Solving Verification Conditions using SMT Solvers

Andrew Reynolds

MOVEP Summer School

June 26, 2020

THE UNIVERSITY OF IOWA

Satisfiability Modulo Theories (SMT) solvers

- *Useful tools* for
  - Verification
  - Interactive theorem proving
  - Symbolic execution
  - Synthesis
  - …(your application here)
- Examples of current SMT solvers:
  - Z3, CVC4, Yices2, Boolector, MathSAT, VeriT

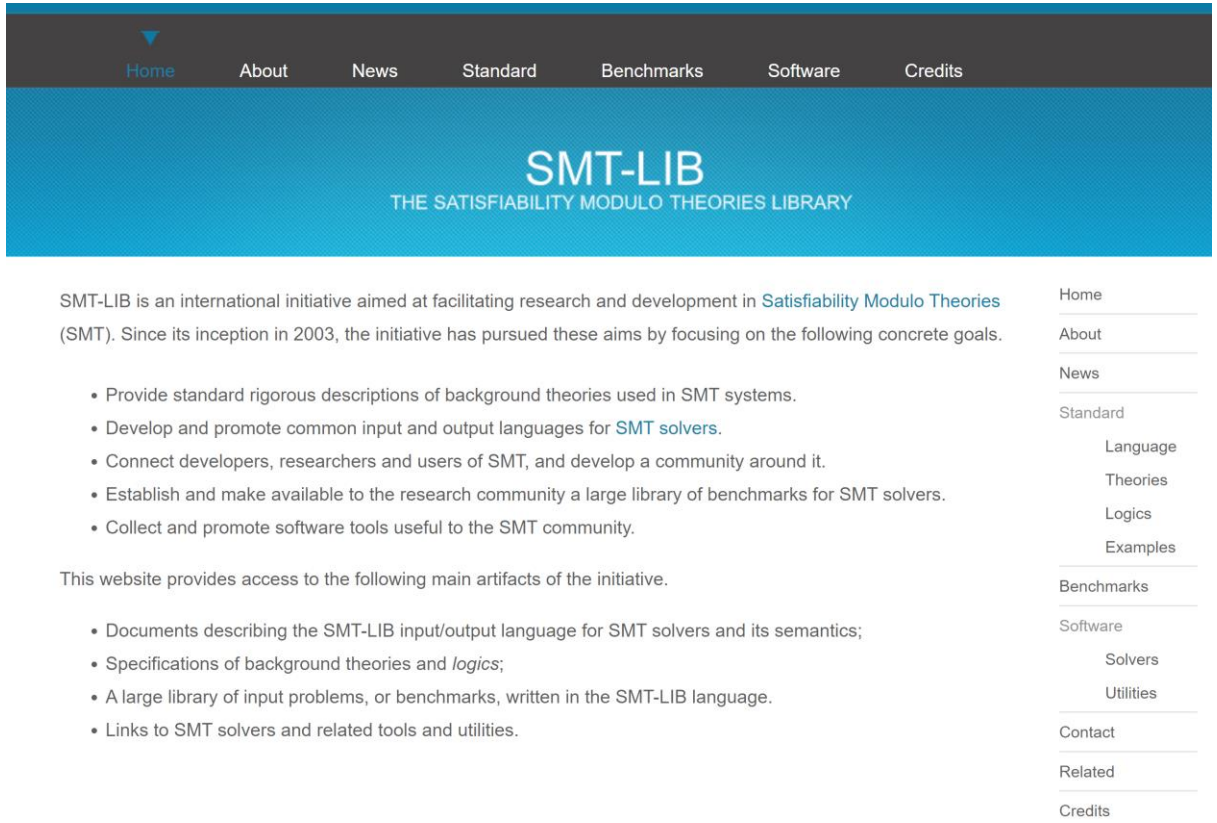
    $\Rightarrow$ CVC4 SMT solver

Acknowledgements

- Thanks to development team of CVC4:
  - Clark Barrett, Cesare Tinelli, Haniel Barbosa, Andres Noetzli, Aina Niemetz, Mathias Preiner, Martin Brain, Ahmed Irfan, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Alex Ozdemir, Ying Sheng, Yoni Zohar
- Past members:
  - Morgan Deters, Dejan Jovanovic, Tim King, Guy Katz, Liana Hadarean, Kshitij Bansal, Tianyi Liang, Nestan Tsiskardidze, Christopher Conway, Francois Bobot, Paul Meng

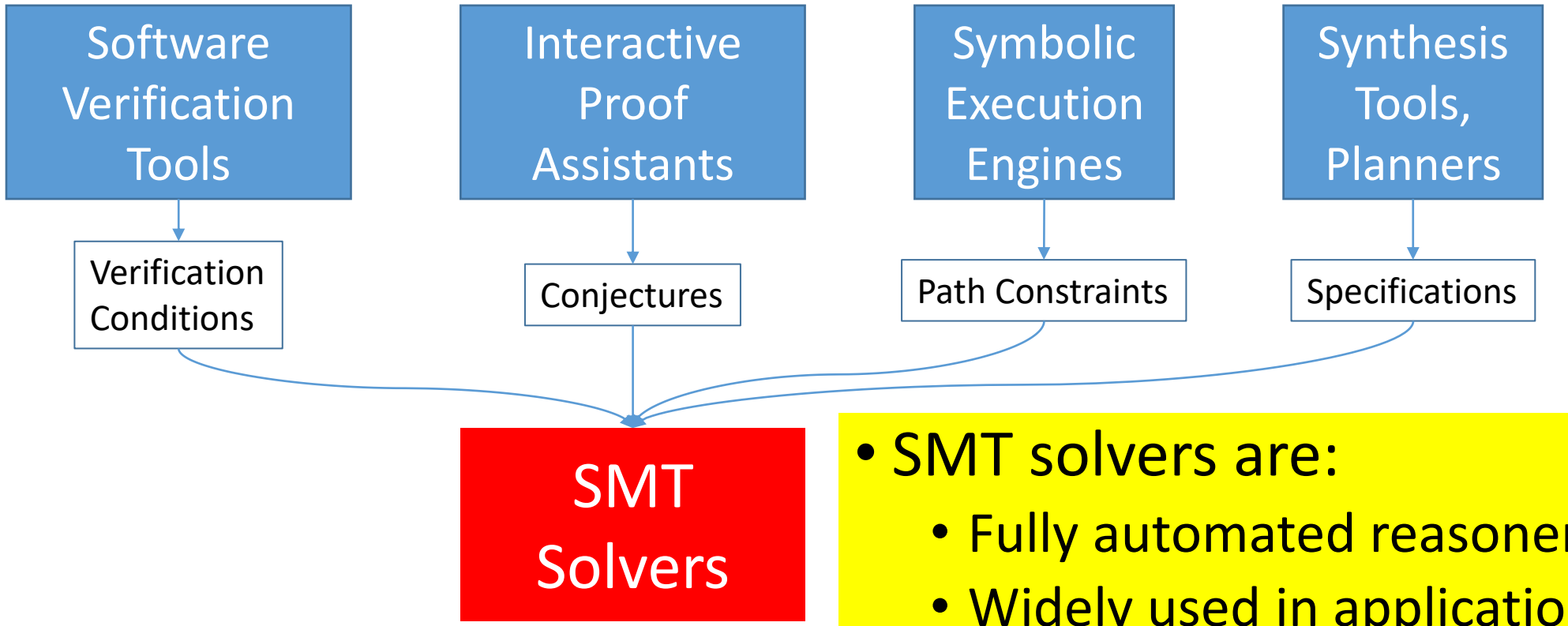
Satisfiability Modulo Theories: Resources

- SMT-LIB (http://smt-lib.org/)
  - Theories
  - Logics
  - Options
  - Other language features
  - Large library of benchmarks



