000000,#b0101010101,#b1111111110,…

  …

- For each bit-width, large signature containing operators for:
  - (Modular) arithmetic
  - Bitwise logical operations
  - Bit-shifting
  - Concatenation/extraction

⇒ Bit-vectors are useful for modelling machine integers, circuits

SMT-LIB Bitvectors

(_ BitVec n)

(_ #bv$X$ n)

#bX

#xX

(bvshl s t)

(bvlshr s t)

(bvashr s t)

(concat s t)

((_ extract i j) s)

((_ repeat i) s)

((_ zero_extend i) s)

((_ sign_extend i) s)

((_ rotate_left i) s)

((_ rotate_right i) s)

(bvnot s)

(bvand s t)

(bvnand s t)

(bvor s t)

(bvnor s t)

(bvxor s t)

(bvxnor s t)

SMT-LIB Bitvectors

(bvneg s)
(bvadd s t)
(bvsub s t)
(bvmul s t)

(bvudiv s t)
(bvurem s t)
(bvsdiv s t)
(bvsrem s t)
(bvsmod s t)

(bvcomp s t)
(bvult s t)
(bvule s t)
(bvugt s t)
(bvuge s t)
(bvslt s t)
(bvsle s t)
(bvsgt s t)
(bvsge s t)

Solving Bit Vectors

- Typically, bit-vector constraints are solved by bit-blasting

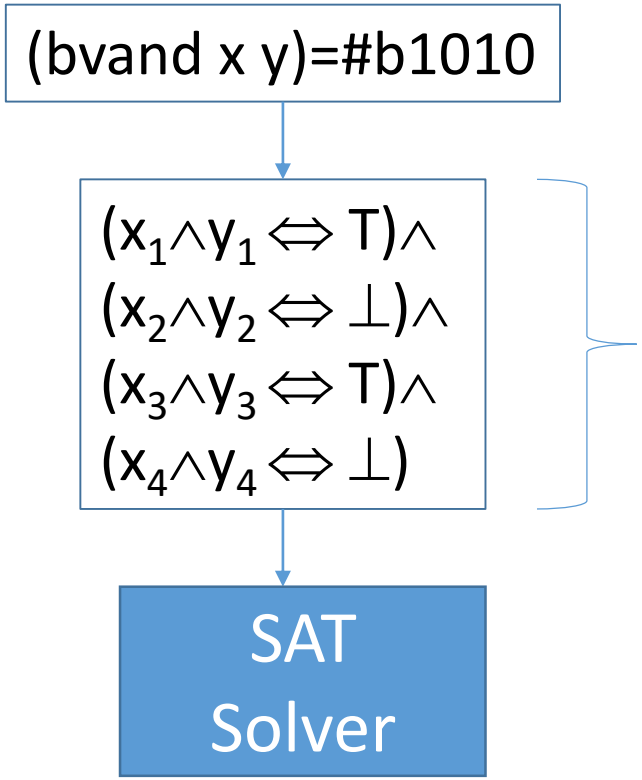
  $\Rightarrow$Eager reduction to propositional satisfiability

- For example:

```
(bvand x y)=#b1010
```

Solving Bit Vectors

- For example:



Relies on developing good encodings for each operator in bit-vector signature

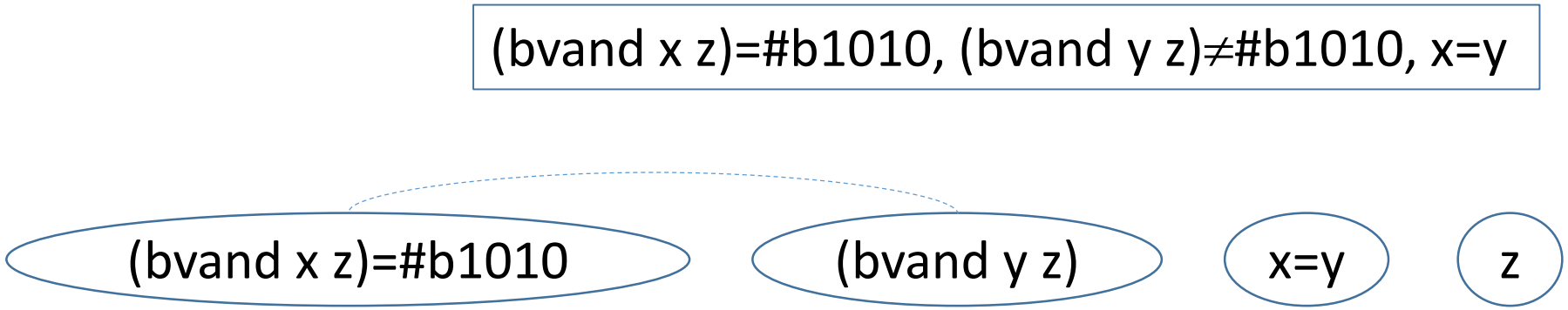
Solving Bit Vectors

- Bit-blasting can also be done lazily **[Bruttomesso et al 2007,Hadarean et al 2014]**
- Instead of:

(bvand x z)=#b1010, (bvand y z)≠#b1010, x=y

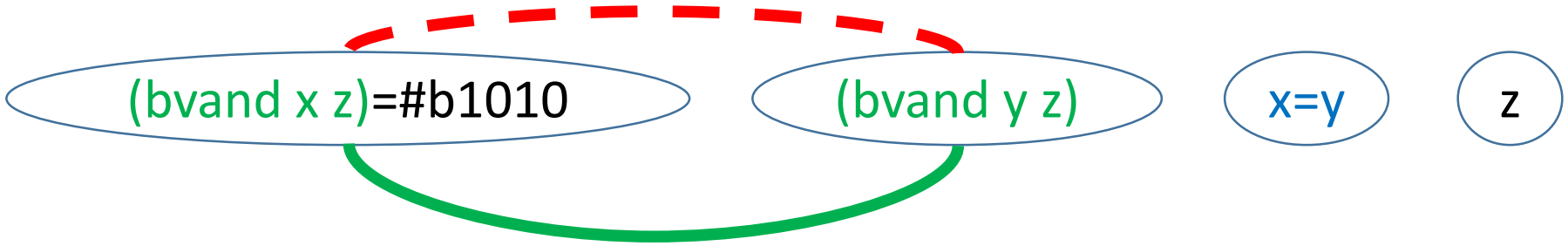
$$
\begin{aligned}
&(x_1 \wedge z_1 \Leftrightarrow T) \wedge (y_1 \wedge z_1 \Leftrightarrow T) \wedge (x_1 \Leftrightarrow y_1) \wedge \\
&(x_2 \wedge z_2 \Leftrightarrow \bot) \wedge (y_2 \wedge z_2 \Leftrightarrow \bot) \wedge (x_2 \Leftrightarrow y_2) \wedge \\
&(x_3 \wedge z_3 \Leftrightarrow T) \wedge (y_3 \wedge z_3 \Leftrightarrow \bot) \wedge (x_3 \Leftrightarrow y_3) \wedge \\
&(x_4 \wedge z_4 \Leftrightarrow \bot) \wedge (y_4 \wedge z_4 \Leftrightarrow T) \wedge (x_4 \Leftrightarrow y_4)
\end{aligned}
$$

Solving Bit Vectors



Solving Bit Vectors

(bvand x z)=#b1010, (bvand y z)≠#b1010, x=y



…merge congruent terms

since x=y, we know (bvand x z)=(bvand y z)

⇒ ***Conflict***, *before resorting to bit-blasting*

Theory Solvers: Finite Sets + Cardinality

Theory of Finite Sets + Cardinality

- Parametric theory of finite sets of elements $\mathtt{T}$
- Signature Σ_{Set}:
  - Empty set $\emptyset$, Singleton $\mathtt{\{a\}}$
  - Membership $\in$: $\mathtt{T}$ x $\mathtt{Set(T)} \rightarrow \mathtt{Bool}$
  - Subset $\subseteq$: $\mathtt{Set(T)}$ x $\mathtt{Set(T)} \rightarrow \mathtt{Bool}$
  - Set connectives $\cup, \cap, \setminus$: $\mathtt{Set(T)}$ x $\mathtt{Set(T)} \rightarrow \mathtt{Set(T)}$

- Example input: $\mathtt{x{=}y{\cap}z \wedge a{+}5{\in}x \wedge y{\subseteq}w}$

$\Rightarrow$Sets are important in databases, knowledge representation, programming languages, e.g. Alloy

Theory of Finite Sets + Cardinality

- Extended signature of theory to include:
  - Cardinality $|.| : \texttt{Set} \rightarrow \texttt{Int}$
- Extended decision procedure for cardinality constraints
  [Bansal et al IJCAR2016]

- Example input: $x = y \cup z \wedge |x| = 14 \wedge |y| \geq |z| + 5$

Extended to Theory of Finite Relations [Meng et al 2017]

$\Rightarrow$ *Relations* Rel(α) can be modeled as a *set of tuples* Set(Tup(α))

Tuple constructor:

$$\langle \_, \ldots, \_ \rangle : \alpha_1 \times \cdots \times \alpha_n \rightarrow \mathrm{Tup}_n(\alpha_1, \ldots, \alpha_n)$$

Product: $* : \mathrm{Rel}_m(\alpha) \times \mathrm{Rel}_n(\beta) \rightarrow \mathrm{Rel}_{m+n}(\alpha, \beta)$

Join: $\bowtie : \mathrm{Rel}_{p+1}(\alpha, \gamma) \times \mathrm{Rel}_{q+1}(\gamma, \beta) \rightarrow \mathrm{Rel}_{p+q}(\alpha, \beta)$
with $p + q > 0$

Transpose: $\_^{-1} : \mathrm{Rel}_m(\alpha_1, \cdots, \alpha_m) \rightarrow \mathrm{Rel}_m(\alpha_m, \cdots, \alpha_1)$

Transitive Closure: $\_^{+} : Rel_2(\alpha, \alpha) \rightarrow Rel_2(\alpha, \alpha)$

Theory Solvers: Strings

Basic String Constraints

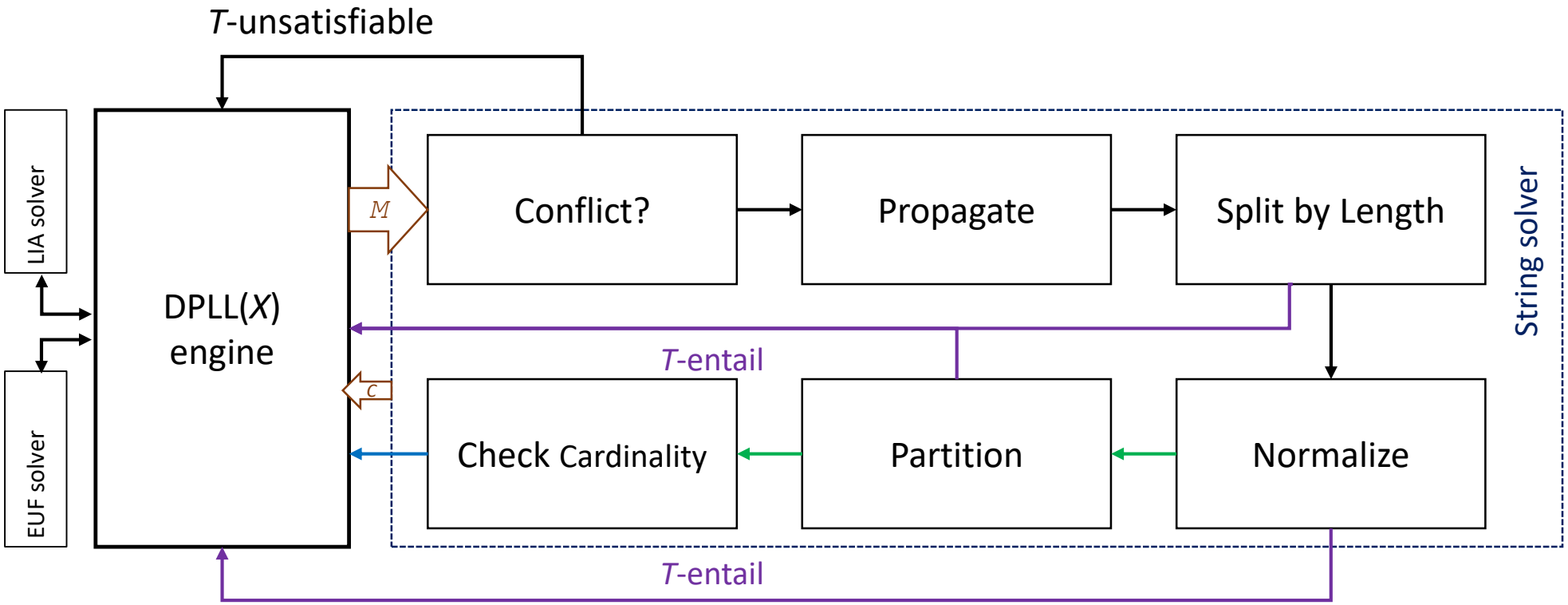
- Equalities and disequalities between *string terms* :
  - Variables: x
  - String constants: “abc”
  - Concatenation: $x \cdot \text{“abc”}$
  - Length: $|x|$
- *Linear arithmetic* constraints: $|x|+4>|y|$

Example: $x \cdot \text{“a”} = y \wedge |y| > |x| + 2$

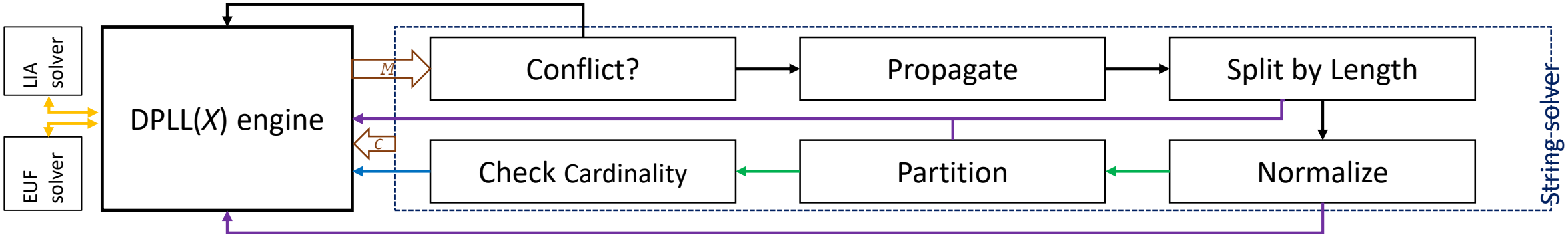
Procedures in **[Abdulla et al CAV2014, Liang et al CAV2014]**

⇒Strings are important in security applications, e.g. for detecting attack vulnerabilities

General String Solver Architecture



DPLL(*T*): Find Satisfying Assignment



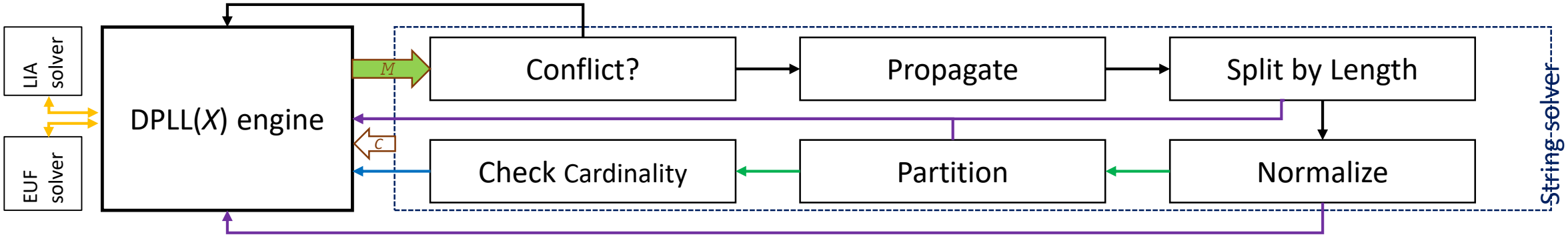
$$|x| > |y|$$
$$(x \cdot z = y \cdot w \cdot \text{“ab”} \vee x = y)$$

SAT Solver

Arithmetic Solver

String Solver

DPLL(*T*): Find Satisfying Assignment



|x|>|y|
(x·z=y·w·"ab" ∨ x=y)

SAT Solver

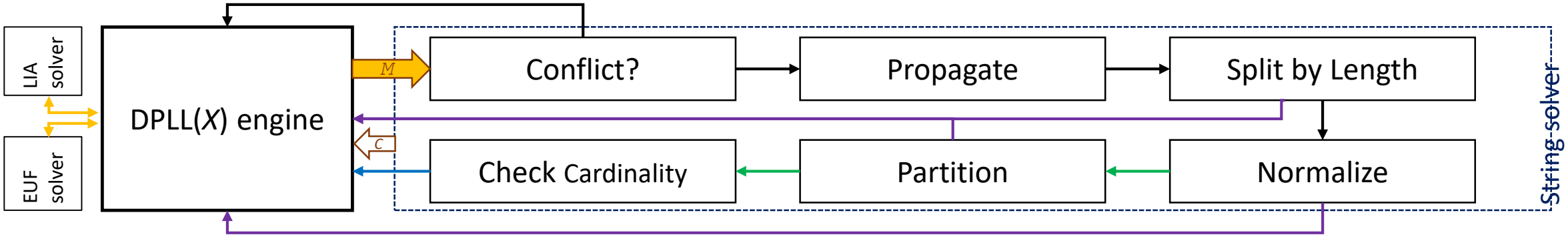
|x|>|y|

Arithmetic Solver

x·z = y·w·"ab"

String Solver

DPLL(*T*): Find Satisfying Assignment



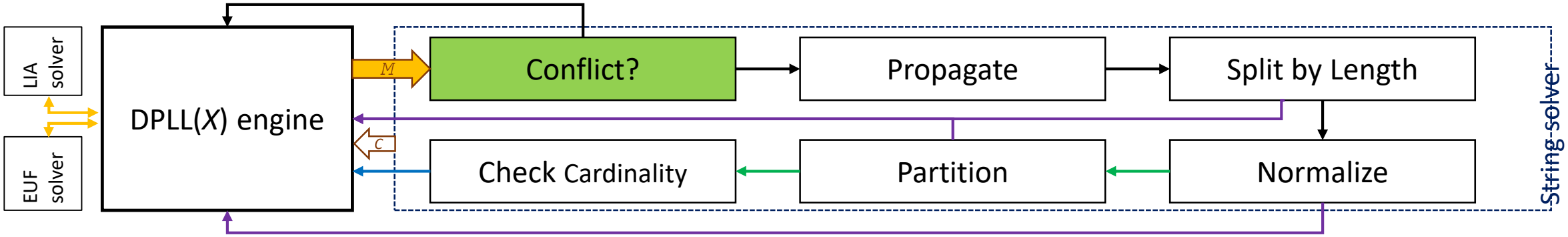
`|x| > |y|`

Arithmetic Solver

`x·z = y·w·"ab"`

String Solver

Conflict Checking



`|x| > |y|`

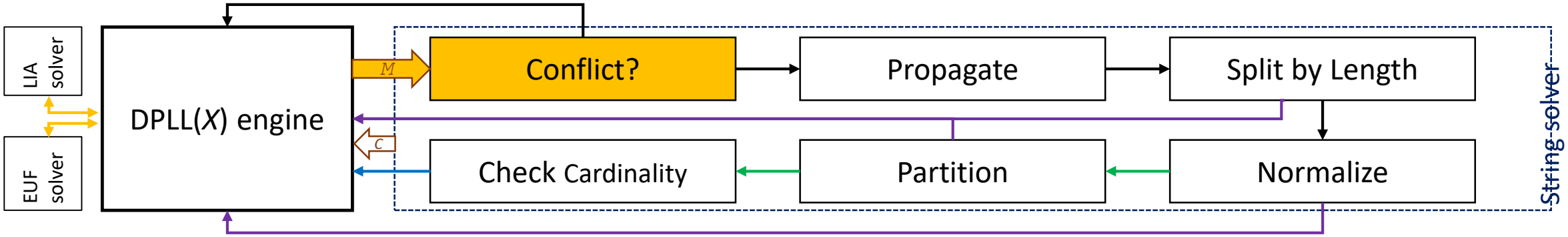
Arithmetic Solver

`x·z = y·w·"ab"`

String Solver

- String context contains dis-equalities like: **x≠x**
- Arithmetic context contains a contradiction: **|x|<|y|<|x|**

Conflict Checking



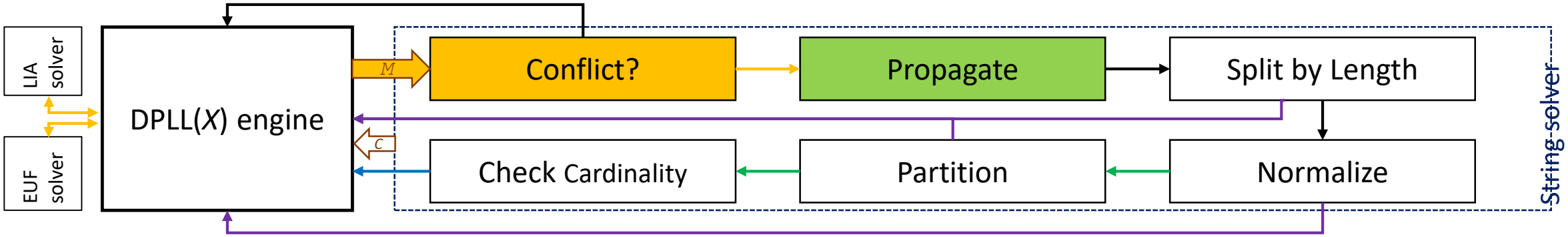
`|x| > |y|`

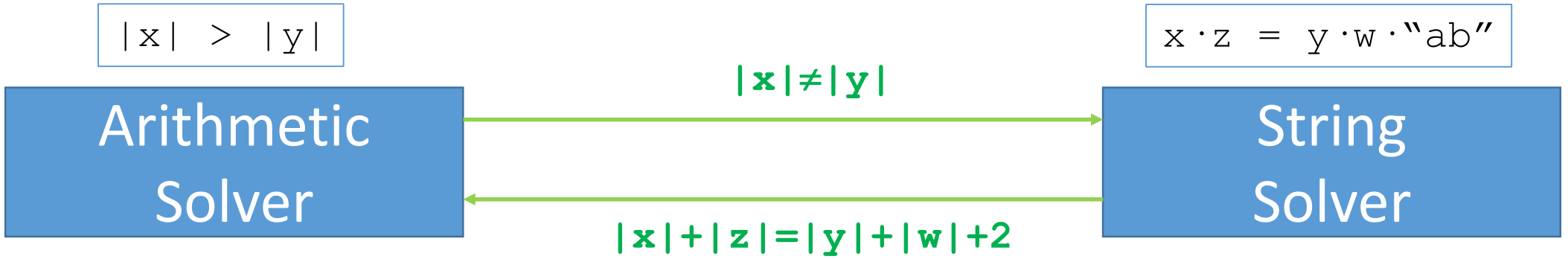
Arithmetic Solver

`x·z = y·w·"ab"`

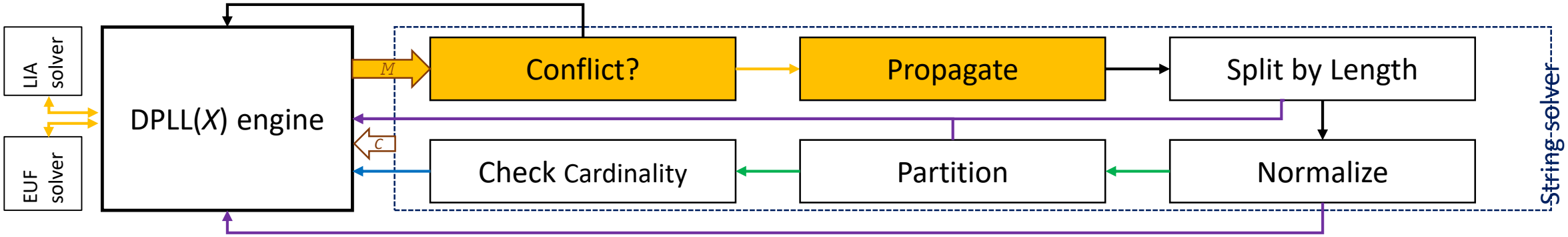
String Solver

Shared Term Propagation





Shared Term Propagation



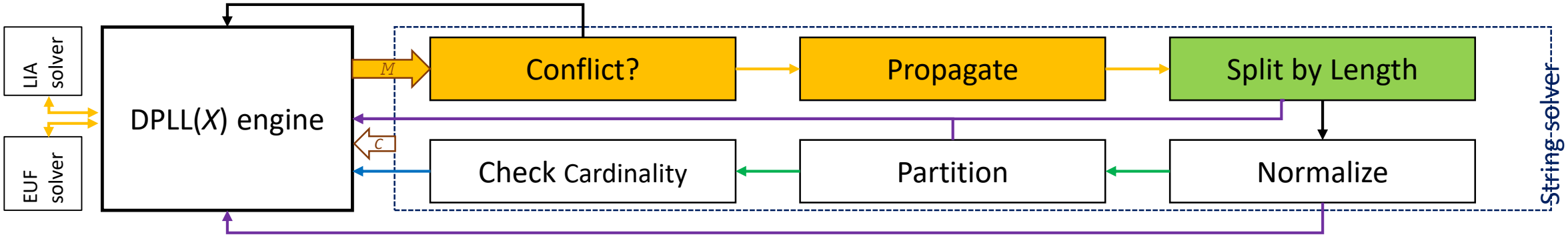
$|x| + |z| = |y| + |w| + 2$
$|x| > |y|$

Arithmetic Solver

$|x| \neq |y|$
$x \cdot z = y \cdot w \cdot \text{“ab”}$

String Solver

Length Splitting



$|x| + |z| = |y| + |w| + 2$
$|x| > |y|$

Arithmetic Solver

$|x| \neq |y|$
$x \cdot z = y \cdot w \cdot \text{“ab”}$

String Solver

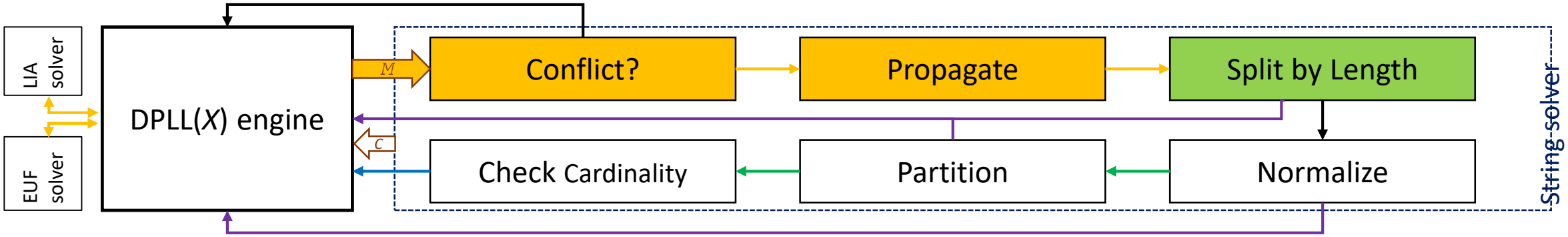
$|x| = 0$ $|y| = 0$ …

$|x| = 0$ $|y| > 0$ …

$|x| > 0$ $|y| = 0$ …

$|x| > 0$ $|y| > 0$ …

Length Splitting



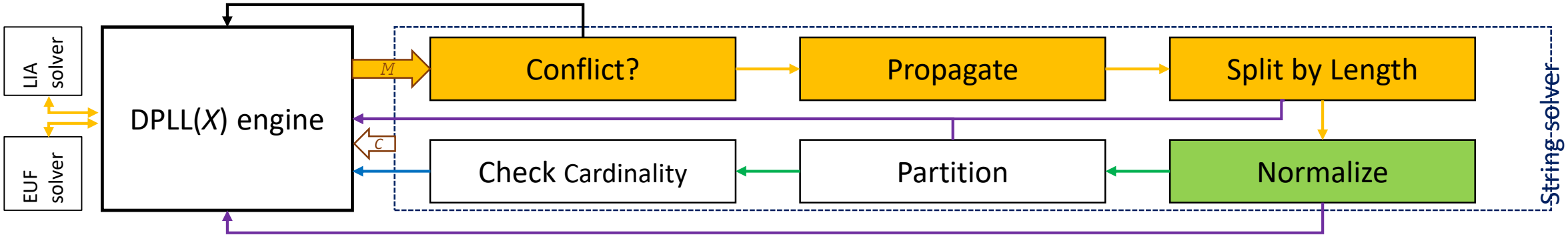
$$|x| + |z| = |y| + |w| + 2$$
$$|x| > |y|$$
$$|x| > 0$$
$$|y| > 0$$

Arithmetic Solver

$$|x| \neq |y|$$
$$x \cdot z = y \cdot w \cdot \text{“ab”}$$

String Solver

Normalization



$|x| \neq |y|$

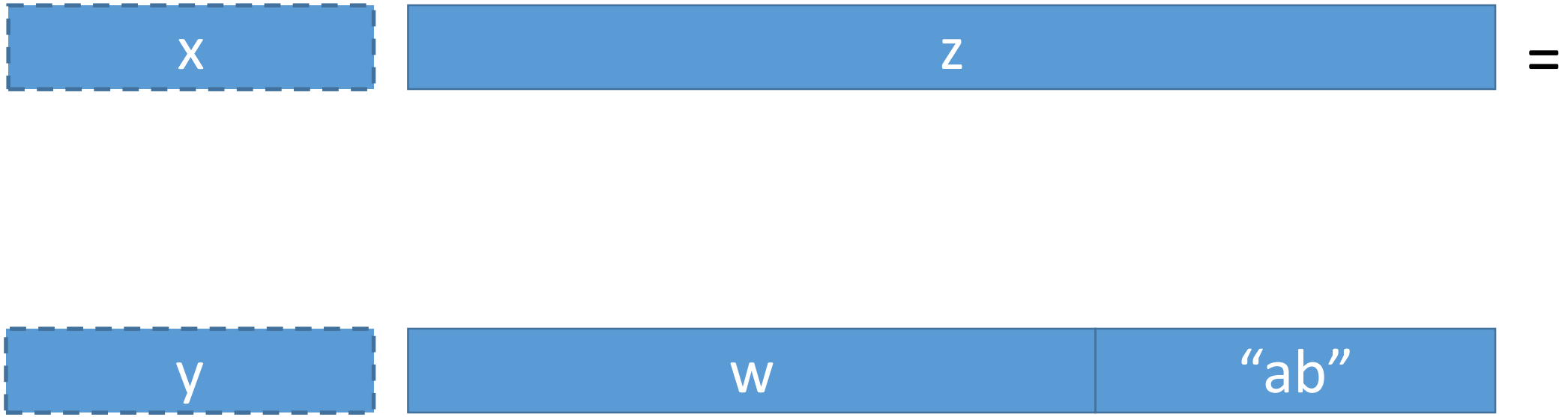
$x \cdot z = y \cdot w \cdot \text{“ab”}$

String Solver

Normalize Equalities

S { **x** ·z=**y** ·w ·“ab”

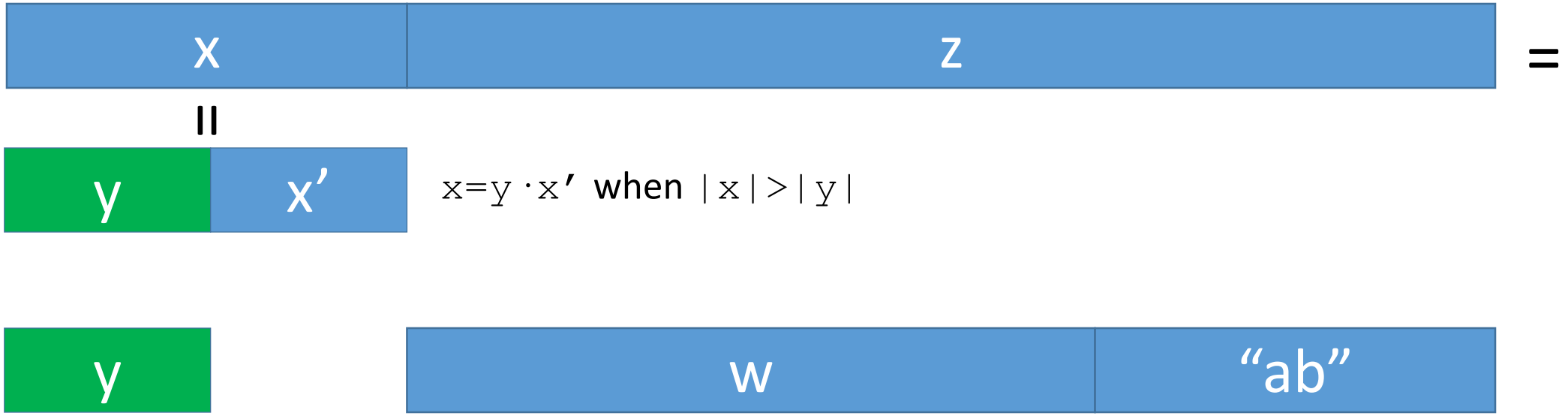
To handle string equalities with prefix **x** and **y** must consider 3 cases…



Normalize Equalities

S { x·z=y·w·"ab"
x=y·x'

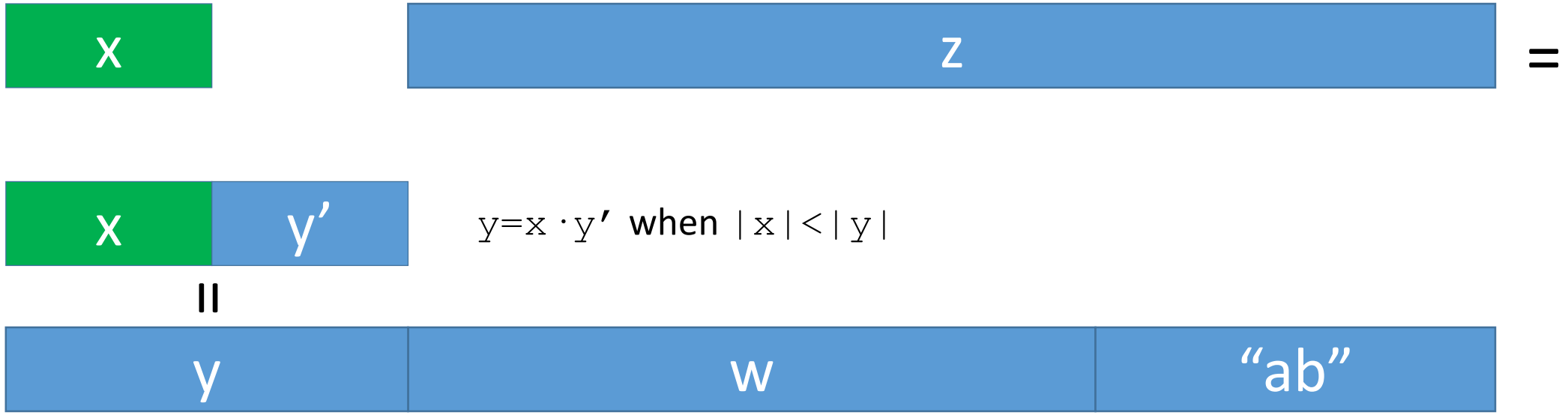
L { |x|>|y|



Normalize Equalities

S { x·z=y·w·"ab"
y=x·y′

L { |x|<|y|



Normalize Equalities

S

$x \cdot z = y \cdot w \cdot \text{“ab”}$

$x = y$

L

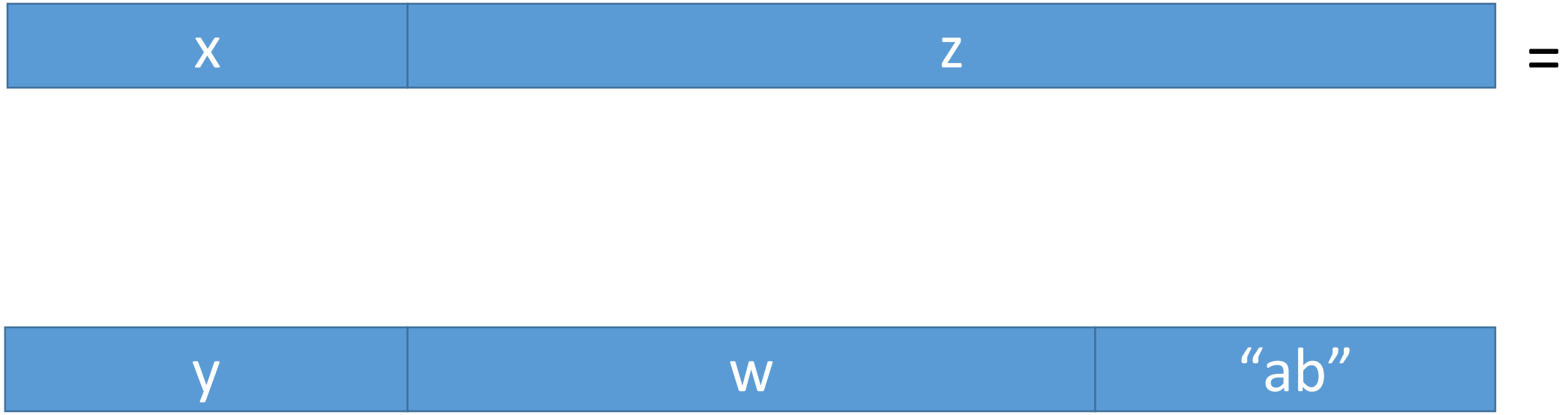
$|x| = |y|$

II $x = y$ when $|x| = |y|$

Normalize Equalities

S { x·z=y·w·“ab”