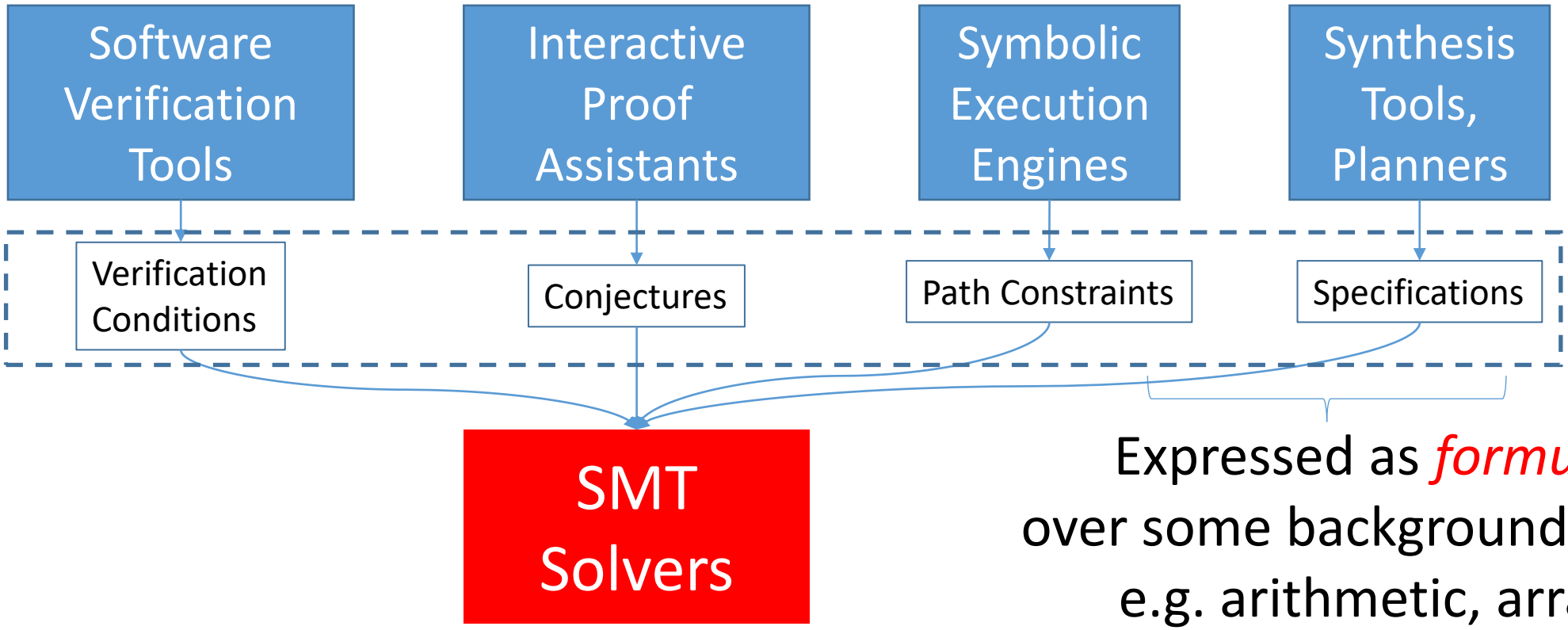
- SMT-COMP (https://smt-comp.github.io/2020/)
  - Yearly competition of SMT solvers

Satisfiability Modulo Theories (SMT) Solvers



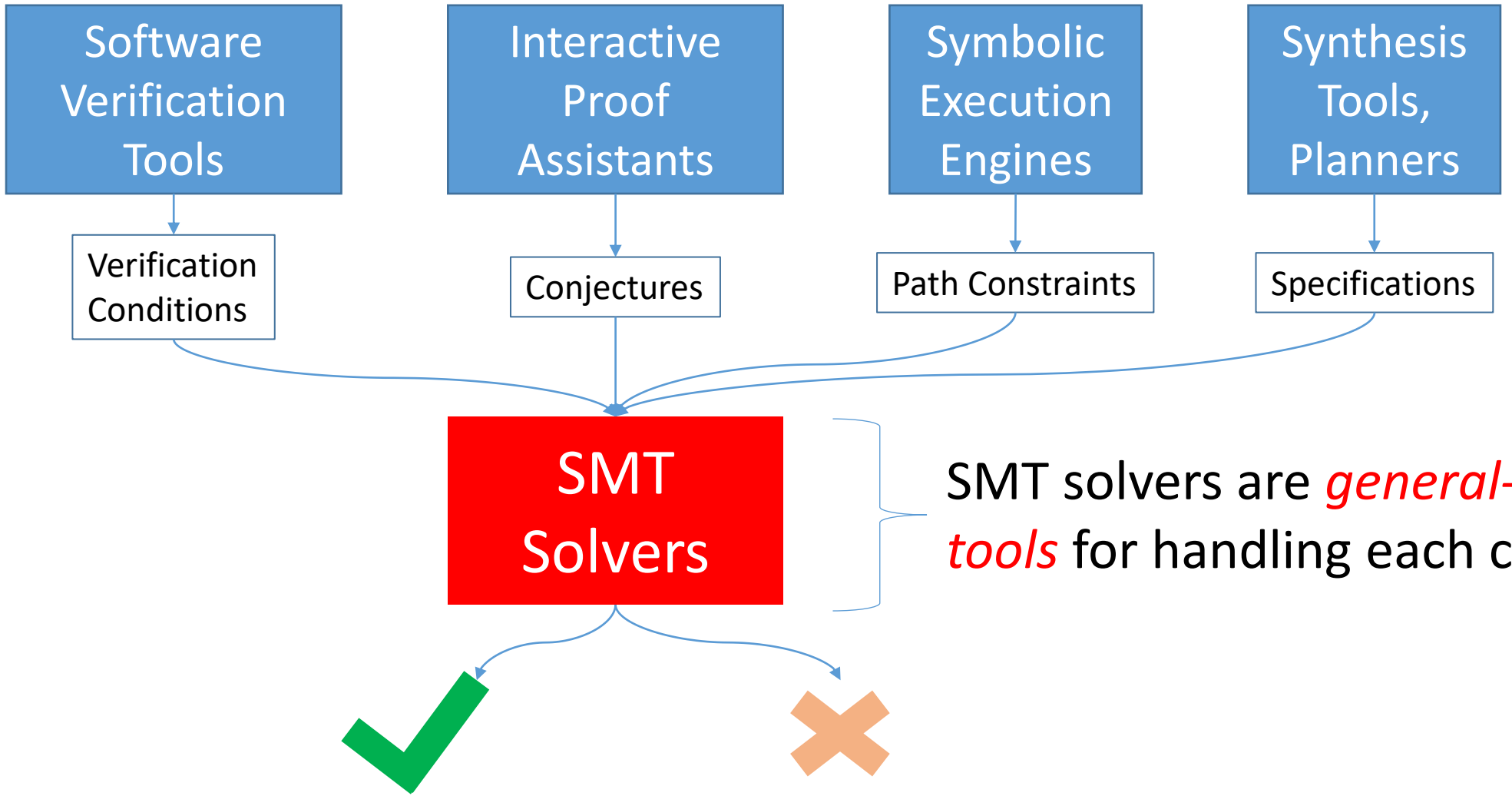
- SMT solvers are:
  - Fully automated reasoners
  - Widely used in applications

Satisfiability Modulo Theories (SMT) Solvers



Expressed as *formulas*
over some background *theory*
e.g. arithmetic, arrays

Satisfiability Modulo Theories (SMT) Solvers



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

Contract-Based Software Verification

```
@precondition: x_in>y_in
void swap(int x, int y)
{
      x := x + y;
      y := x - y;
      x := x - y;
}
```

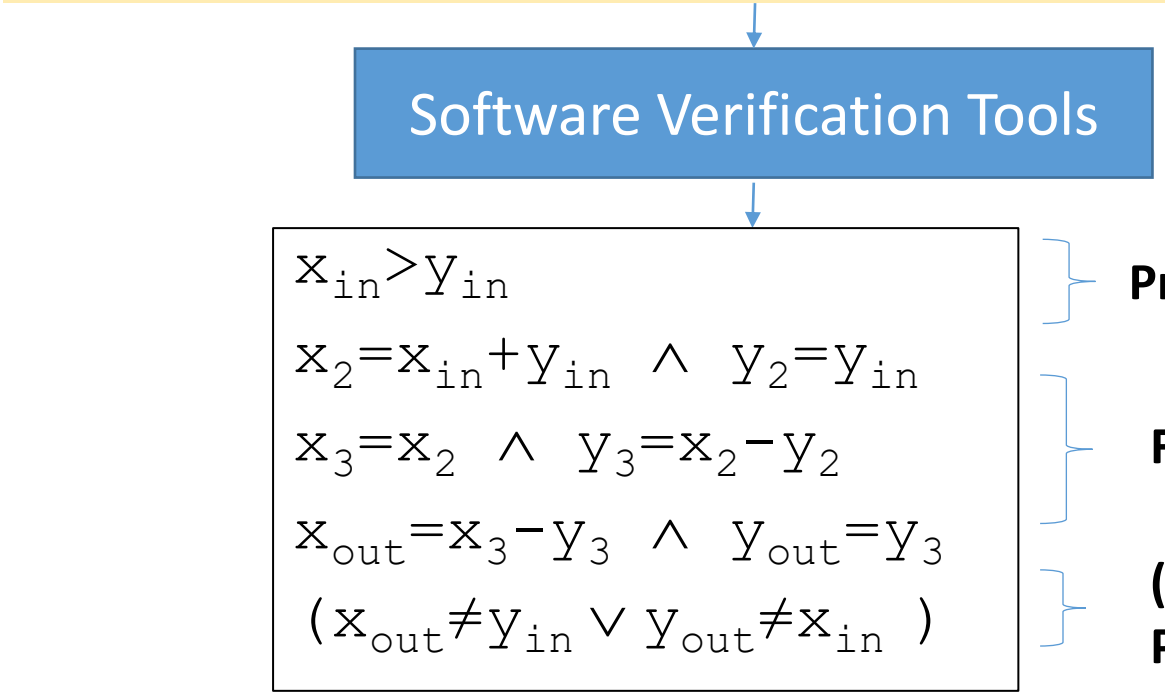
…does this function ensure that $x_{out}=y_{in} \wedge y_{out}=x_{in}$?

Software Verification Tools

Contract-Based Software Verification

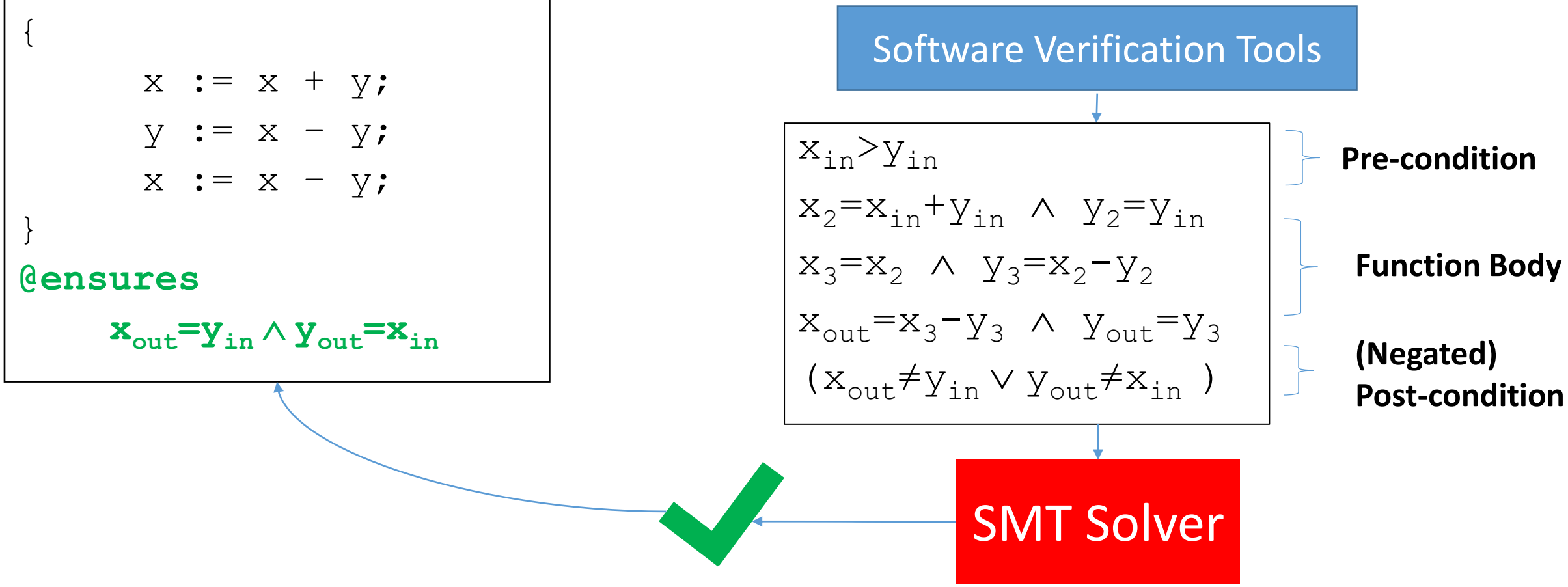
```
@precondition: x_in>y_in
void swap(int x, int y)
{
      x := x + y;
      y := x - y;
      x := x - y;
}
```

…does this function ensure that $x_{out}=y_{in} \wedge y_{out}=x_{in}$?



Contract-Based Software Verification

```
@precondition: x_in>y_in
void swap(int x, int y)
{
        x := x + y;
        y := x - y;
        x := x - y;
}
@ensures
        x_out=y_in ∧ y_out=x_in
```



Interactive Proof Assistants

```
Theorem app_rev:
  forall (x : list) (y : list), rev append x y = append (rev y) (rev x).
Proof.
```

….does this theorem hold? What is the proof?

Interactive Proof Assistants

Theorem app_rev:
forall (x : list) (y : list), rev append x y = append (rev y) (rev x).
Proof.
….does this theorem hold? What is the proof?


```
List := cons( head : Int, tail : List ) | nil
```

Signature

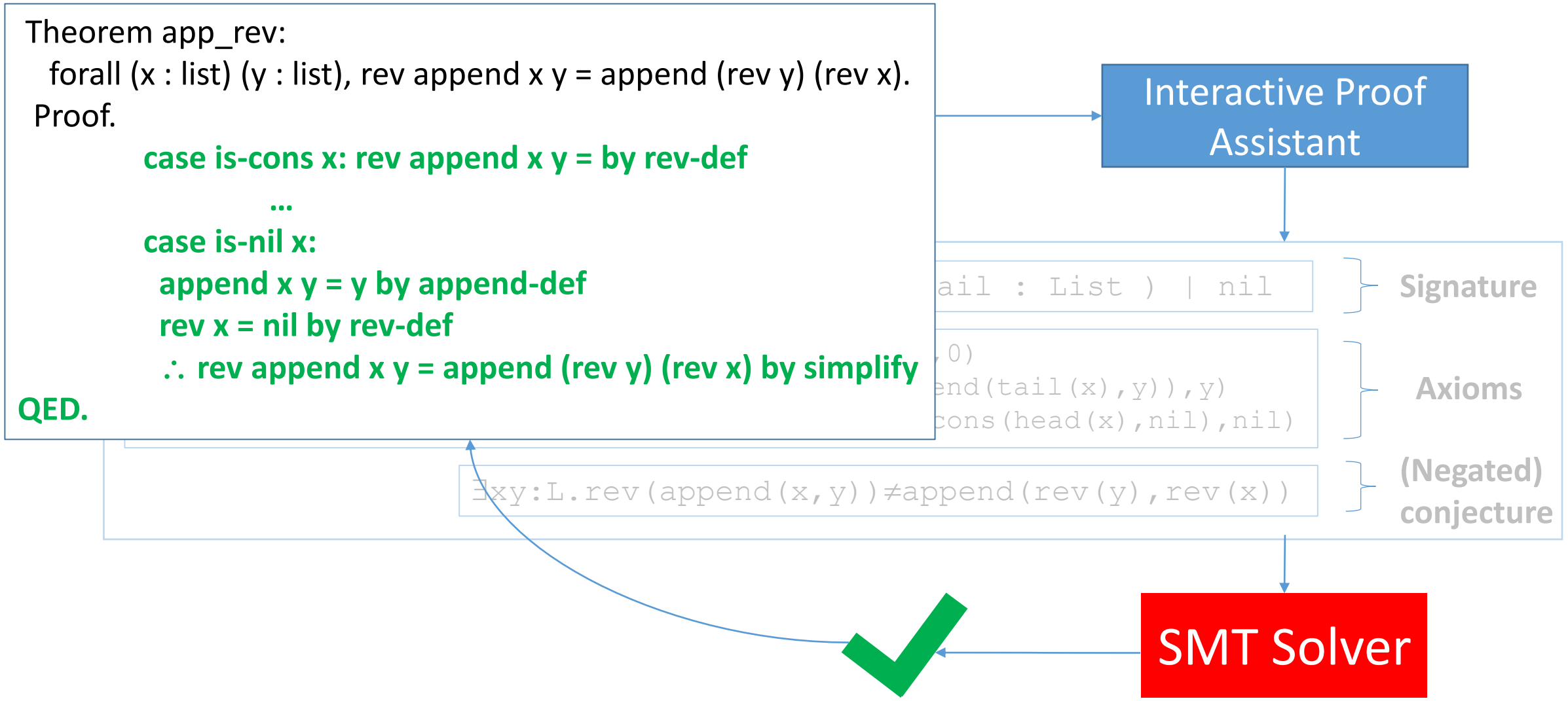
```
∀x:L.length(x)=ite(is-cons(x),1+length(tail(x)),0)
∀xy:L.append(x)=ite(is-cons(x),cons(head(x),append(tail(x),y)),y)
∀x:L.rev(x)=ite(is-cons(x),append(rev(tail(x)),cons(head(x),nil),nil)
```