L { |x|=|y|
|z|>|w|



Normalize Equalities

S

$x \cdot z = y \cdot w \cdot \text{“ab”}$

$x = y$

L

$|x| = |y|$

$|z| > |w|$

II Since $|x| = |y|$

Normalize Equalities

S

$x \cdot z = y \cdot w \cdot \text{“ab”}$

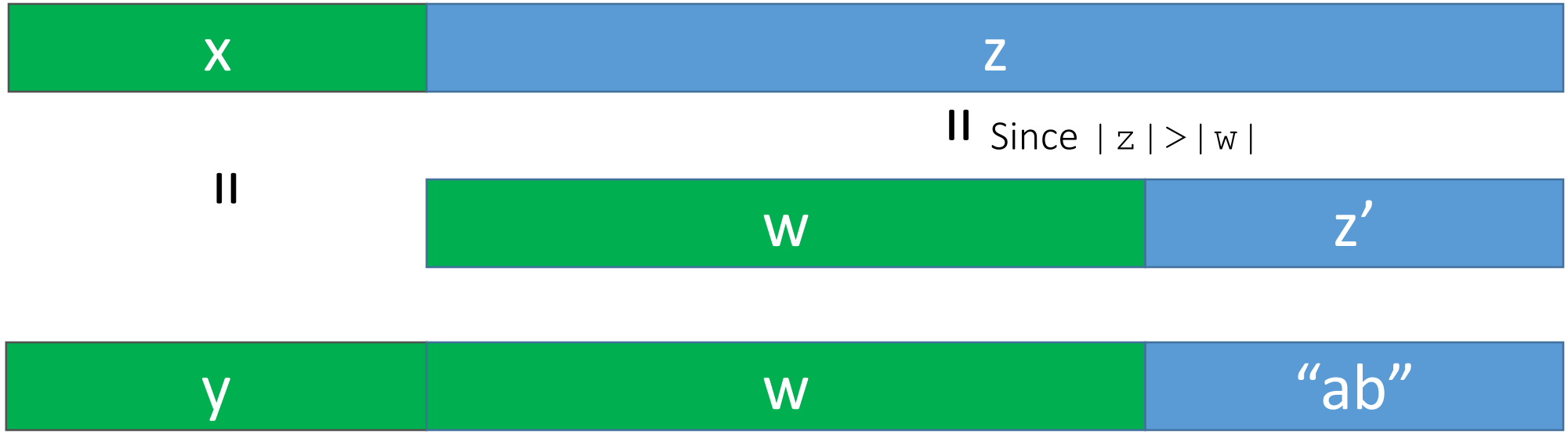
$x = y$

$z = w \cdot z'$

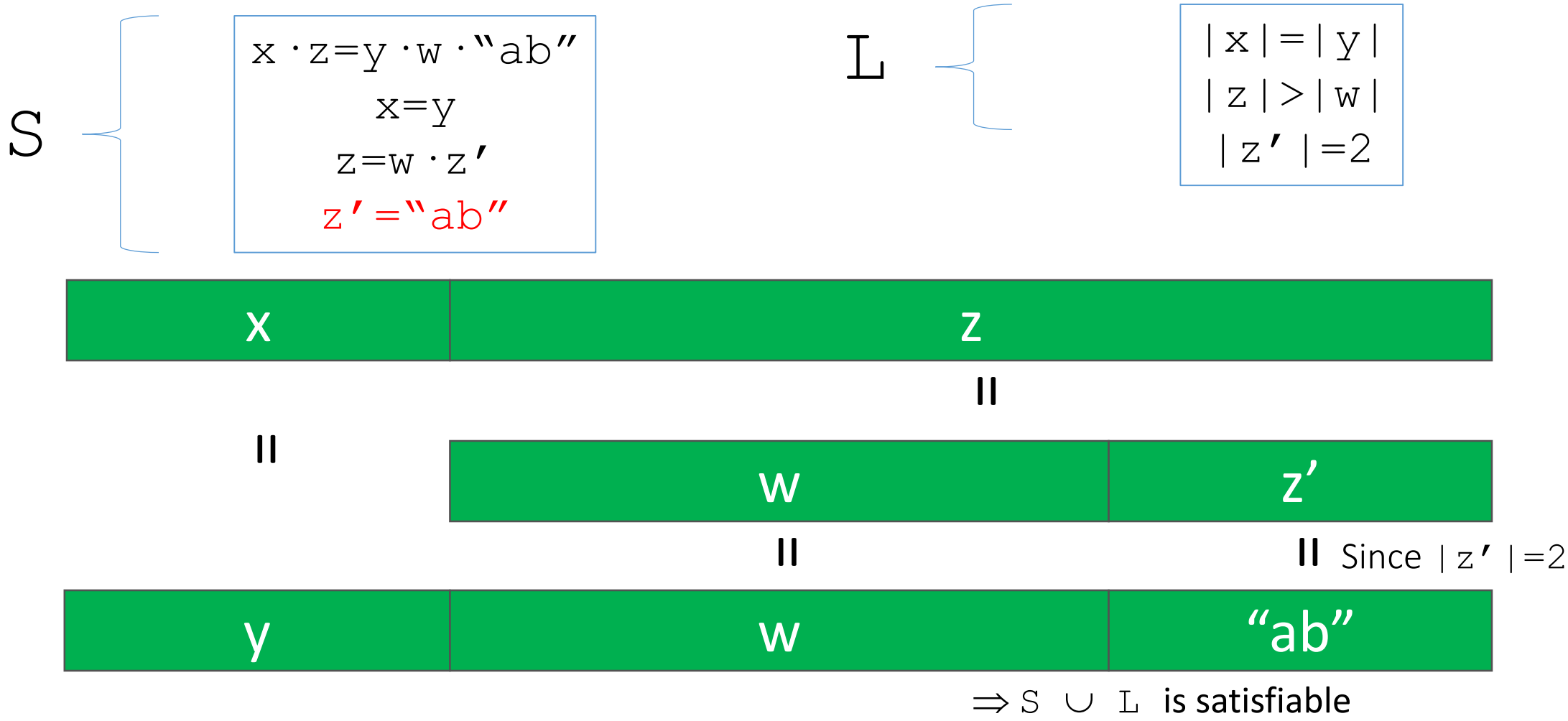
L

$|x| = |y|$

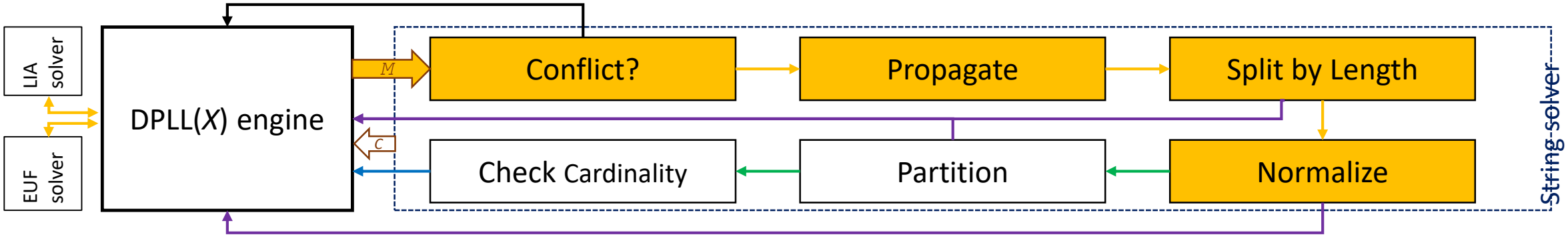
$|z| > |w|$



Normalize Equalities



Normalization



S

$x \cdot z = y \cdot w \cdot \text{“ab”}$

$x = y$

$z = w \cdot z'$

$z' = \text{“ab”}$

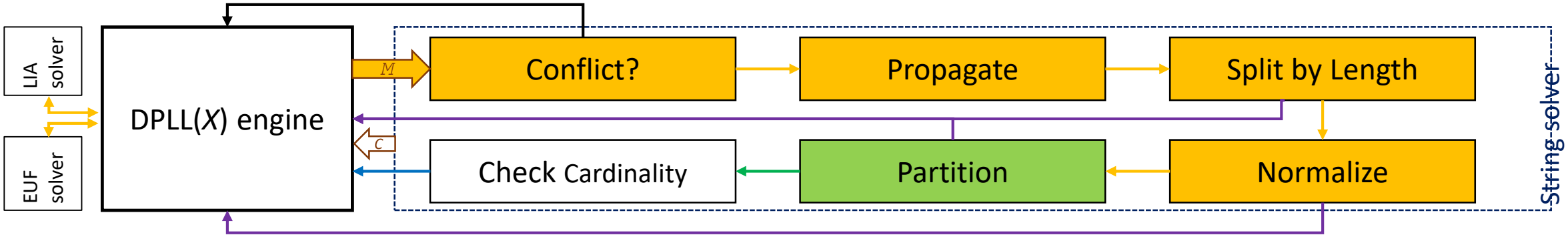
L

$|x| = |y|$

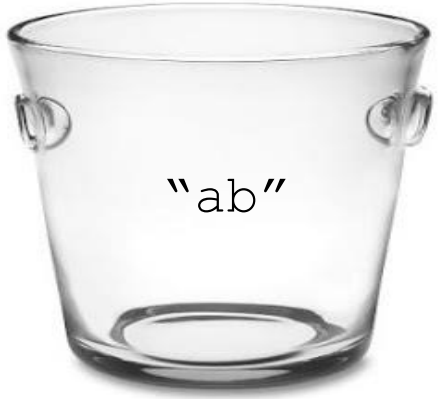
$|z| > |w|$

$|z'| = 2$

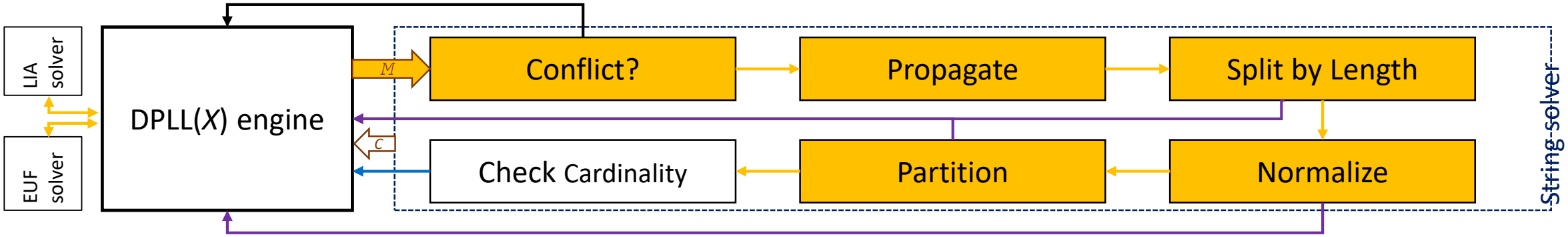
Partition



For strings `x, z, z', "ab"` `|x|=|z|, |x| ≠ |z'| ≠ |"ab"| ≠ |x|`



Check Cardinality of Σ

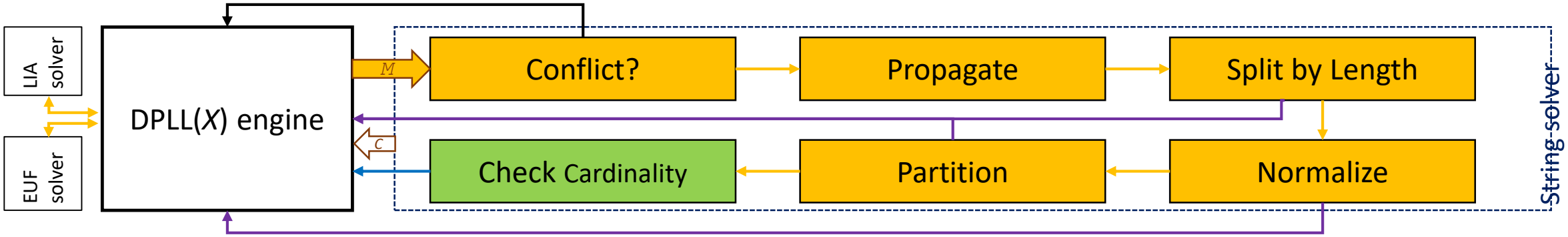


What if, for strings $s_1, s_2, \ldots, s_{256}, s_{257}$:

S — $s_i \neq s_j$ for each i, j such that $1 \leq i < j \leq 257$

L — $|s_1| = |s_2| = \ldots = |s_{256}| = |s_{257}|$

Check Cardinality of Σ

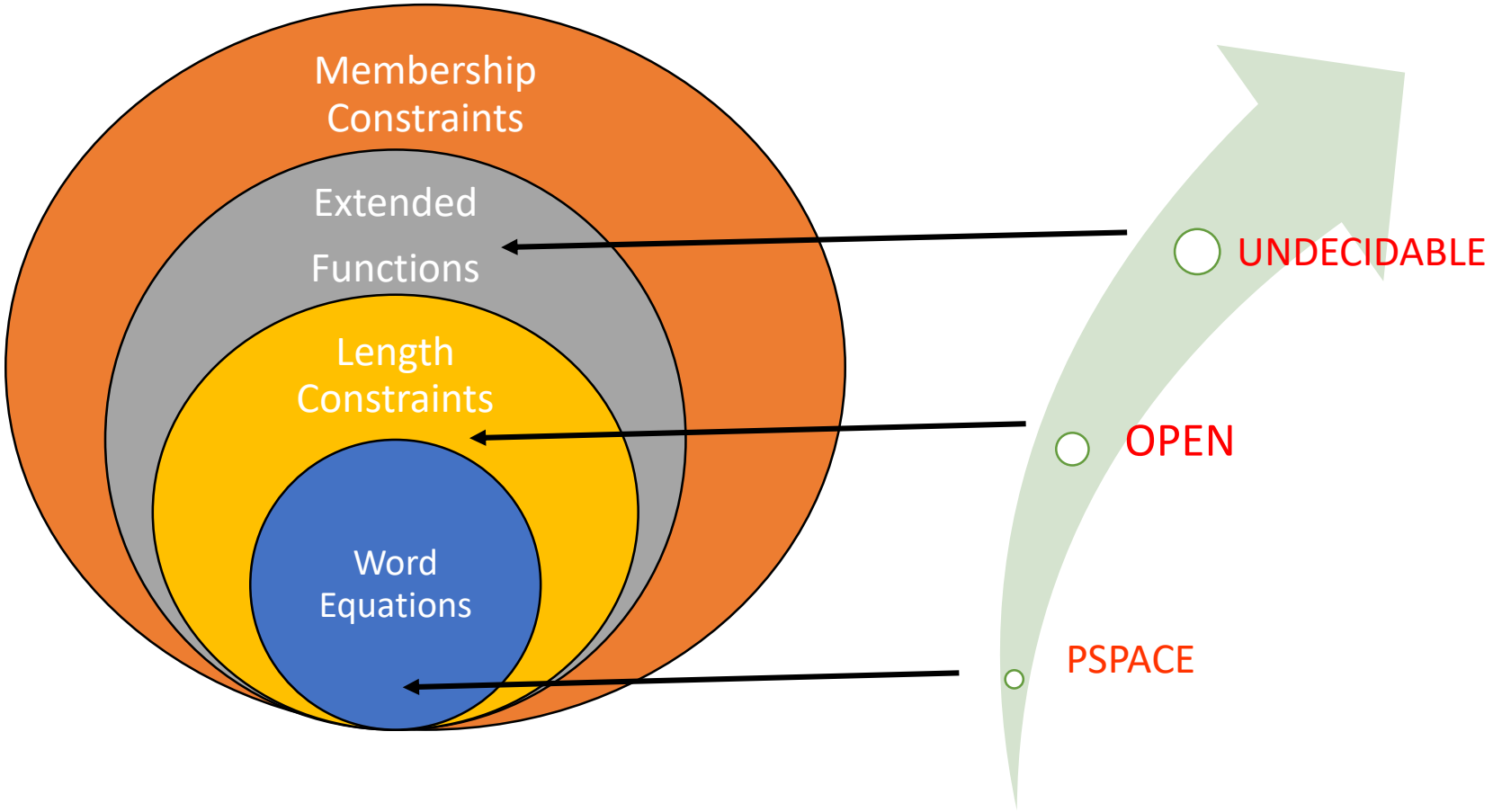


- May be unsatisfiable since Σ is finite
- For instance, if:
  - Σ is a finite alphabet of 256 characters, and
  - $S \cup L$ entails that 257 distinct strings of length 1 exist

  $\Rightarrow$ Then $S \cup L$ is unsatisfiable

$\therefore$ $(distinct(s_1, \ldots, s_{257}) \wedge |s_1| = \ldots = |s_{257}|) \Rightarrow |s_1| > 1$

Theoretical Complexity Challenges



Extended String Constraints

- Equalities and disequalities between:
  - *Basic string* terms*:*
    - String constants
    - String concatenation
    - String length
  - *Linear arithmetic* terms
  - *Extended* string terms:
    - Substring
    - String Contains
    - String find “index of”
    - String Replace

Examples

`“abc”`

`x·“abc”`

`|x|`

`x+4, y>2`

`substr(“abcde”,1,3)` = `“bcd”`

`contains(“abcde”,”cd”)` = `T`

`indexof(“abcde”,”d”,0)` = `3`

`replace(“ab”,“b”,“c”)` = `“ac”`

Example: ¬`contains(substr(x,0,3),“a”)` ∧ `0≤indexof(x,“ab”,0)<4`

How do we handle Extended String Constraints?

```
¬contains(x,“a”)
```

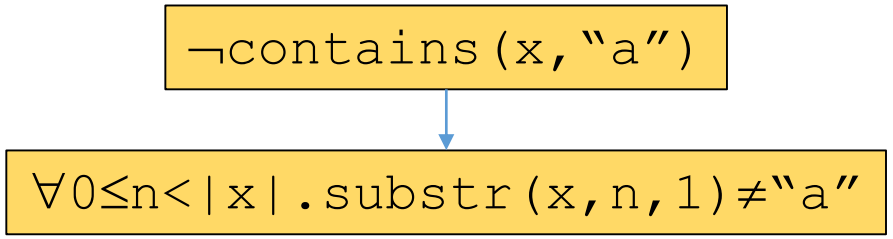
How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

```
¬contains(x,“a”)
```

How do we handle Extended String Constraints?