Axioms

```
∃xy:L.rev(append(x,y))≠append(rev(y),rev(x))
```

(Negated) conjecture

Interactive Proof Assistants

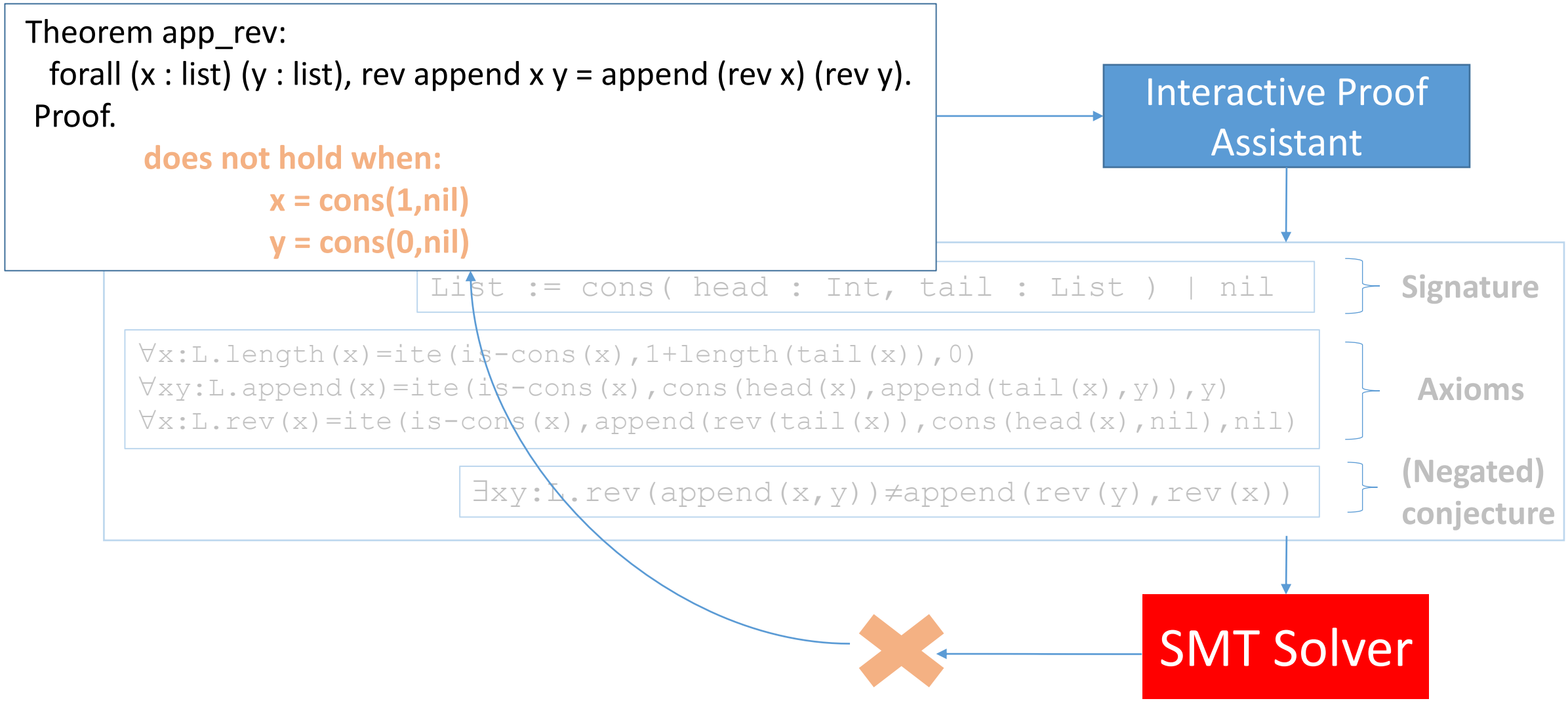


Interactive Proof Assistants

```
Theorem app_rev:
  forall (x : list) (y : list), rev append x y = append (rev x) (rev y).
Proof.
```

….does this theorem hold? What is the proof?

Interactive Proof Assistants



Symbolic execution

```
char buff[15];
char pass;
cout << "Enter the password :";
gets(buff);
if (regex_match(buff, std::regex("([A-Z]+)") )) {
    if(strcmp(buff, “PASSWORD")) {
        cout << "Wrong Password";
    } else {
        cout << "Correct Password";
        pass = 'Y';
    }
    if(pass == 'Y') {
        grant_root_permission();
        Assert(strcmp(buff,”PASSWORD”)==0);
    }
}
```

Does this assertion hold for all executions?

Symbolic Execution Engine

Symbolic execution

```
char buff[15];
char pass;
cout << "Enter the password :";
gets(buff);
if (regex_match(buff, std::regex("([A-Z]+)") )) {
    if(strcmp(buff, “PASSWORD")) {
        cout << "Wrong Password";
    } else {
        cout << "Correct Password";
        pass = 'Y';
    }
    if(pass == 'Y') {
        grant_root_permission();
        Assert(strcmp(buff,”PASSWORD”)==0);
    }
}
```

Does this assertion hold for all executions?

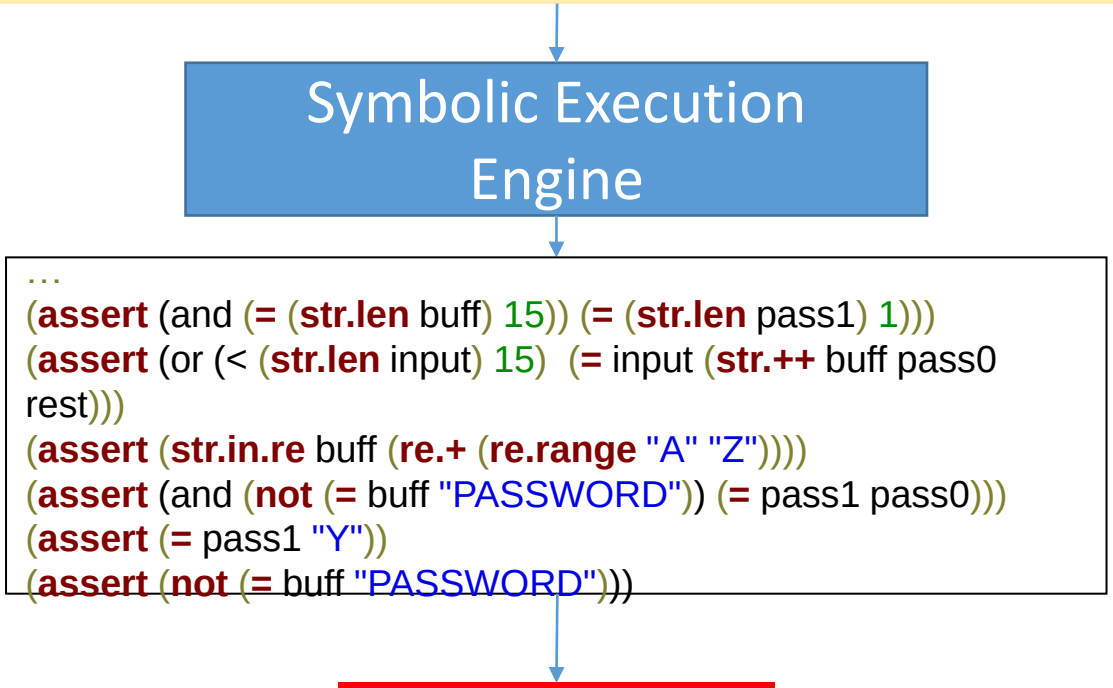
Symbolic Execution Engine

```
…
(assert (and (= (str.len buff) 15)) (= (str.len pass1) 1)))
(assert (or (< (str.len input) 15)  (= input (str.++ buff pass0
rest)))
(assert (str.in.re buff (re.+ (re.range "A" "Z"))))
(assert (and (not (= buff "PASSWORD")) (= pass1 pass0)))
(assert (= pass1 "Y"))
(assert (not (= buff "PASSWORD")))
```

Symbolic execution

```
char buff[15];
char pass;
cout << "Enter the password :";
gets(buff);
if (regex_match(buff, std::regex("([A-Z]+)") )) {
    if(strcmp(buff, “PASSWORD")) {
        cout << "Wrong Password";
    } else {
        cout << "Correct Password";
        pass = 'Y';
    }
    if(pass == 'Y') {
        grant_root_permission();
        Assert(strcmp(buff,”PASSWORD”)==0);
    }
}
```

Does this assertion hold for all executions?




```
(define-fun input () String “AAAAAAAAAAAAAAY”)
(define-fun buff () String “AAAAAAAAAAAAAAA”)
(define-fun pass () String “Y”)
```

Symbolic execution

```
char buff[15];
char pass;
cout << "Enter the password :";
gets(buff);  ← "AAAAAAAAAAAAAAY"
if (regex_match(buff, std::regex("([A-Z]+)")) ) {
    if(strcmp(buff, "PASSWORD")) {
        cout << "Wrong Password";
    } else {
        cout << "Correct Password";
        pass = 'Y';
    }
    if(pass == 'Y') {
        grant_root_permission();
        Assert(strcmp(buff,"PASSWORD")==0);
    }
}
```

(define-fun input () String "AAAAAAAAAAAAAAY")
(define-fun buff () String "AAAAAAAAAAAAAAA")
(define-fun pass () String "Y")

Symbolic Execution Engine

```
…
(assert (and (= (str.len buff) 15)) (= (str.len pass1) 1)))
(assert (or (< (str.len input) 15)  (= input (str.++ buff pass0
rest)))
(assert (str.in.re buff (re.+ (re.range "A" "Z"))))
(assert (and (not (= buff "PASSWORD")) (= pass1 pass0)))
(assert (= pass1 "Y"))
(assert (not (= buff "PASSWORD")))
```

SMT Solver

Synthesis Tools

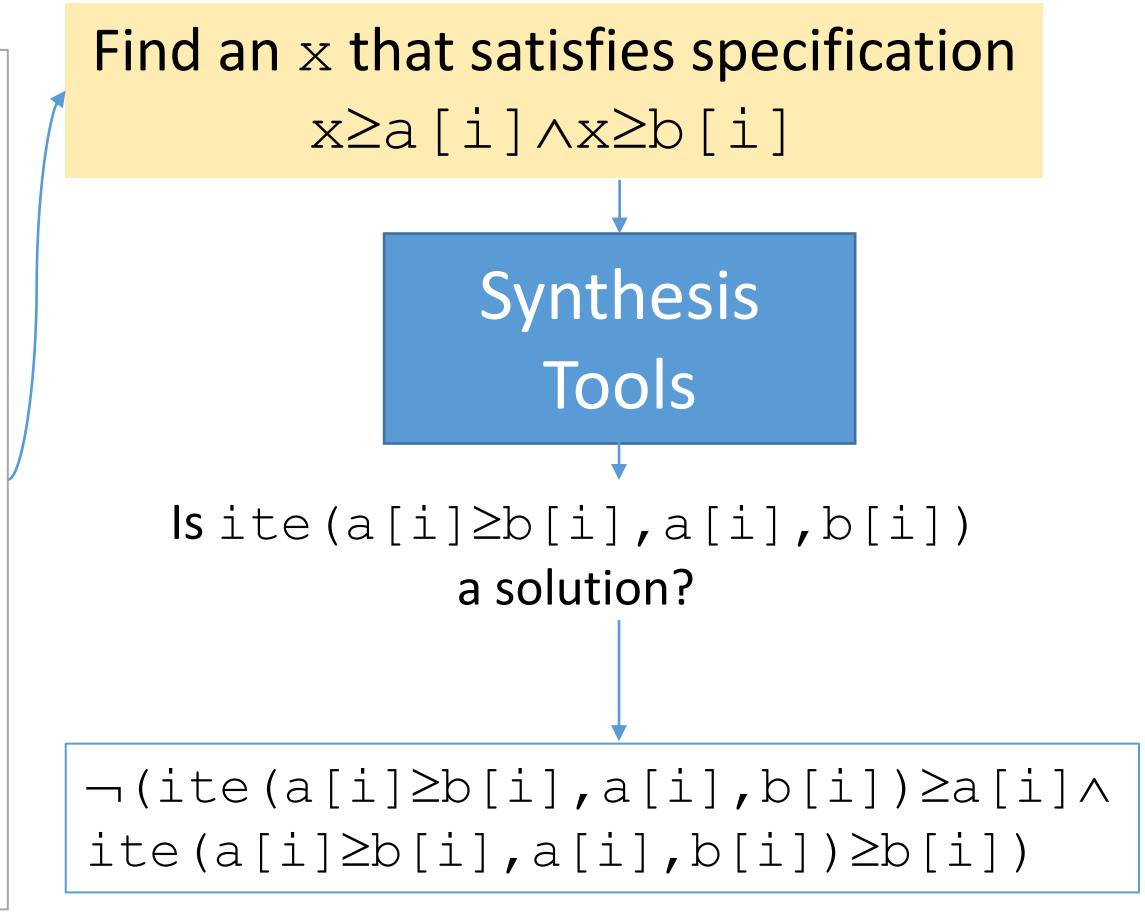
```
void maxList(List a, List b, List& c)
{
  int max;
  for(i=0;i<a.size();i++){
    max = choose(x => x≥a[i]∧x≥b[i]);
    c := c.append(max);
  }
  return c;
}
@ensures: c_out≥a ∧ c_out≥b ?
```