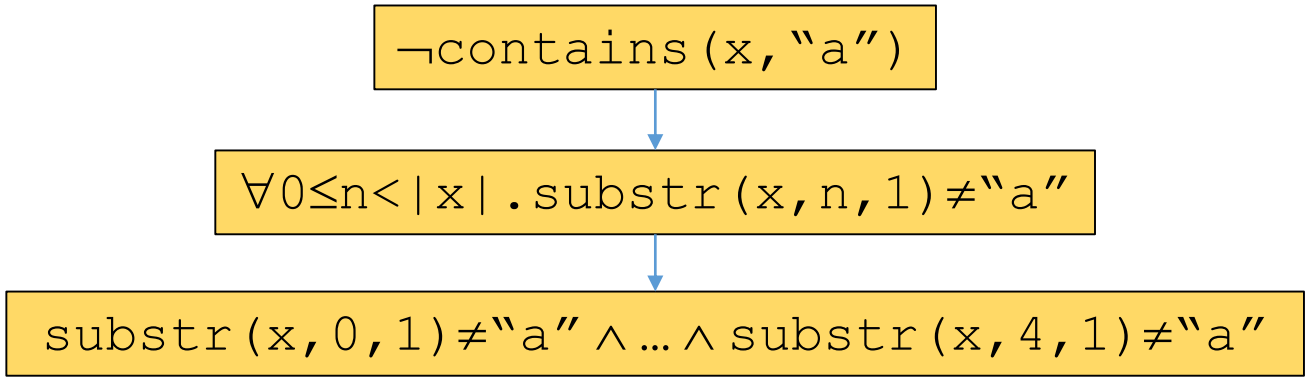
- Naively, by reduction to basic constraints + bounded ∀



Expand `contains`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

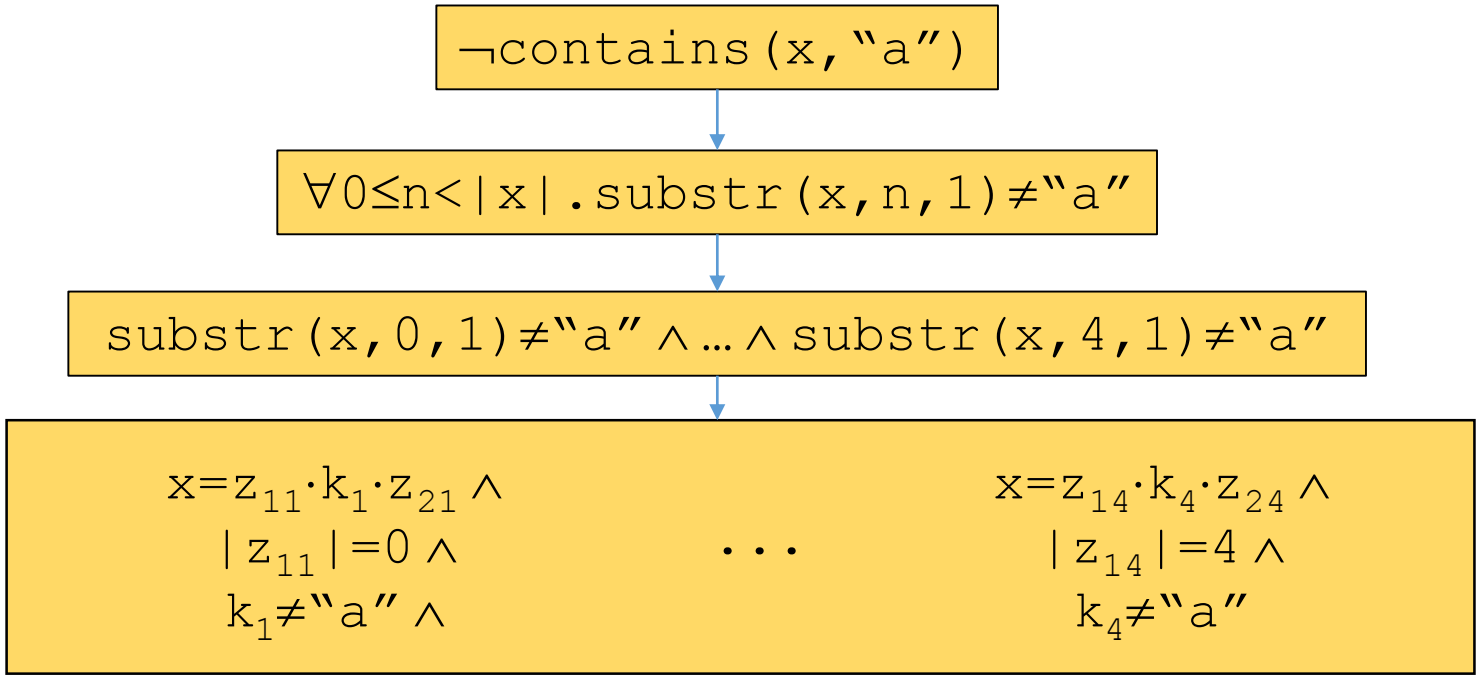


Expand `contains`

Assuming bound `|x|≤5`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀



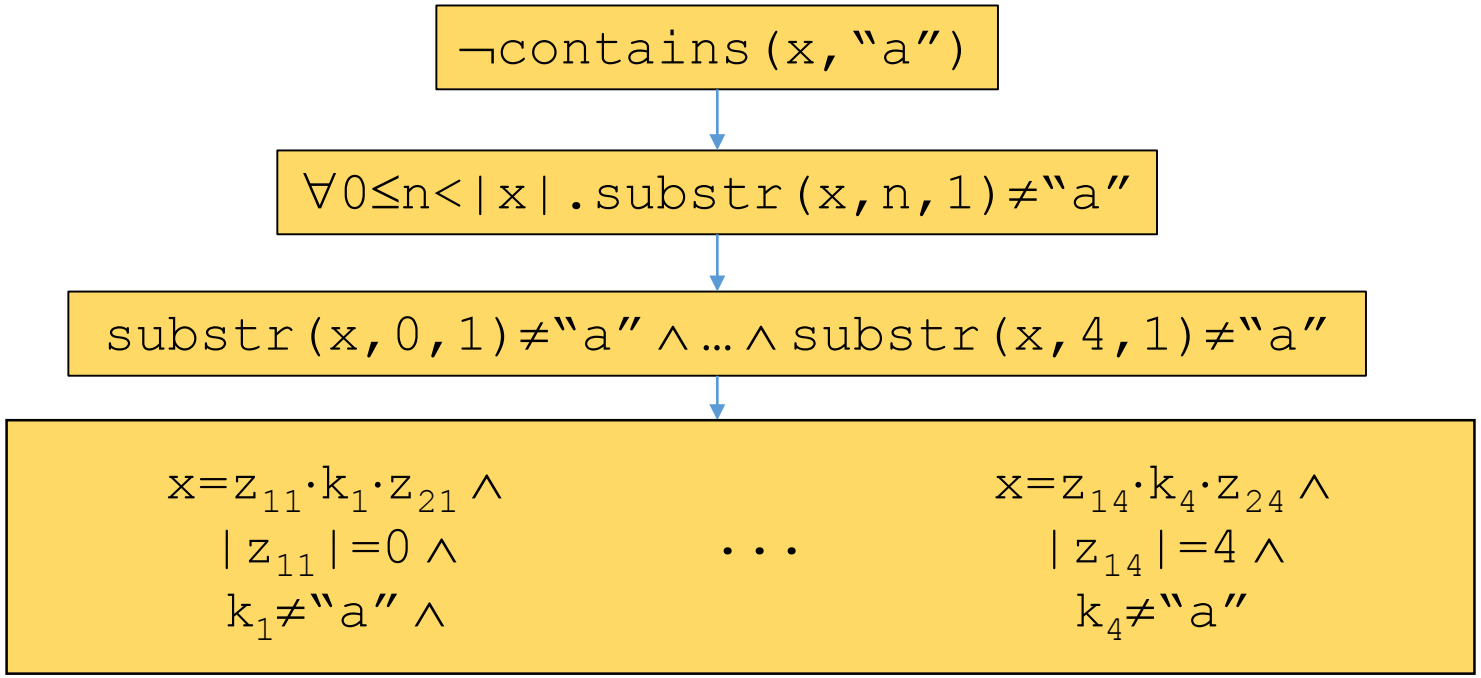
Expand `contains`

Assuming bound $|x| \leq 5$

Expand `substr`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀



Expand `contains`

Assuming bound `|x|≤5`

Expand `substr`

- Approach used by many current solvers

  [Bjorner et al 2009,Zheng et al 2013,Li et al 2013,Trinh et al 2014]

(Eager) Expansion of Extended Constraints

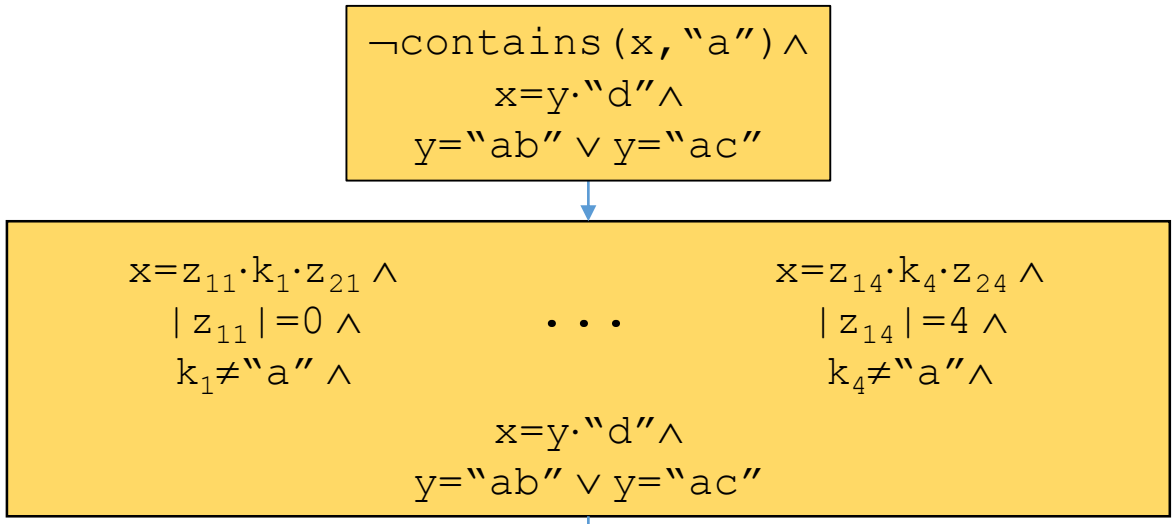
¬contains(x,“a”)∧
x=y·“d”∧
y=“ab” ∨ y=“ac”

SAT Solver

Arithmetic Solver

String Solver

(Eager) Expansion of Extended Constraints



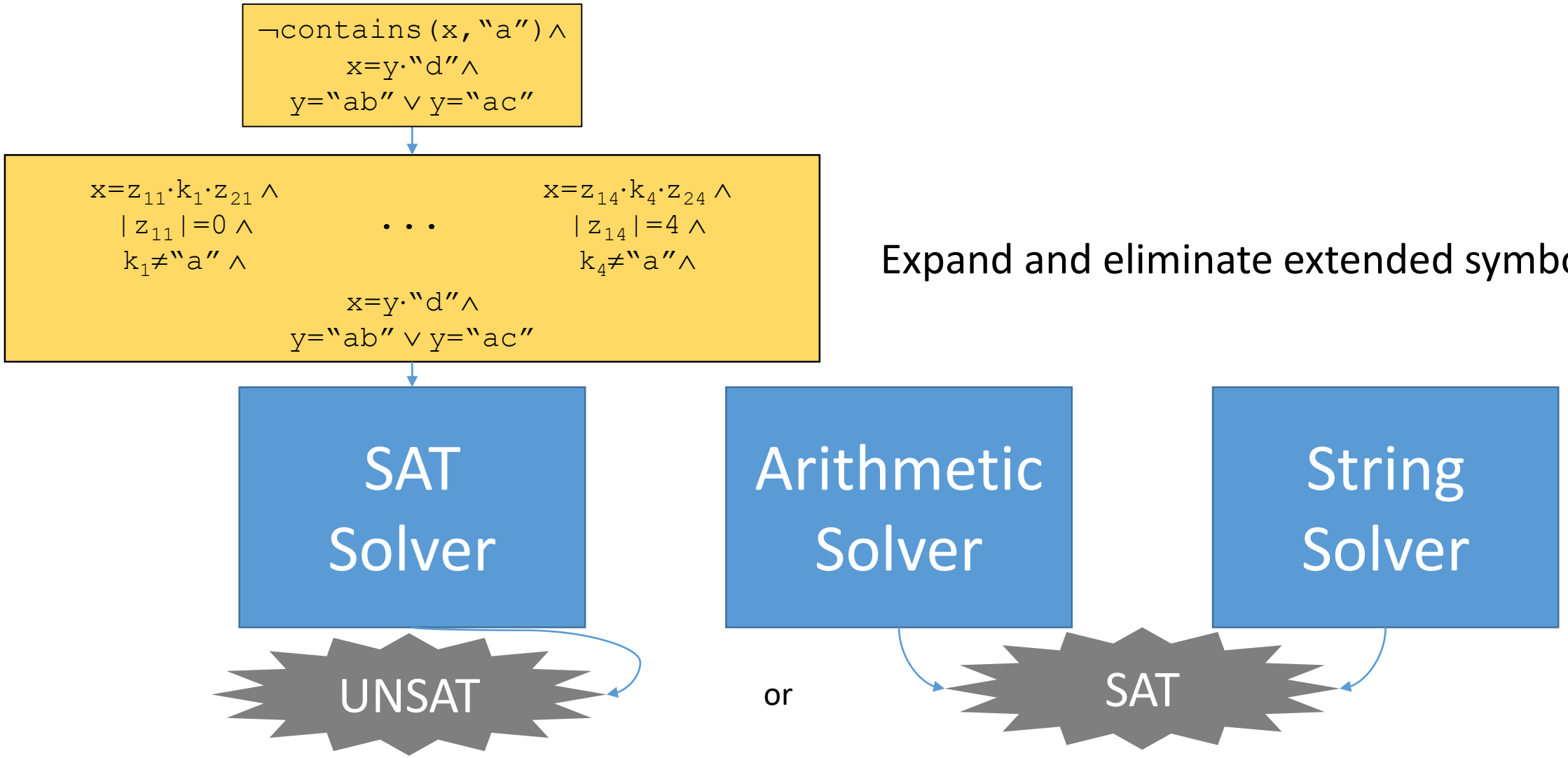
Expand and eliminate extended symbols

SAT Solver

Arithmetic Solver

String Solver

(Eager) Expansion of Extended Constraints



Expand and eliminate extended symbols

(Lazy) Expansion of Extended Constraints

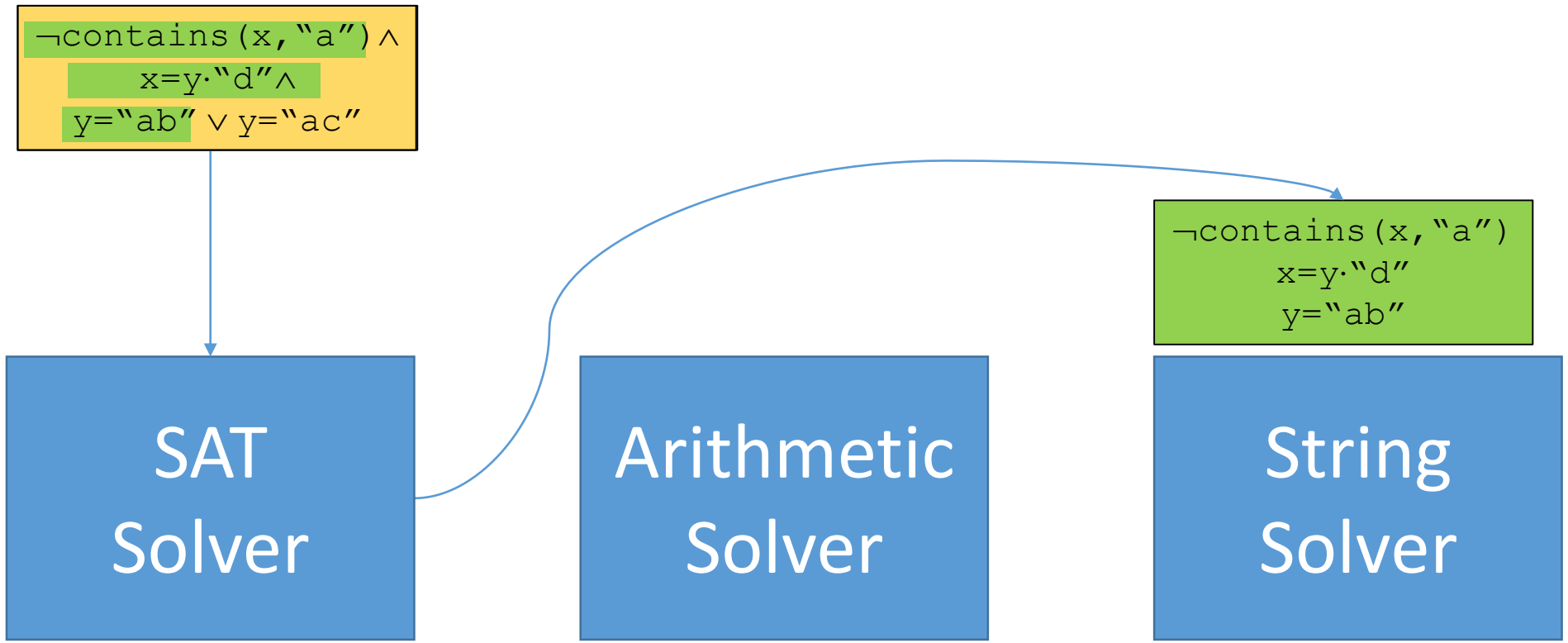
$\neg contains(x, \text{“a”}) \wedge$
$x = y \cdot \text{“d”} \wedge$
$y = \text{“ab”} \vee y = \text{“ac”}$

SAT Solver

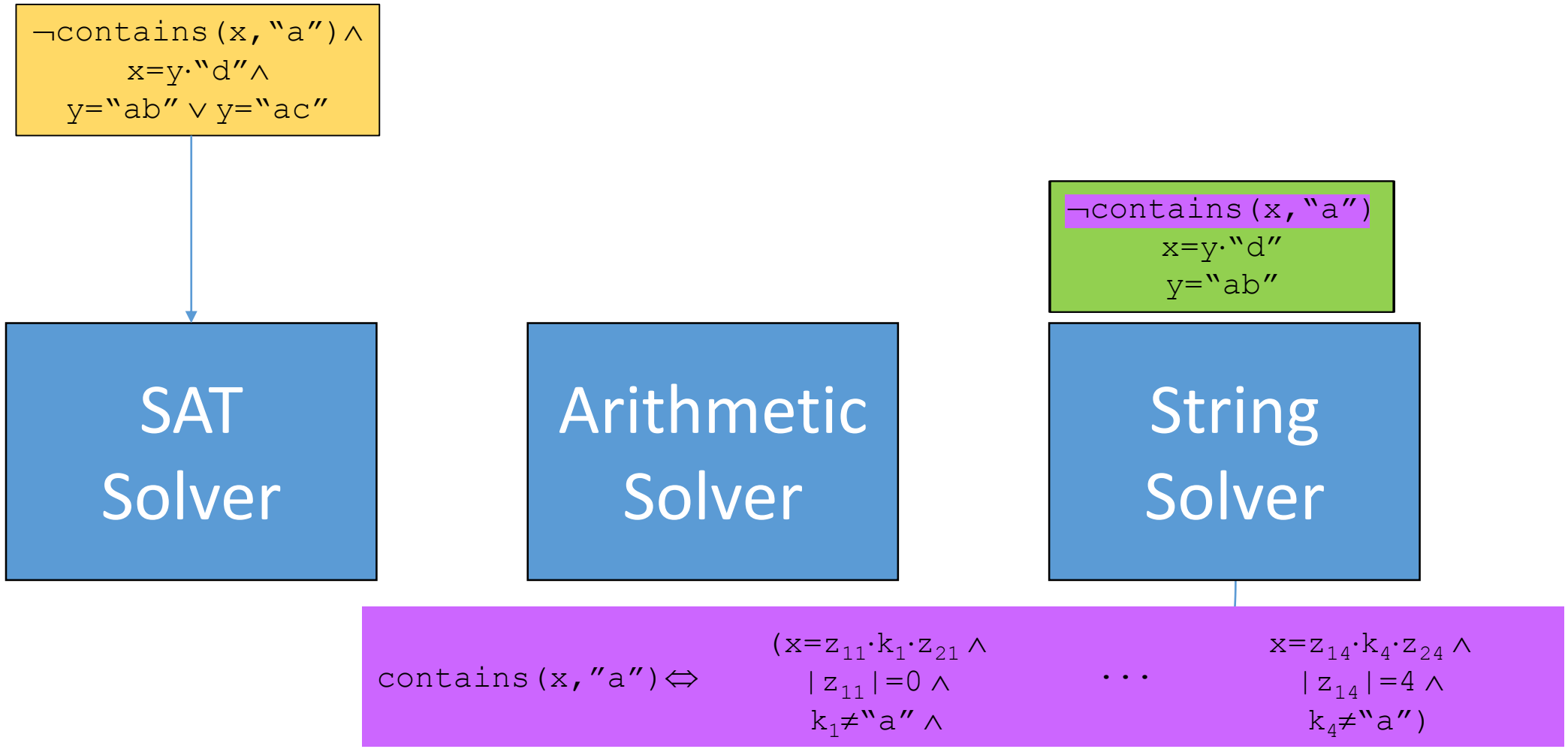
Arithmetic Solver

String Solver

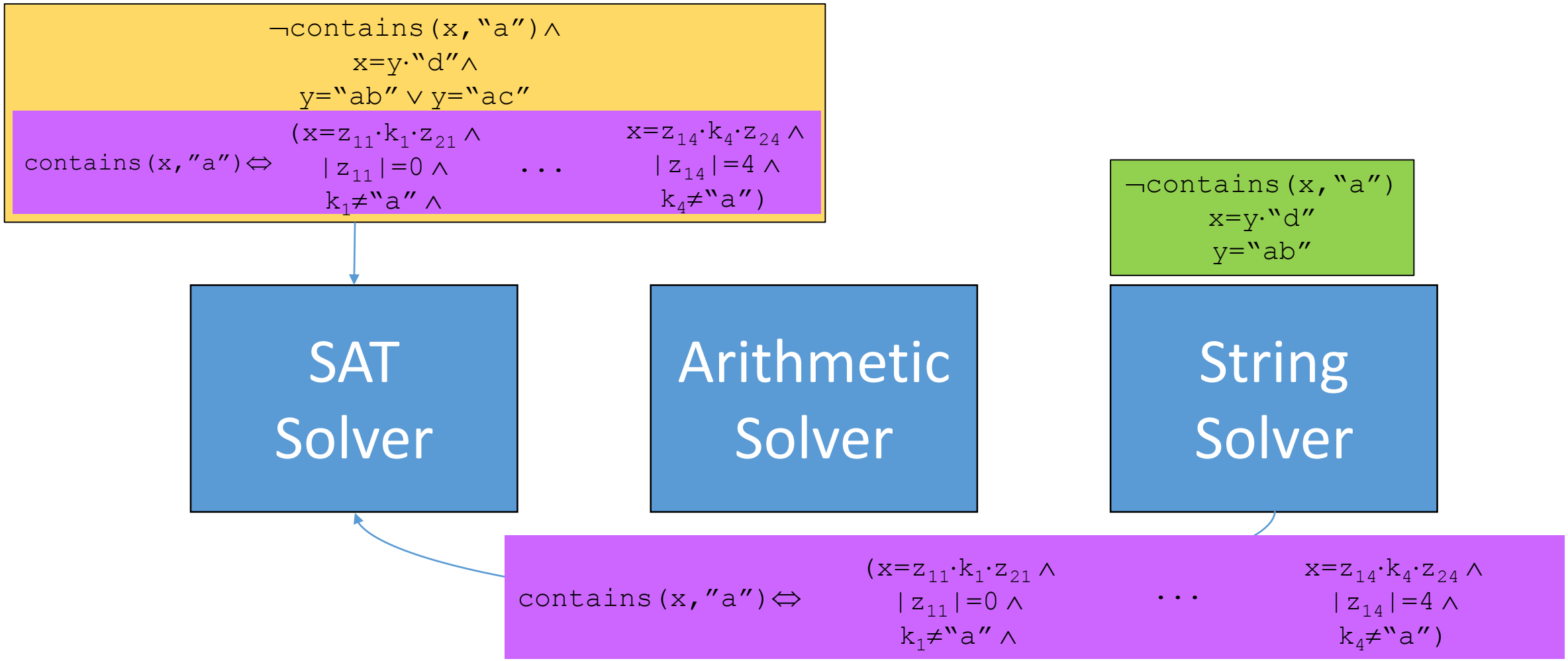
(Lazy) Expansion of Extended Constraints



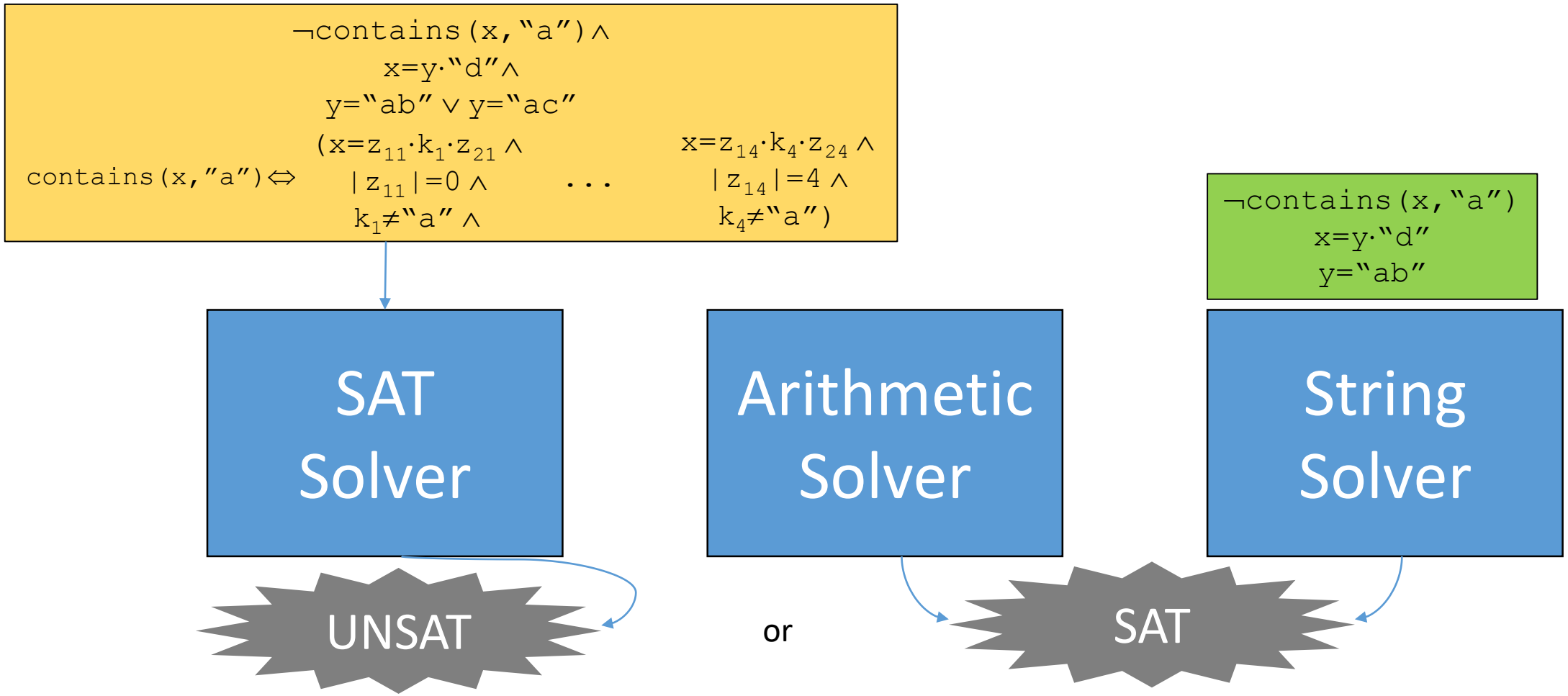
(Lazy) Expansion of Extended Constraints



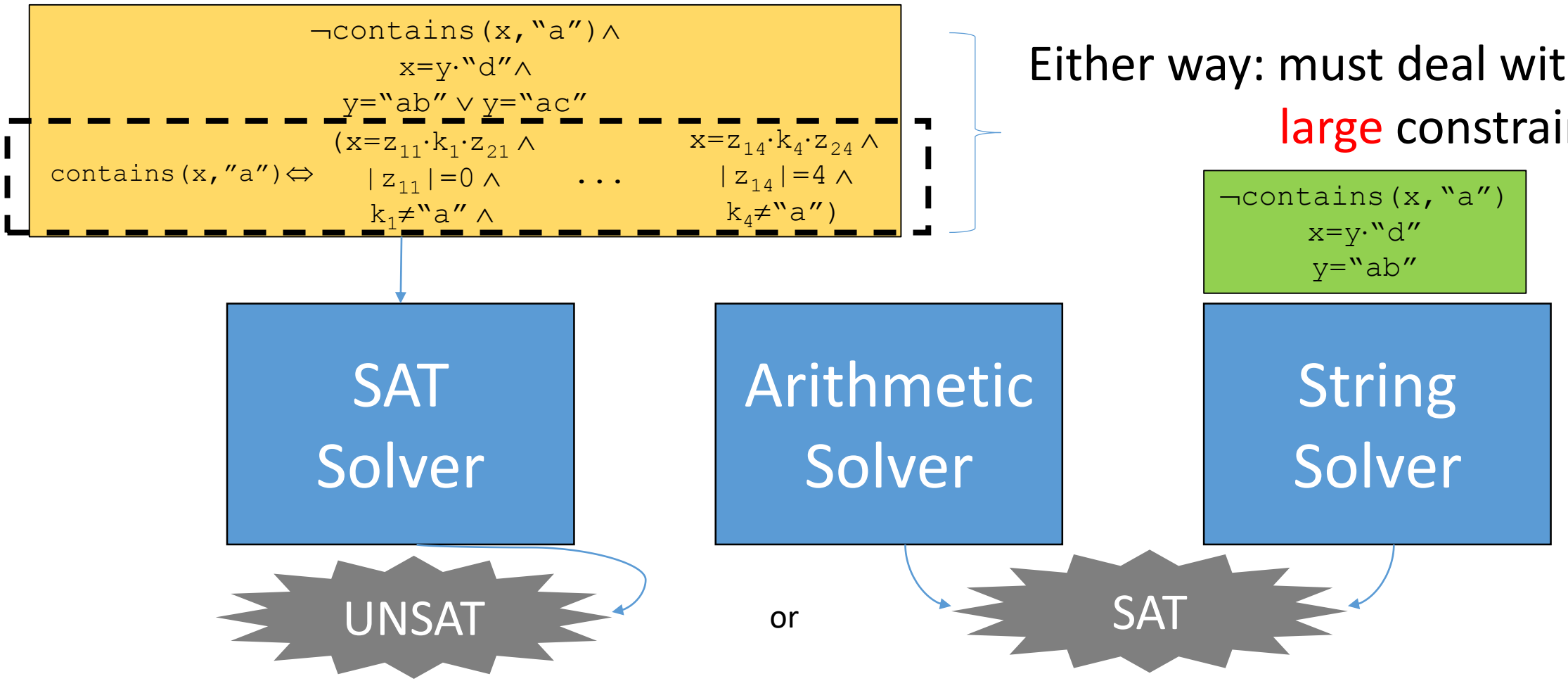
(Lazy) Expansion of Extended Constraints



(Lazy) Expansion of Extended Constraints



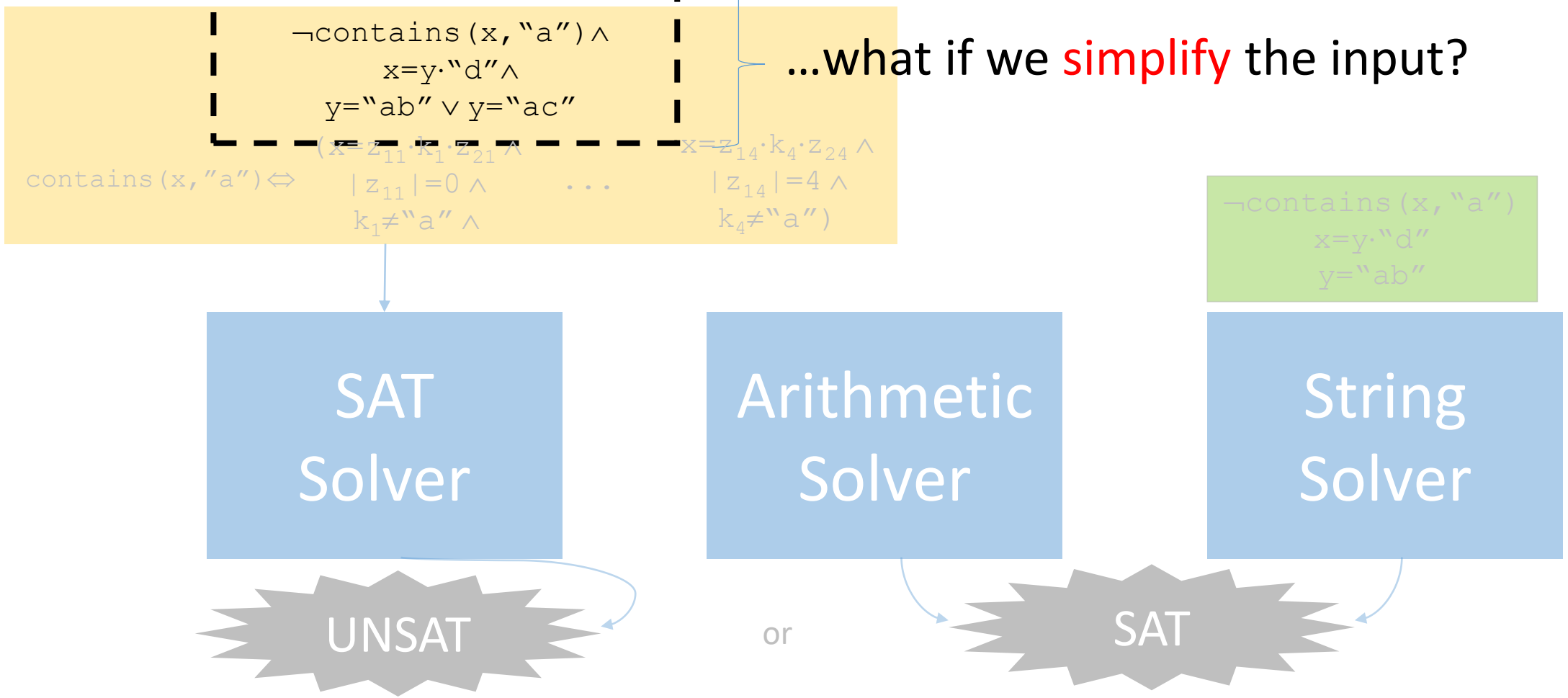
(Lazy) Expansion of Extended Constraints



(Lazy) Expansion of Extended Constraints

¬contains(x,"a")∧
x=y·"d"∧
y="ab" ∨ y="ac"

…what if we simplify the input?



SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

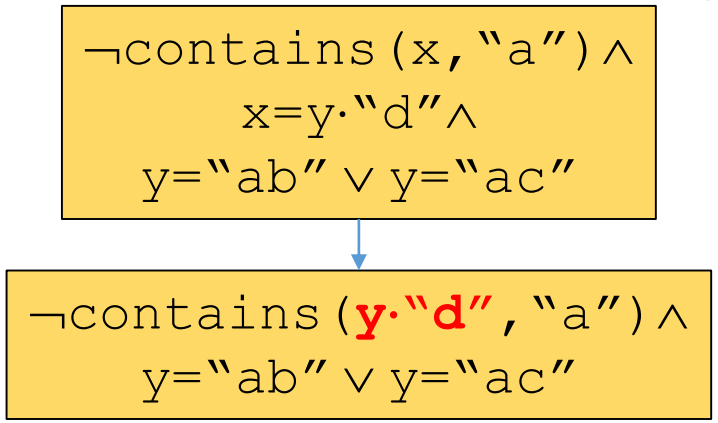
  (also called *normalization* or *rewrite* rules)

$$\neg\texttt{contains}(x,\text{“a”}) \wedge$$
$$x = y \cdot \text{“d”} \wedge$$
$$y = \text{“ab”} \vee y = \text{“ac”}$$

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

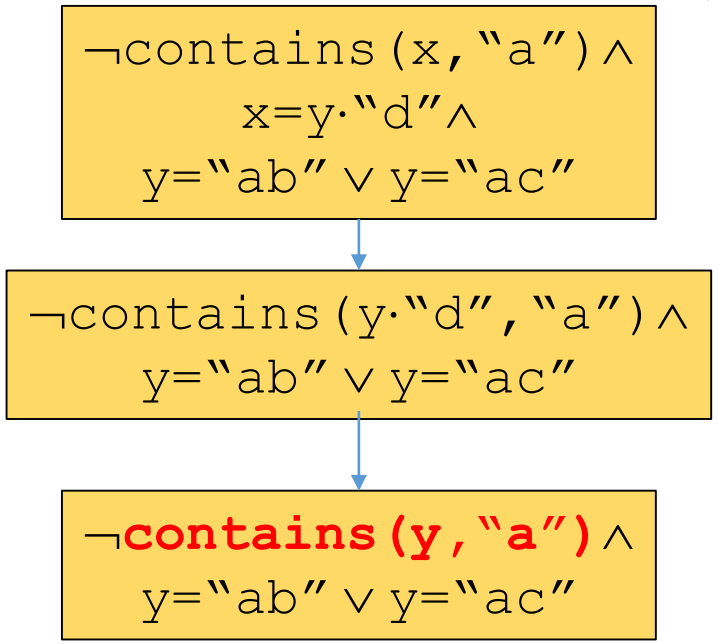
(also called *normalization* or *rewrite* rules)



since x=y·"d"

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques
  (also called *normalization* or *rewrite* rules)



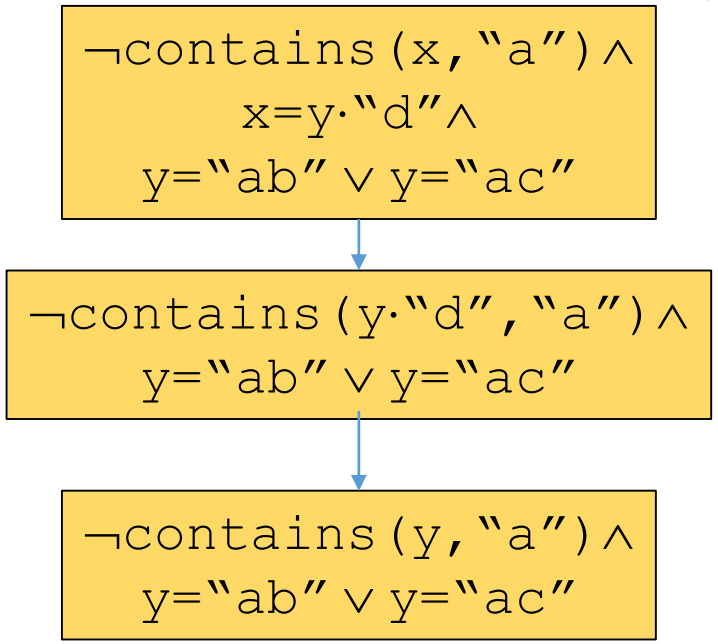
since `x=y·"d"`

since `contains(y·"d","a")⇔contains(y,"a")`

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

  (also called *normalization* or *rewrite* rules)



since `x=y·"d"`

since `contains(y·"d","a")⇔contains(y,"a")`

- Leads to smaller inputs, simpler procedures

(Lazy) Expansion + Simplification

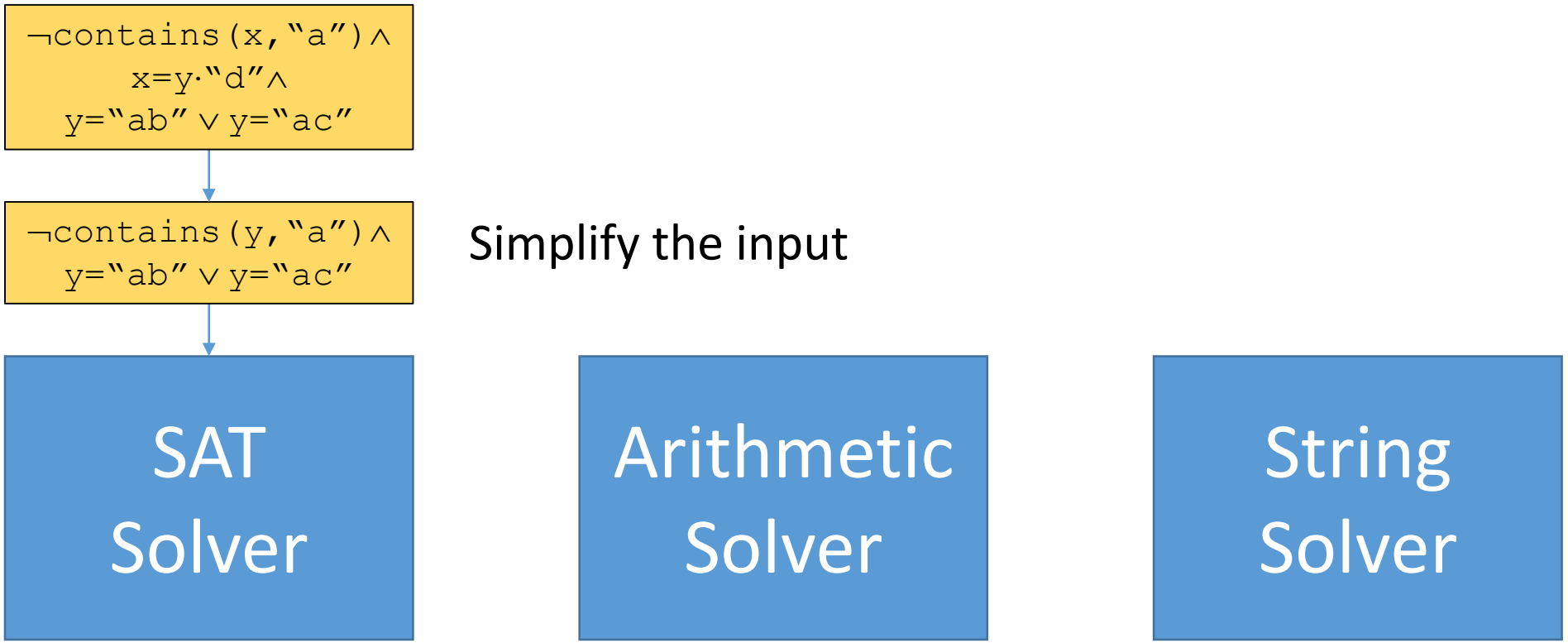
$\neg\text{contains}(x, \text{“a”}) \wedge$
$x = y \cdot \text{“d”} \wedge$
$y = \text{“ab”} \vee y = \text{“ac”}$

SAT Solver

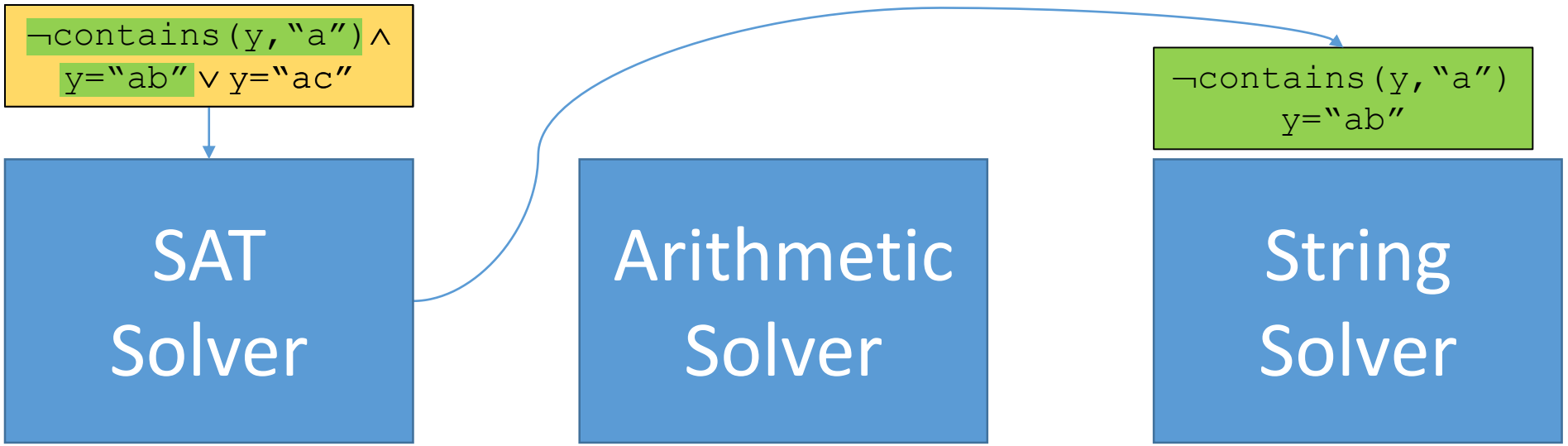
Arithmetic Solver

String Solver

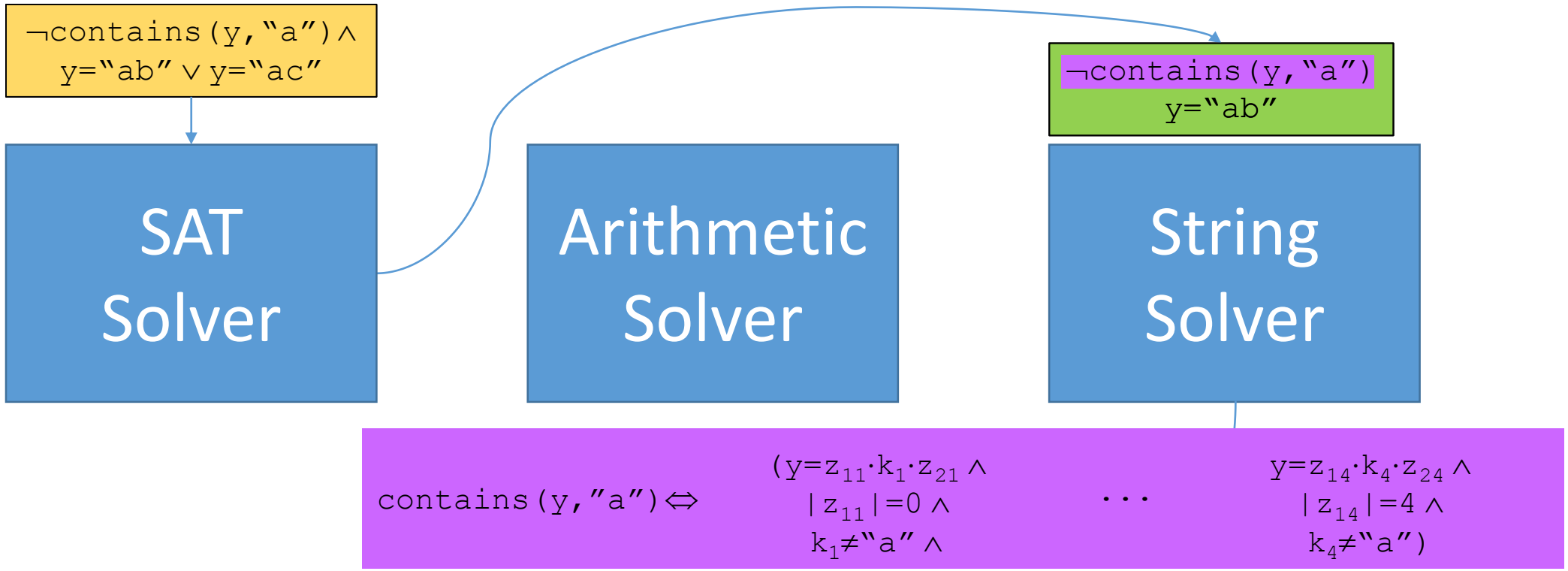
(Lazy) Expansion + Simplification



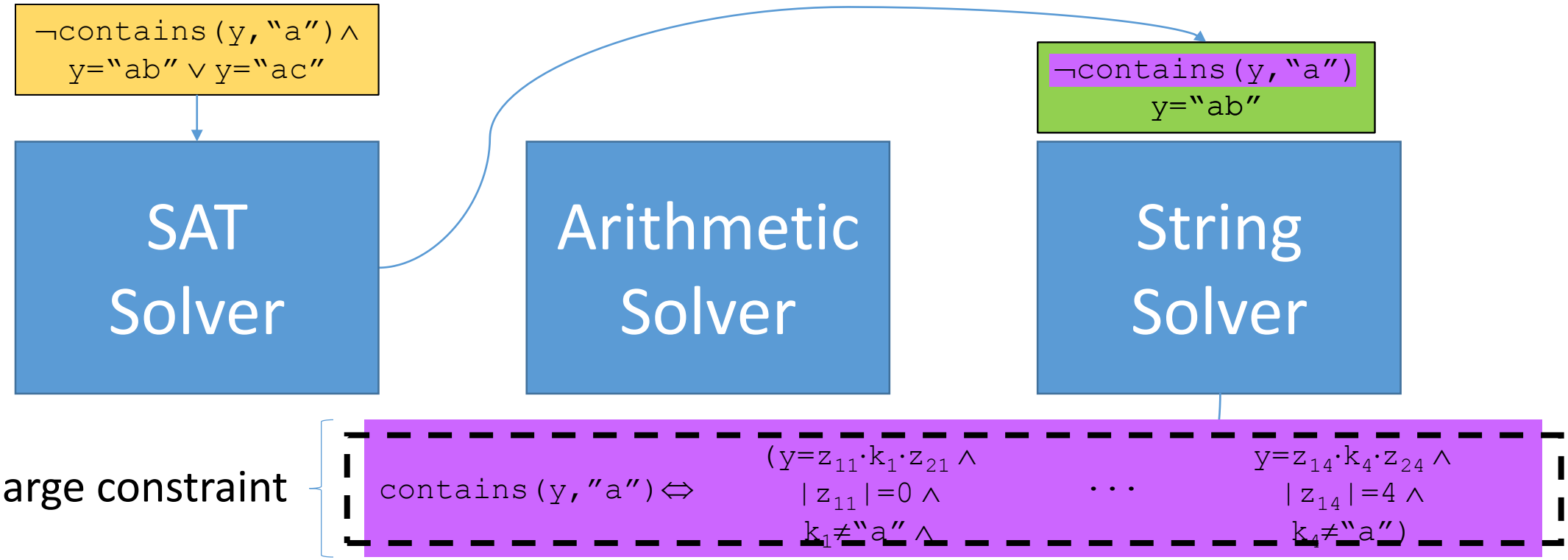
(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification

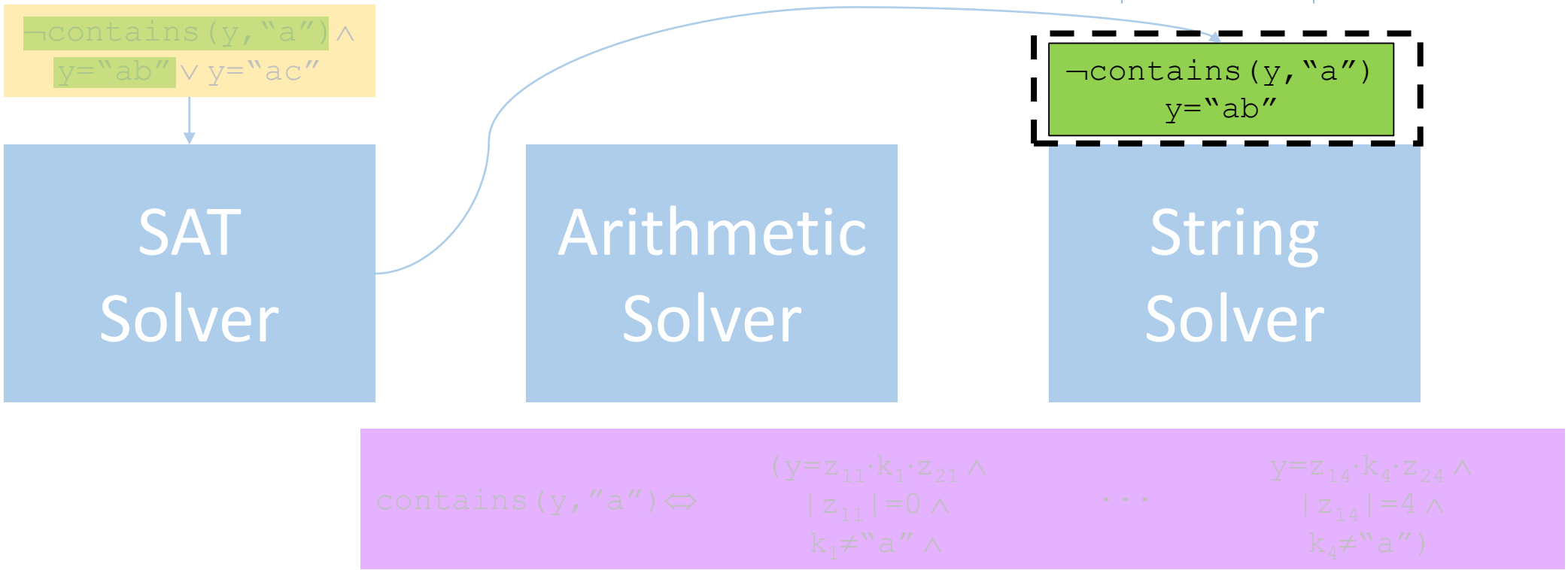


(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification

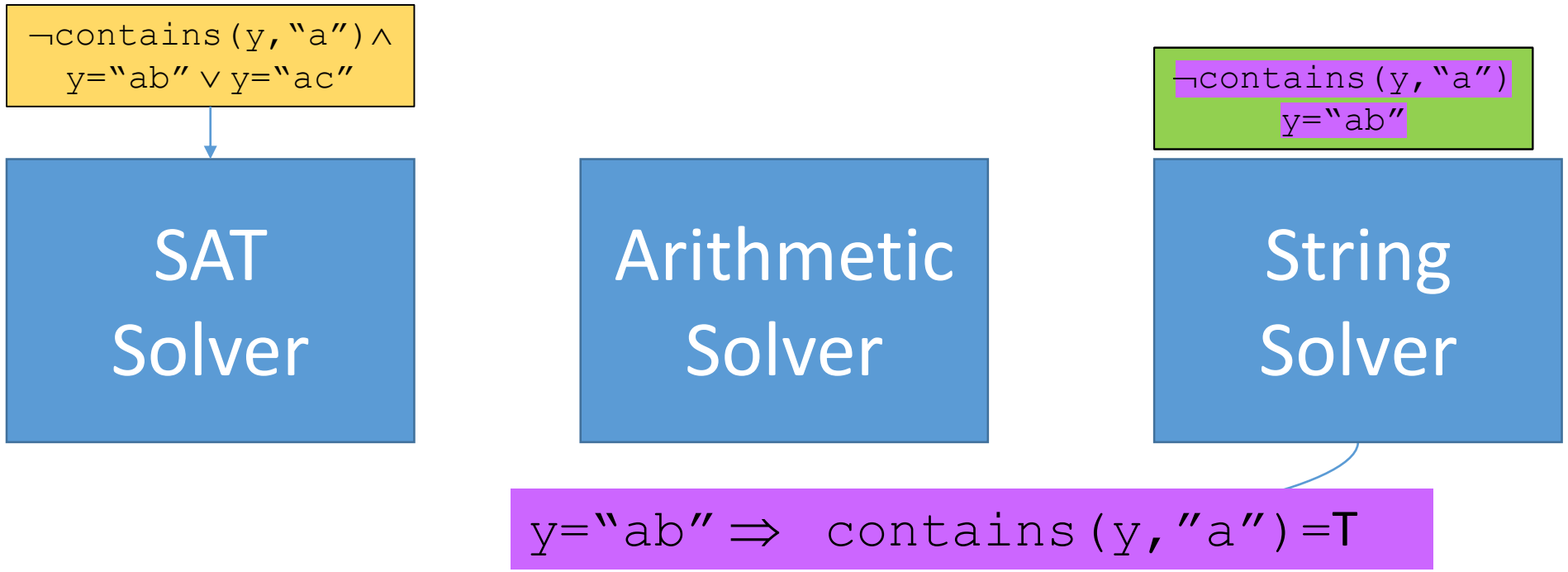
What if we simplify based on the context?