Find an x that satisfies specification

$x \geq a[i] \wedge x \geq b[i]$

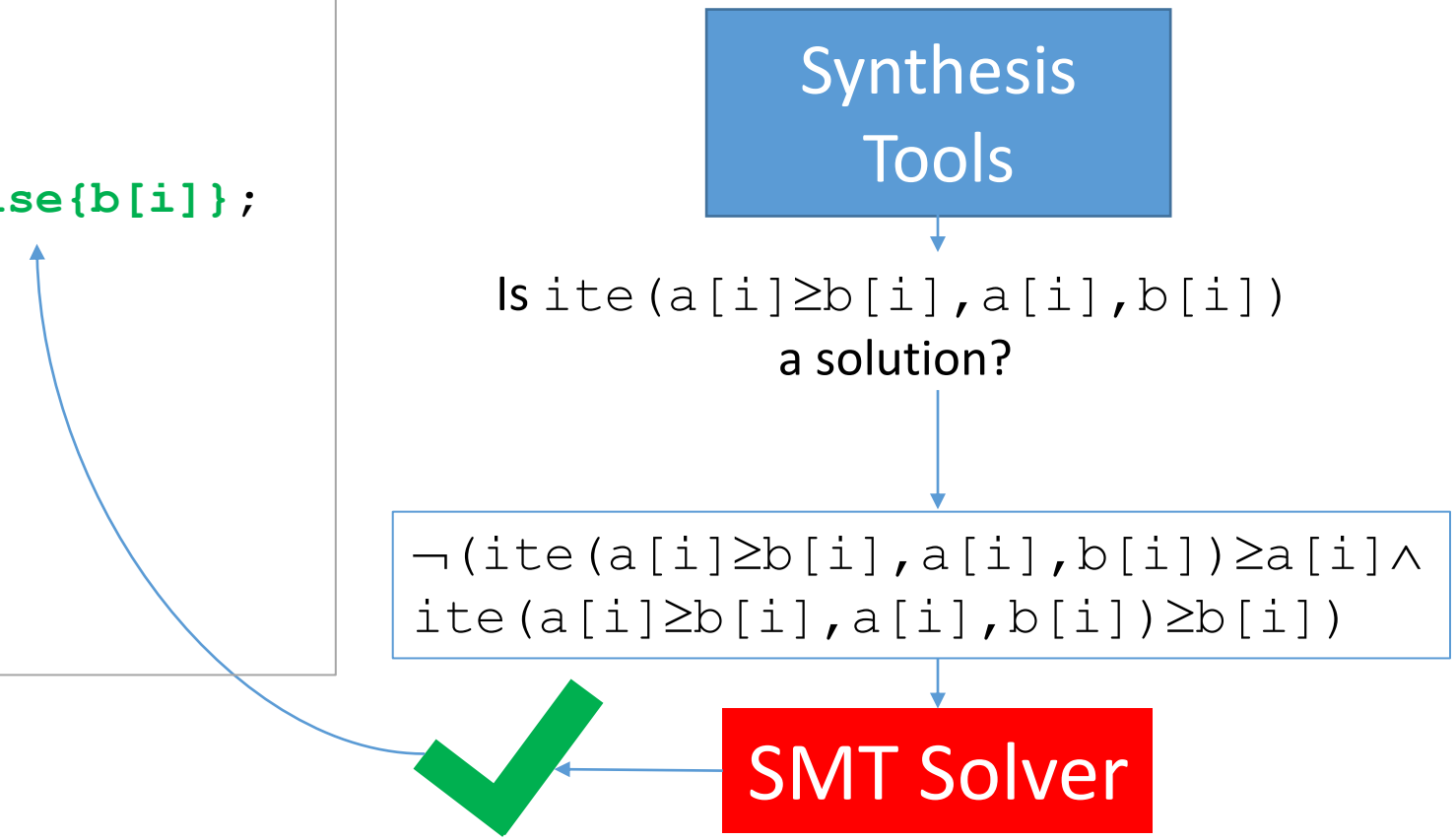
Synthesis Tools

```
void maxList(List a, List b, List& c)
{
  int max;
  for(i=0;i<a.size();i++){
    max = choose(x => x≥a[i]∧x≥b[i]);
    c := c.append(max);
  }
  return c;
}
@ensures: c_out≥a ∧ c_out≥b ?
```