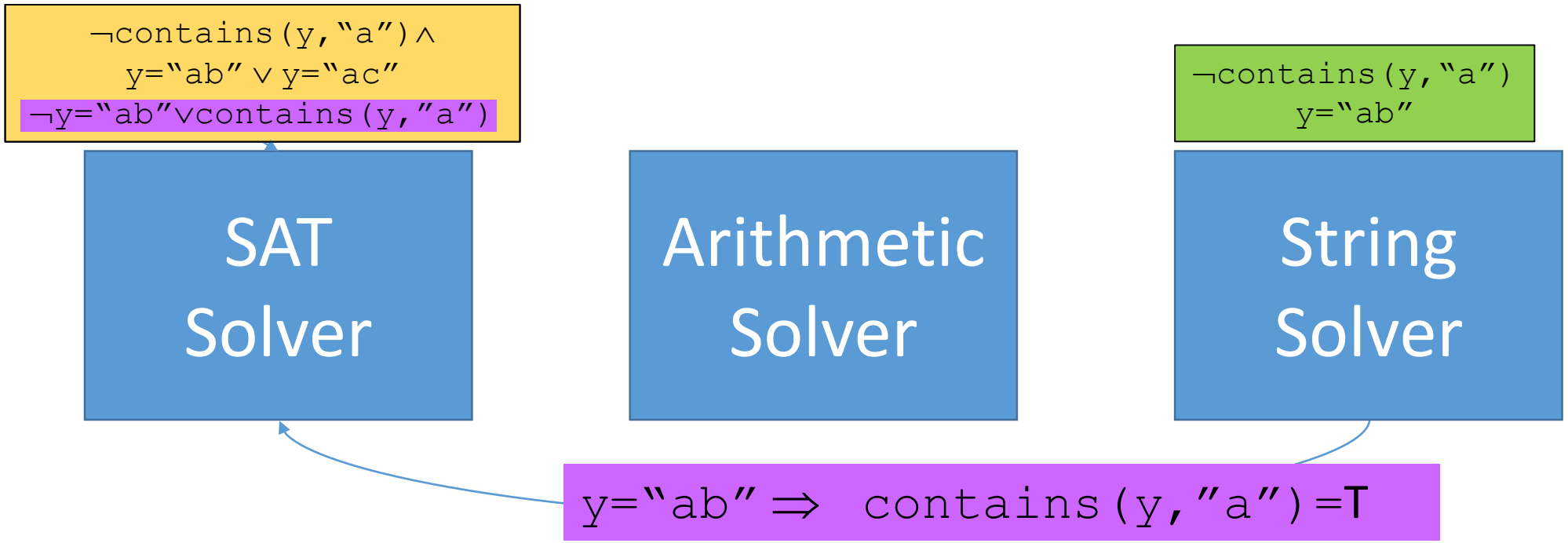
(Lazy) Expansion + Context-Dependent Simplification

Since `contains(y,"a")` is true when `y="ab"` …

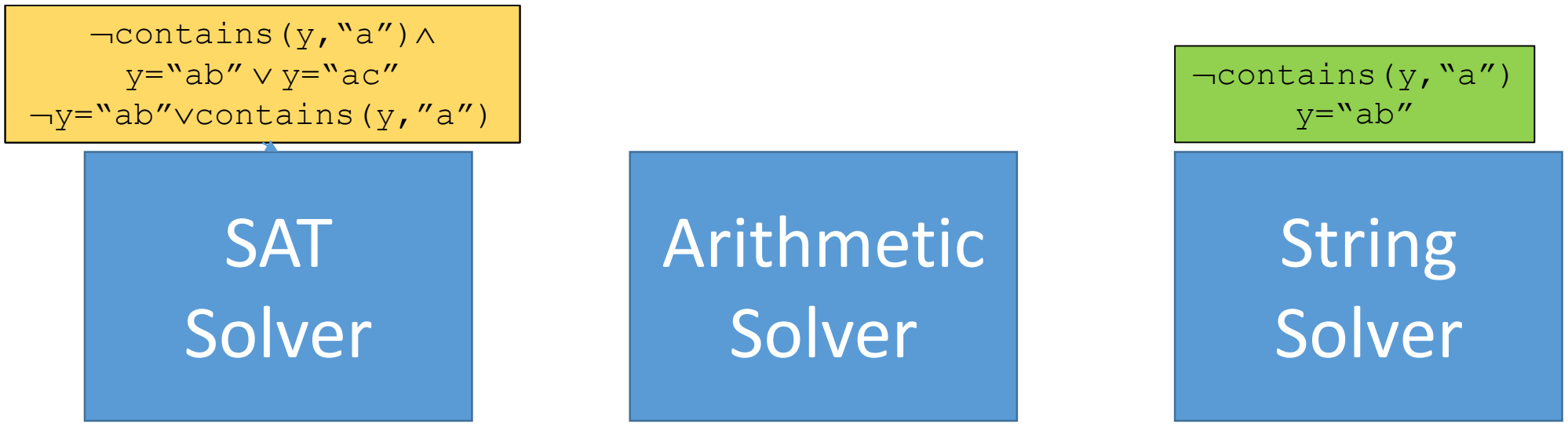
(Lazy) Expansion + Context-Dependent Simplification



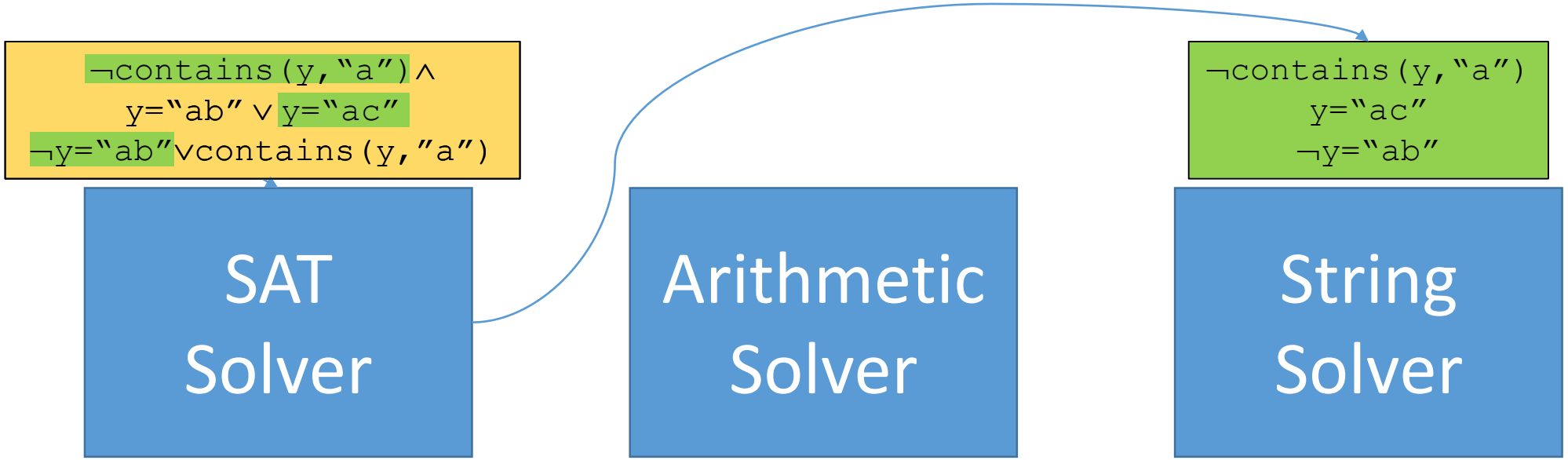
(Lazy) Expansion + Context-Dependent Simplification



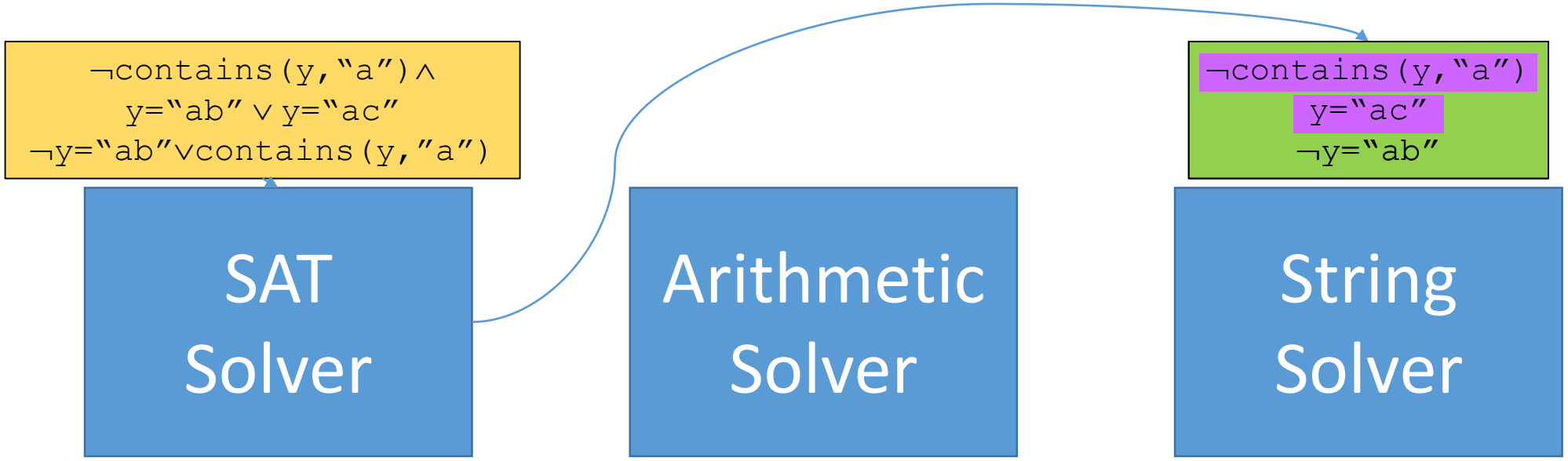
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification

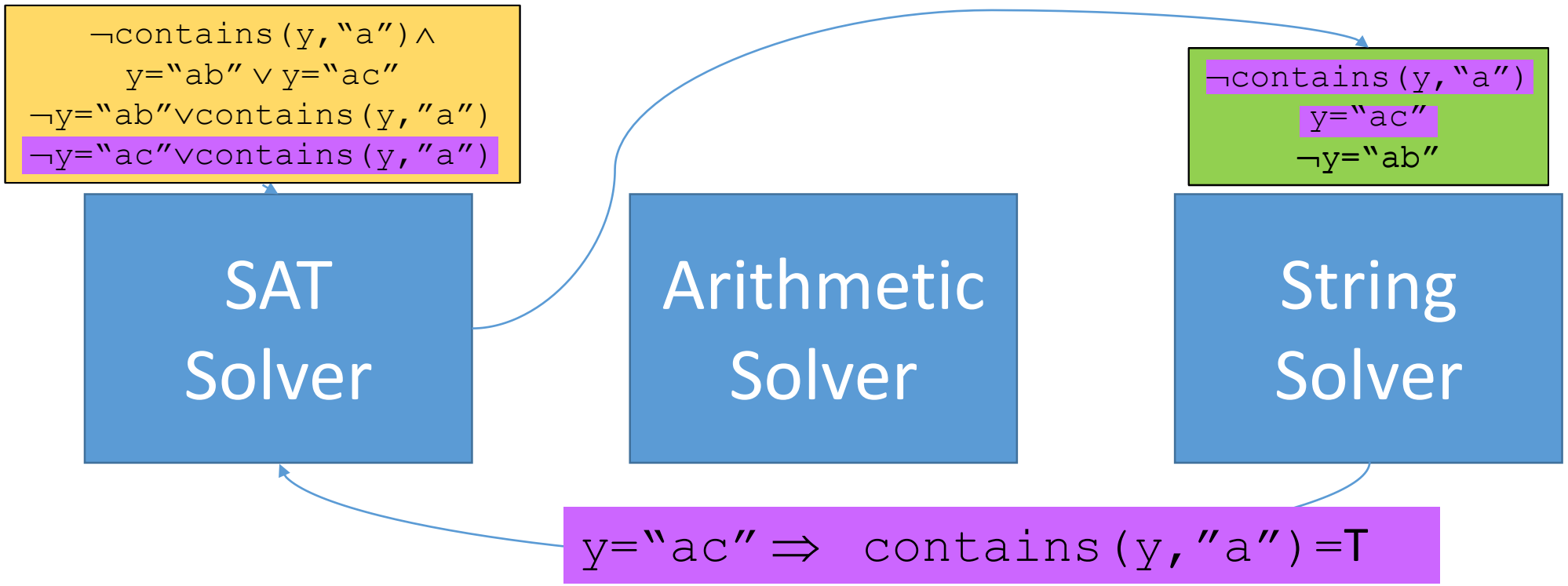


(Lazy) Expansion + Context-Dependent Simplification

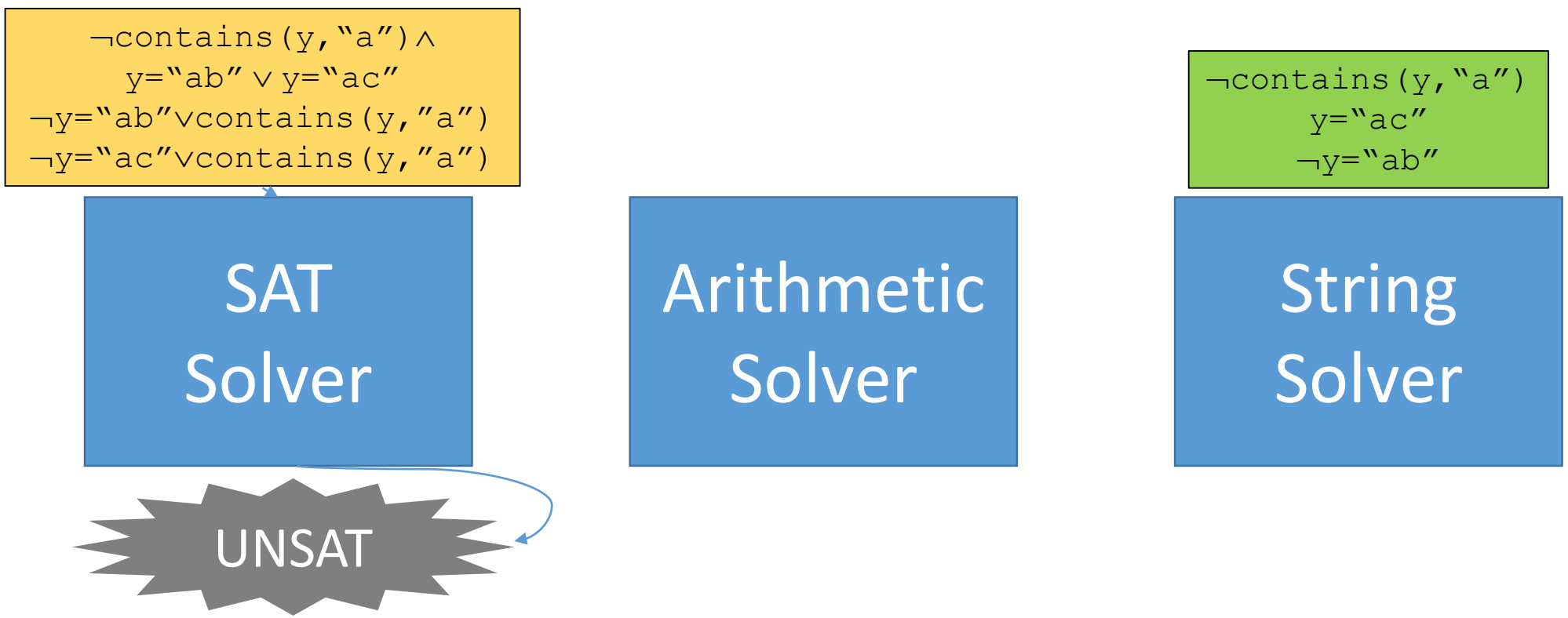


`contains(y,"a")` **is also true when** `y="ac"` **…**

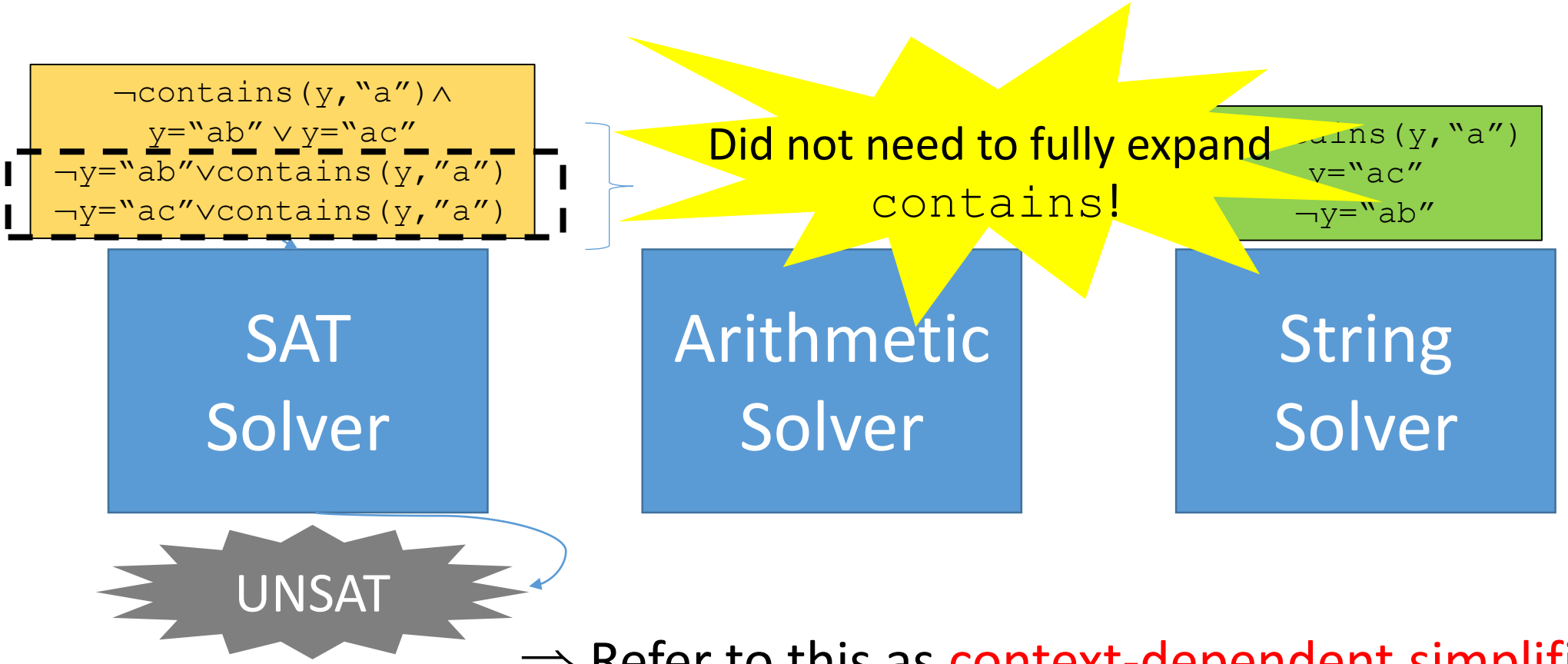
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification

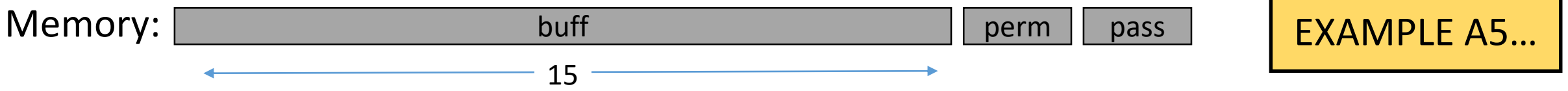


⇒ Refer to this as context-dependent simplification

Examples

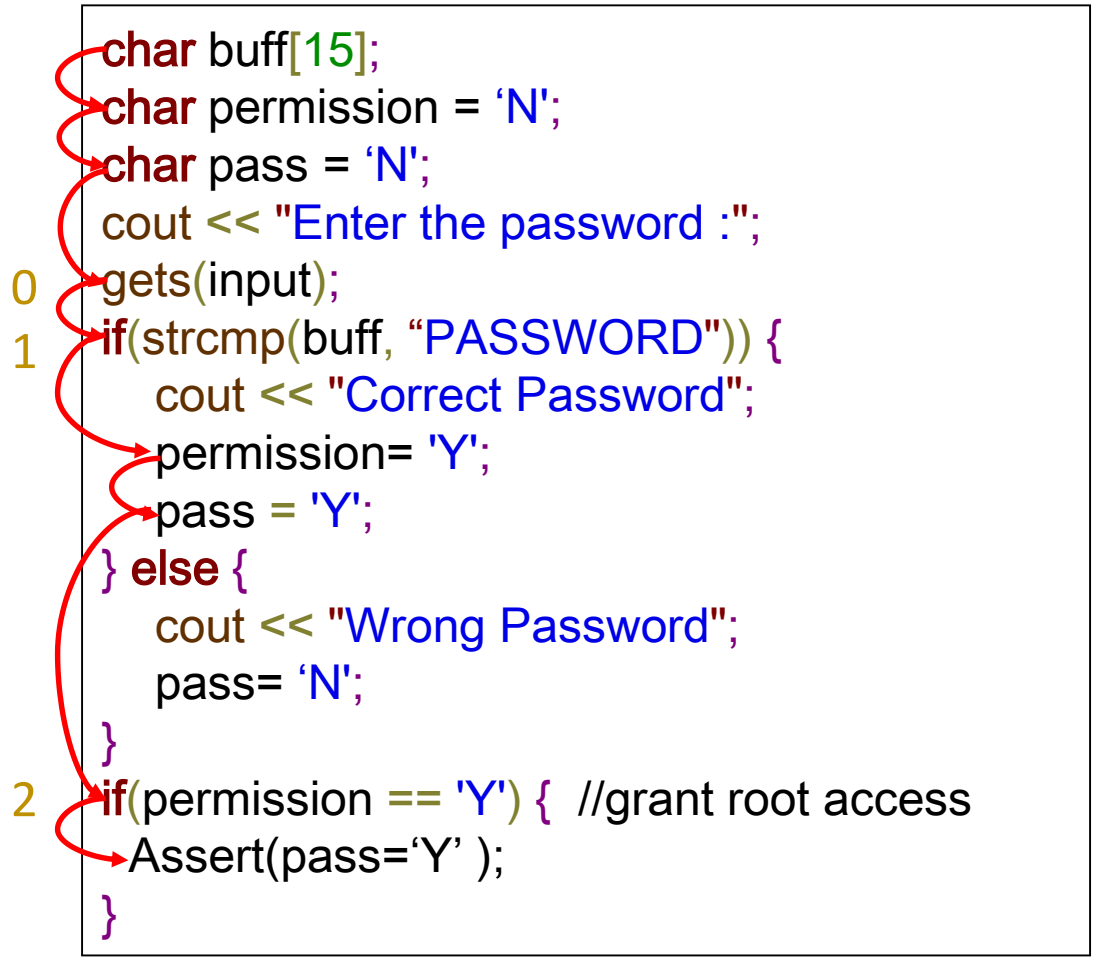
Symbolic Execution

```
char buff[15];
char permission = 'N';
char pass = 'N';
cout << "Enter the password :";
0  gets(input);
1  if(strcmp(buff, "PASSWORD")) {
       cout << "Correct Password";
       permission= 'Y';
       pass = 'Y';
   } else {
       cout << "Wrong Password";
       pass= 'N';
   }
2  if(permission == 'Y') {  //grant root access
     Assert(pass='Y' );
   }
```

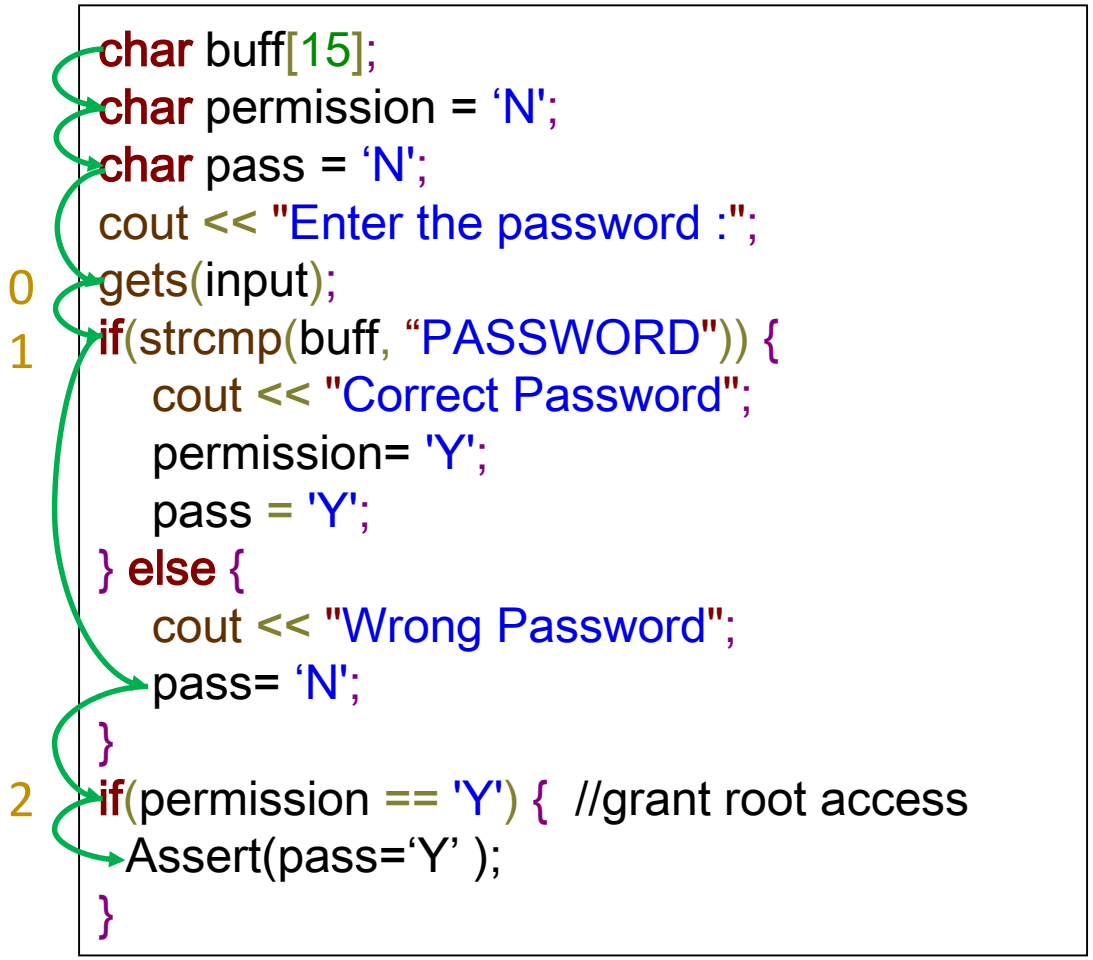


EXAMPLE A5…

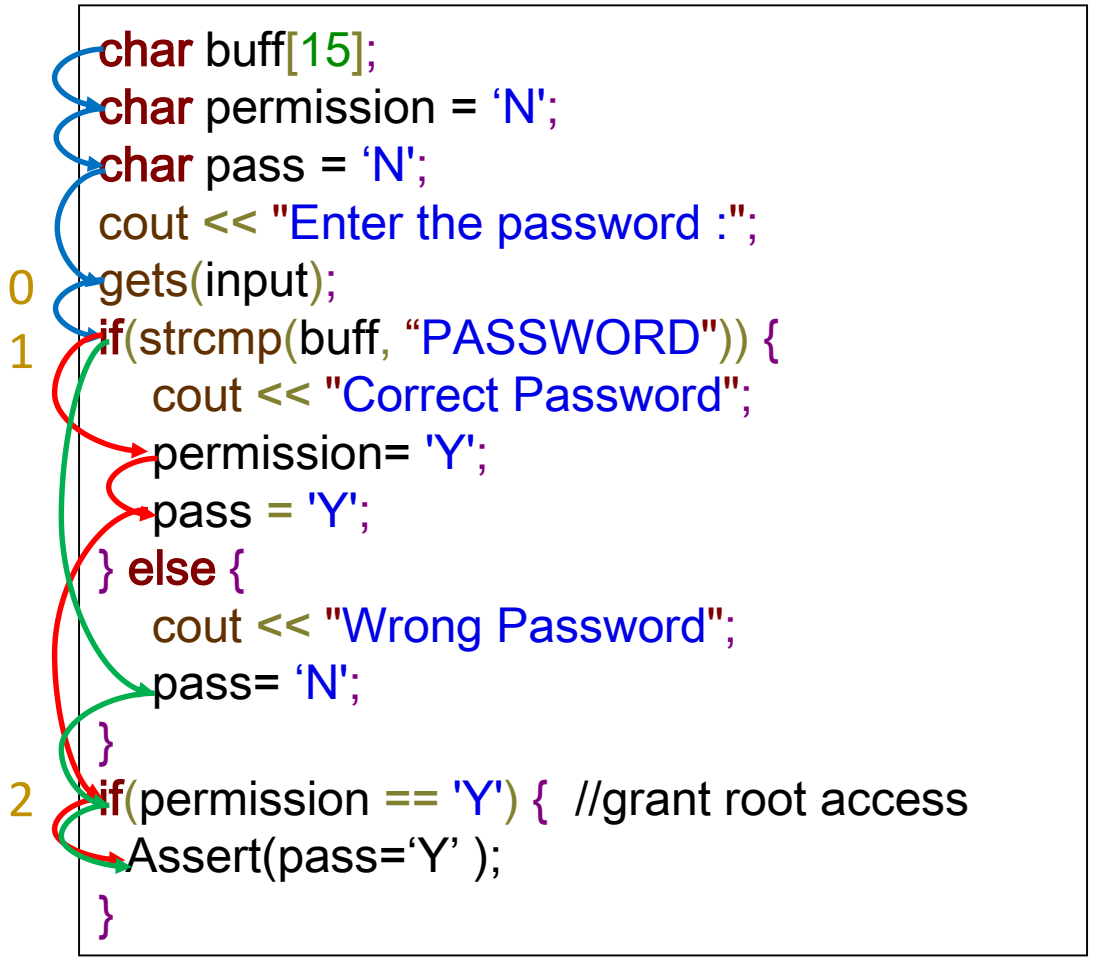
Symbolic Execution



Symbolic Execution



Symbolic Execution



EXAMPLE A5-inc…

Symbolic Execution

```
char buff[15];
char permission = 'N';
char pass = 'N';
cout << "Enter the password :";
gets(input);   ← "AAAAAAAAAAAAAAY"
if(strcmp(buff, "PASSWORD")) {
    cout << "Correct Password";
    permission= 'Y';
    pass = 'Y';
} else {
    cout << "Wrong Password";
    pass= 'N';
}
if(permission == 'Y') {  //grant root access
  Assert(pass='Y' );   ← pass=="N"
}
```

Symbolic Execution #2

```
cout << "Enter text to print :";
string input;
gets(input);
string data= ‘p‘ ++ input ++ ‘;xcvc4;‘;
int index=0;
while(index<str.len(data)){
  int end=indexof(data,’;’,index+1);
  string cmd=substr(data,index+1,end-(index+1));
  if(cmd!=“”){
    if(substr(data,index,1)=‘p’){
      cout << curr_cmd << endl;
    }else if(substr(data,index,1)=‘x’){
      exec(cmd);
      Assert(cmd==“cvc4”);
    }else{
      cout << “Bad command” << endl;
    }
  }
  index=end+1;
}
```

EXAMPLE A6…

Symbolic Execution #2

```
cout << "Enter text to print :";
string input;
gets(input);
string data= ‘p‘ ++ input ++ ‘;xcvc4;‘;
int index=0;
while(index<str.len(data)){
  int end=indexof(data,’;’,index+1);
  string cmd=substr(data,index+1,end-(index+1));
  if(cmd!=“”){
    if(substr(data,index,1)=‘p’){
      cout << curr_cmd << endl;
    }else if(substr(data,index,1)=‘x’){
      exec(cmd);
      Assert(cmd==“cvc4”);
    }else{
      cout << “Bad command” << endl;
    }
  }
  index=end+1;
}
```

Expects data to be a string like:

p[TEXT1];x[PROG1];x[PROG2];p[TEXT2];…

where

- pTEXT; prints “TEXT”
- xPROG; runs program “PROG”

Symbolic Execution #2

```
cout << "Enter text to print :";
string input;
gets(input);   ← ";xATTACK"
string data= 'p' ++ input ++ ';xcvc4;';
int index=0;
while(index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  if(cmd!=""){
    if(substr(data,index,1)='p'){
      cout << curr_cmd << endl;
    }else if(substr(data,index,1)='x'){
      exec(cmd);   ← exec("ATTACK")
      Assert(cmd=="cvc4");
    }else{
      cout << "Bad command" << endl;
    }
  }
  index=end+1;
}
```

Symbolic Execution #3

```
cout << "Enter text to print :";
string input;
char setup='N';
gets(input);
string data= 'p' ++ input ++ ';xi;xroot';
int index=0;
while(0≤index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)='p'){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)='g'){
    index := str.to.int(index);
  }else if(substr(data,index,1)='x'){
    exec(cmd);
    if(cmd=="setup"){
      setup='Y';
    }else if(cmd="root"){
     Assert(setup='Y');
    }
  }
}
```

start index

'p' ++ input ++ ';xsetup;xroot';

- pTEXT; prints "TEXT"
- xPROG; runs program "PROG"
  - xsetup; sets setup='Y'
- gN; goes to index N

EXAMPLE A7…

Symbolic Execution #3

```
cout << "Enter text to print :";
string input;
char setup='N';
gets(input);
if(contains(input,'x')){
  exit();
}
string data= 'p' ++ input ++ ';xi;xroot';
int index=0;
while(0≤index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)='p'){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)='g'){
    index := str.to.int(index);
  }else if(substr(data,index,1)='x'){
    exec(cmd);
    if(cmd=="setup"){
      setup='Y';
    }else if(cmd="root"){
     Assert(setup='Y');
    }
  }
}
```

Symbolic Execution #3

```
cout << "Enter text to print :";
string input;
char setup='N';
gets(input);
if(contains(input,'x')){
  exit();
}
string data= 'p' ++ input ++ ';xi;xroot';
int index=0;
while(0≤index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

start index

↓

'p' ++ input ++ ';xsetup;xroot';

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)='p'){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)='g'){
    index := str.to.int(index);
  }else if(substr(data,index,1)='x'){
    exec(cmd);
    if(cmd=="setup"){
      setup='Y';
    }else if(cmd="root"){
     Assert(setup='Y');
    }
  }
}
```

- pTEXT; prints "TEXT"
- xPROG; runs program "PROG"
  - xsetup; sets setup='Y'
- gN; goes to index N

Symbolic Execution #3

```
cout << "Enter text to print :";
string input;     ← ";g13"
char setup='N';
gets(input);
if(contains(input,'x')){
  exit();
}
string data= 'p' ++ input ++ ';xi;xroot';
int index=0;
while(0≤index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)='p'){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)='g'){
    index := str.to.int(index);
  }else if(substr(data,index,1)='x'){
    exec(cmd);
    if(cmd=="setup"){
      setup='Y';
    }else if(cmd="root"){
     Assert(setup='Y');
    }
  }
}
```

'p;g13;xsetup;xroot;';