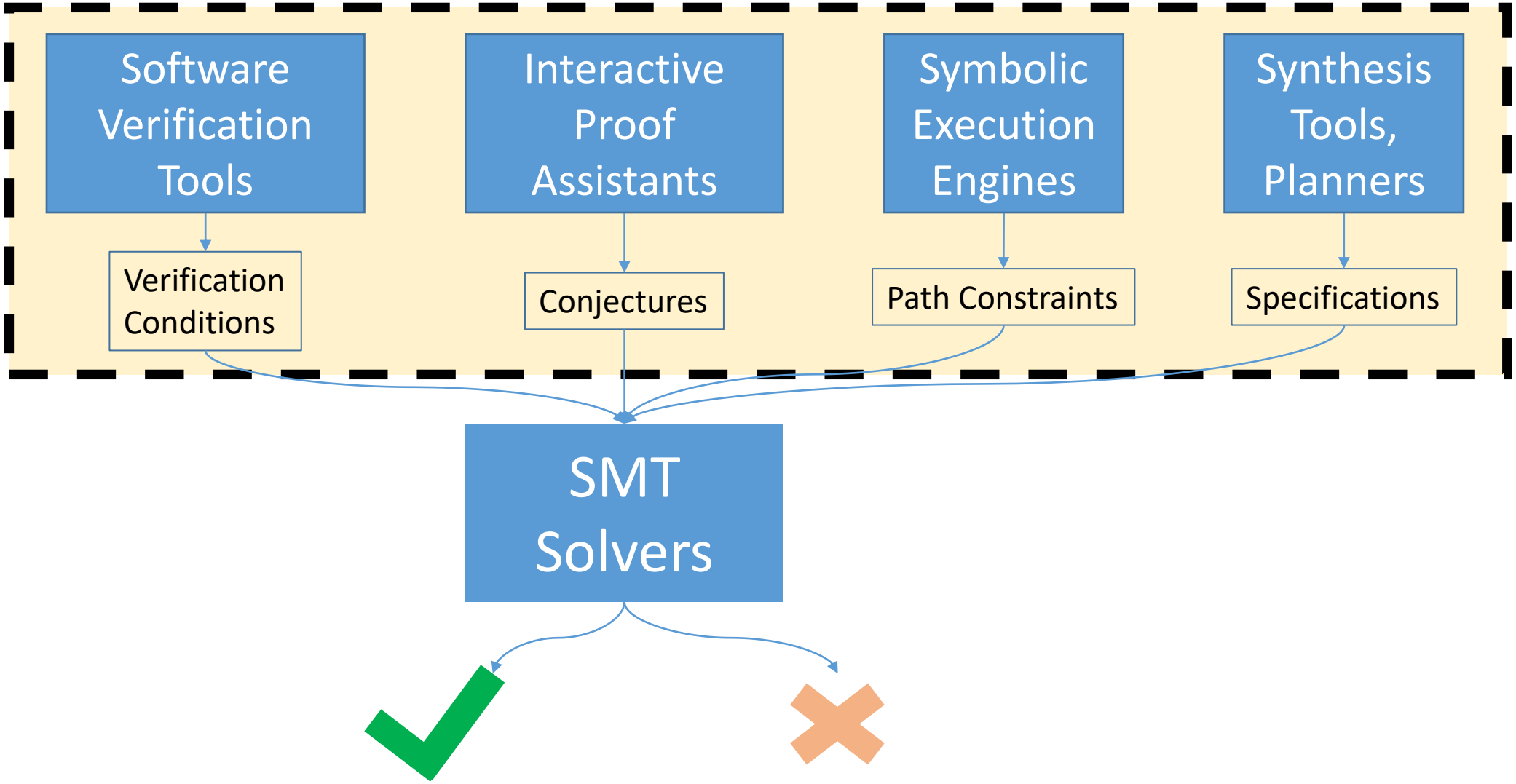


Synthesis Tools

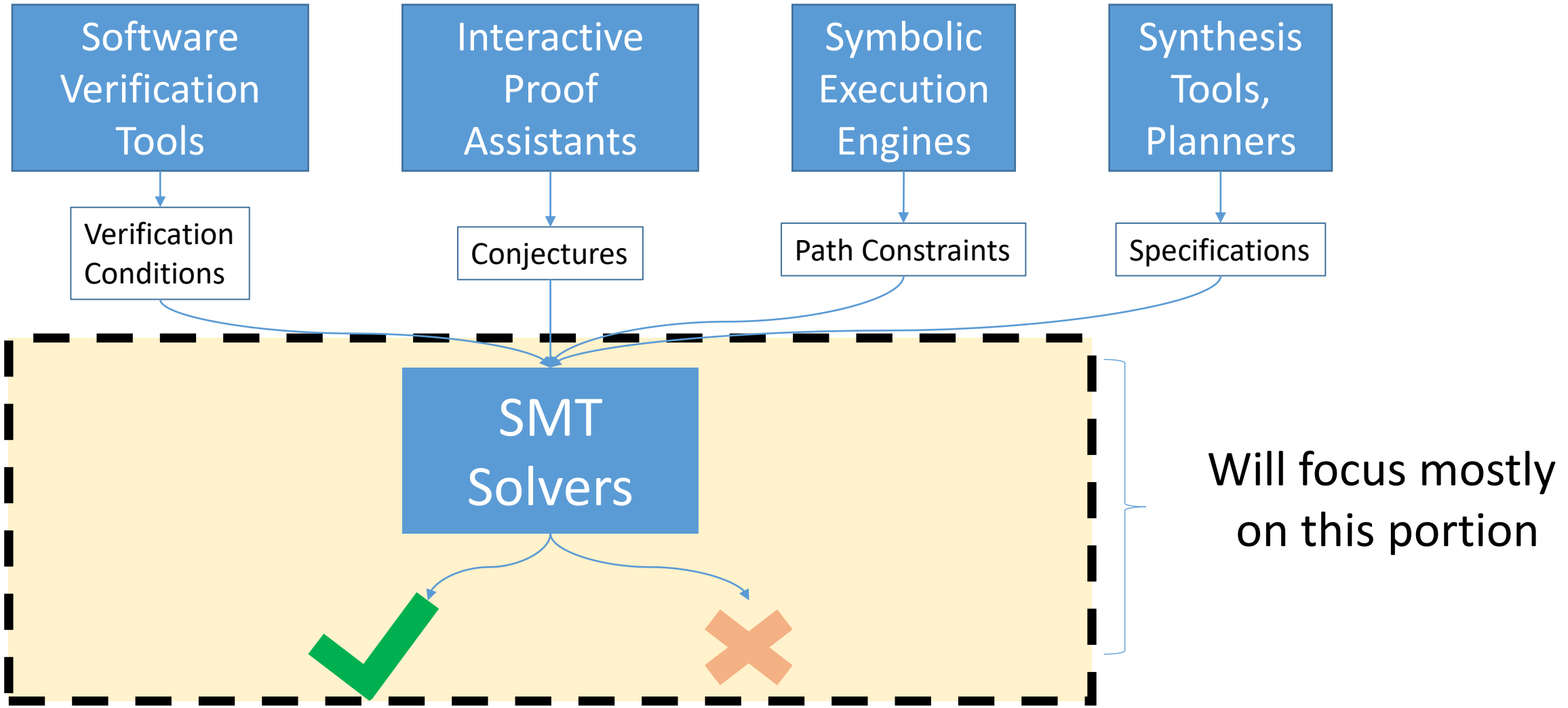
```
void maxList(List a, List b, List& c)
{
  int max;
  for(i=0;i<a.size();i++){
    max = if(a[i]≥b[i]{a[i]}else{b[i]};
    c := c.append(max);
  }
  return c;
}
@ensures: c_out≥a ∧ c_out≥b
```



Satisfiability Modulo Theories (SMT) Solvers



Satisfiability Modulo Theories (SMT) Solvers

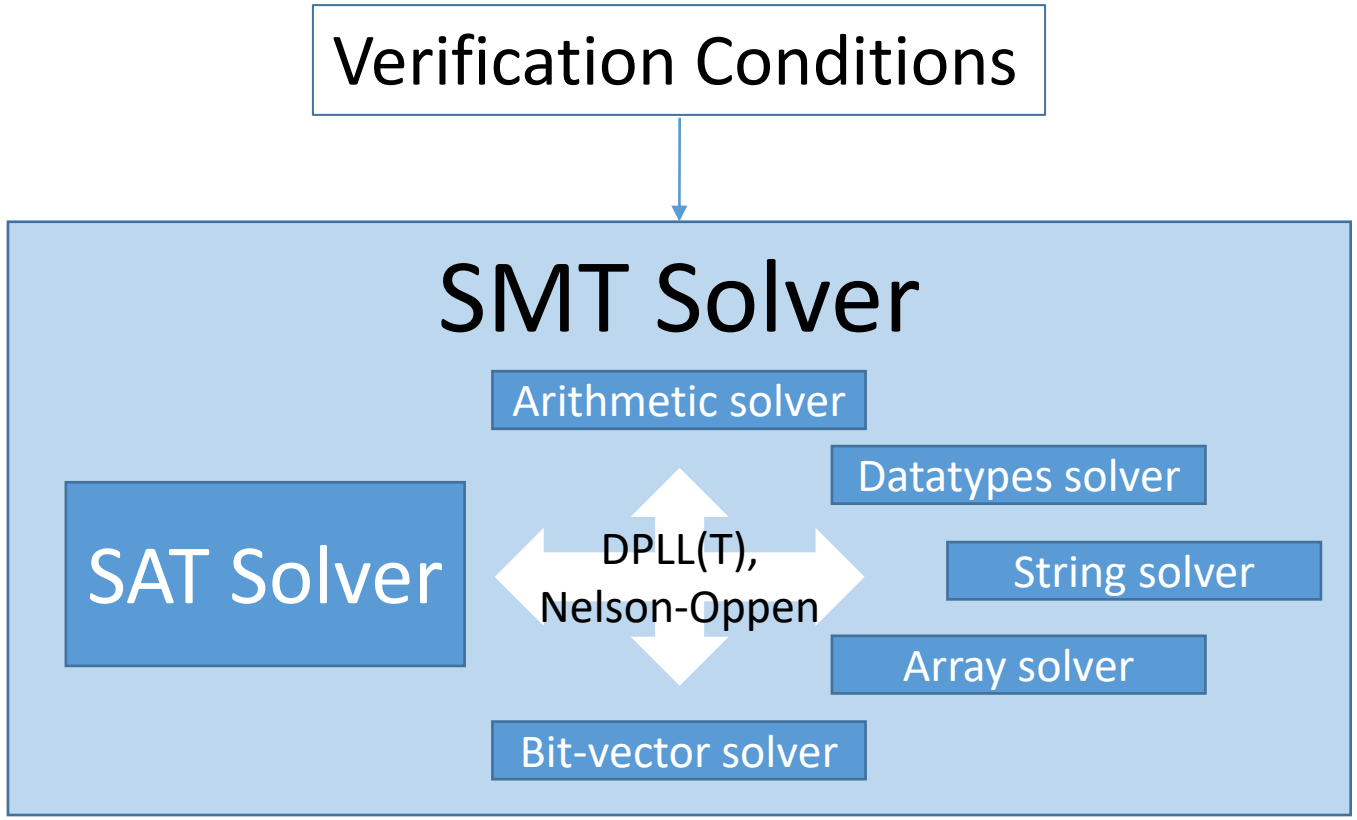


Overview

- Satisfiability Modulo Theories (SMT) solvers: how they work
  - DPLL, DPLL(T), decision procedures, Nelson-Oppen combination
- How to use SMT solvers
  - `smt2` language, models, proofs, unsat cores, incremental mode
- Things that SMT solvers can (and cannot) do well

SMT solvers

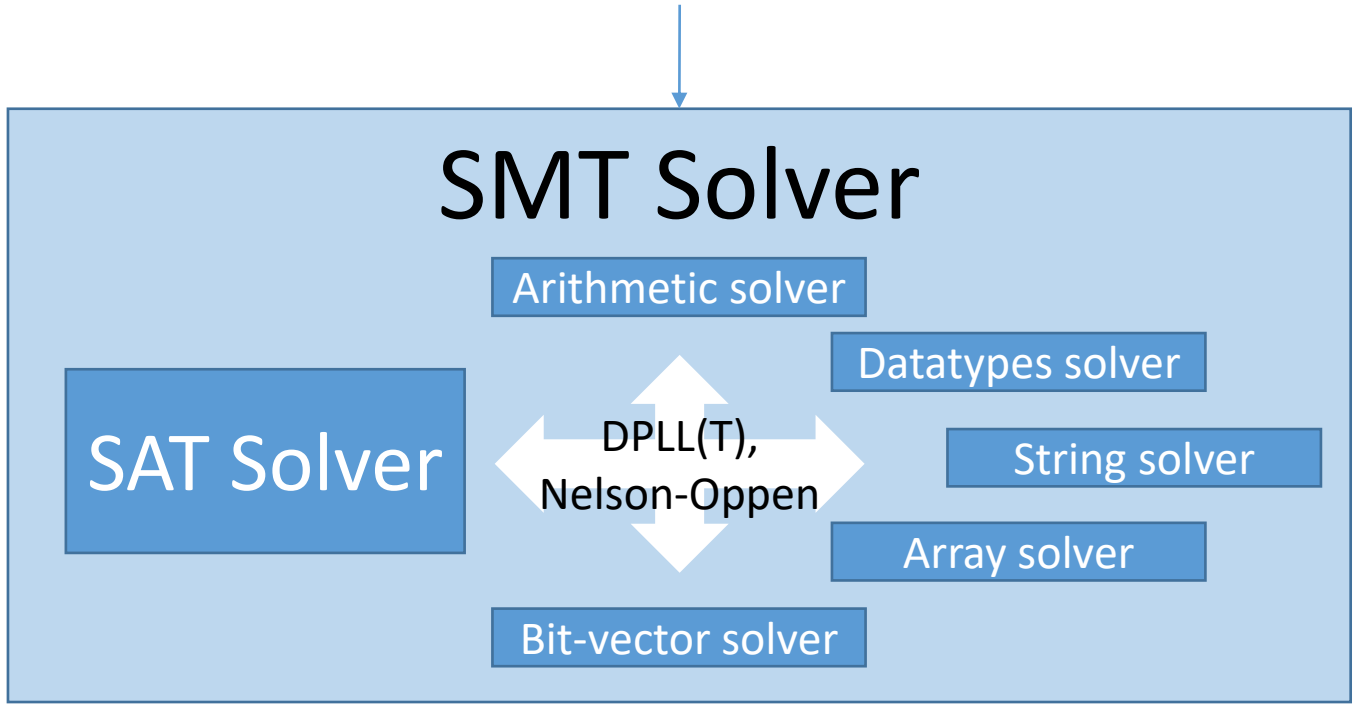
- Efficient tools for satisfiability *modulo theories*



SMT solvers

- Efficient tools for satisfiability and satisfiability *modulo theories*

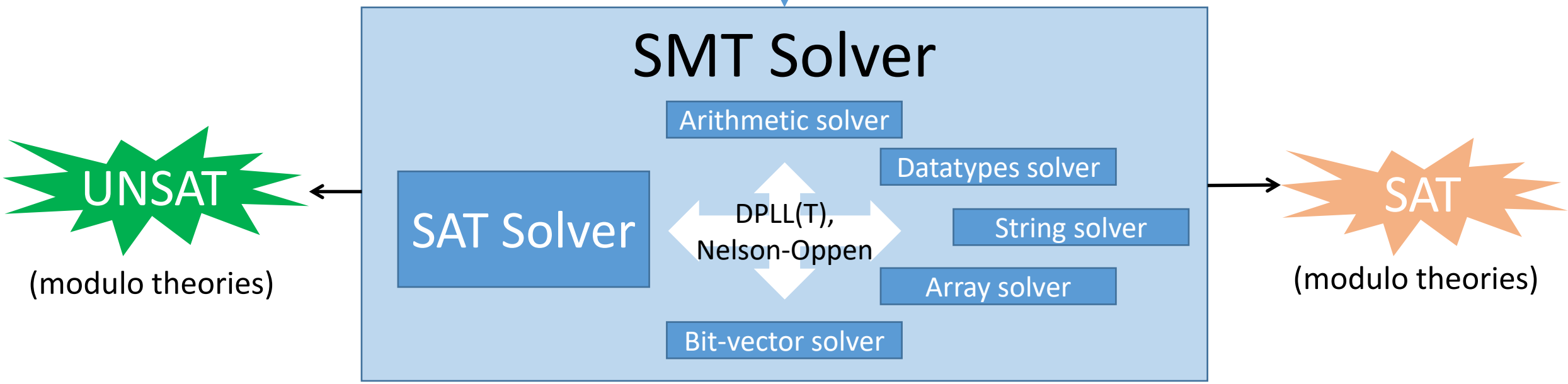
$$(A[x]+B[x]>0 \vee x+y>0) \wedge (cons(\text{“abc”},d_1)\neq d_2 \vee x<0)$$



SMT solvers

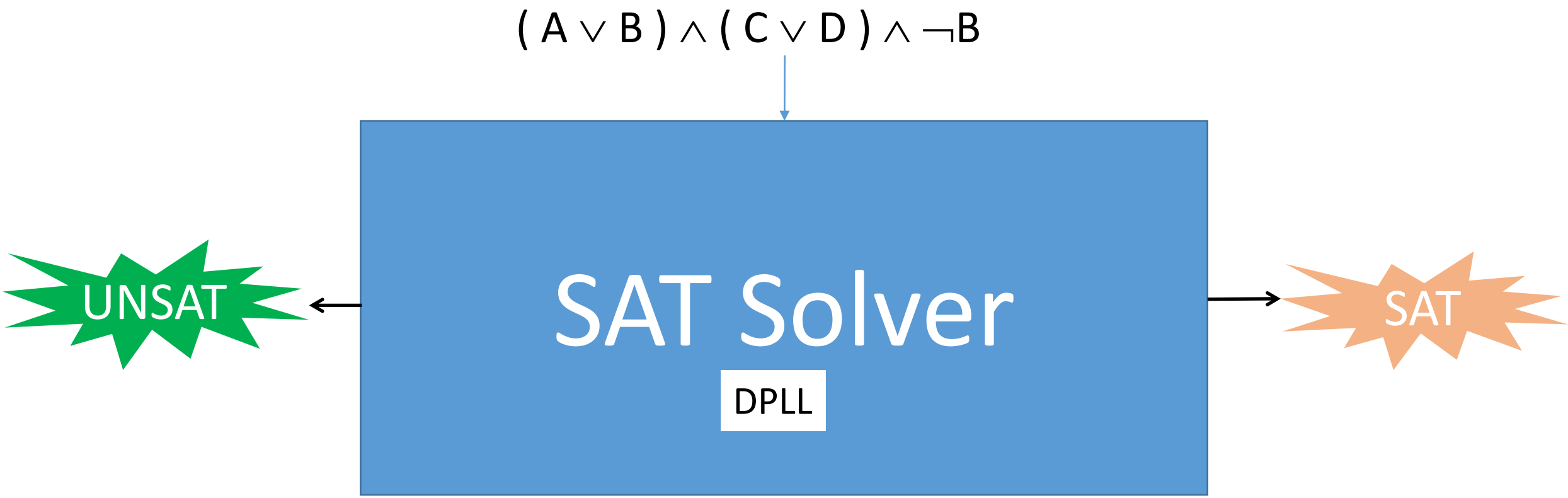
- Efficient tools for satisfiability and satisfiability *modulo theories*

$$(A[x]+B[x]>0 \vee x+y>0) \wedge (cons(\text{“abc”},d_1)\neq d_2 \vee x<0)$$



…but first : SAT solvers

- Efficient tools for *satisfiability*



[Davis–Putnam–Logemann–Loveland 1962]

DPLL

$$(\neg A \Rightarrow B) \wedge (C \vee D) \wedge \neg B$$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

Convert to clausal normal form (CNF)

- A formula is CNF if it is a conjunction of clauses
- A *clause* is a disjunction of literals e.g. $(A \vee B)$
- A *literal* is an atom or its negation e.g. A, $\neg B$,…

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- Alternate between:
  - Propagations : assign values to atoms whose value is forced
  - Decisions : choose an arbitrary value for an unassigned atom

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- DPLL algorithm
  - Propagate : $B \rightarrow$ false

Context

$B \rightarrow \bot$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- DPLL algorithm
  - Propagate : B $\rightarrow$ false
  - Propagate : A $\rightarrow$ true

Context

$B \rightarrow \bot$

$A \rightarrow T$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- DPLL algorithm
  - Propagate : B $\rightarrow$ false
  - Propagate : A $\rightarrow$ true
  - Decide : C $\rightarrow$ true

Context

$B \rightarrow \bot$

$A \rightarrow T$

$C \rightarrow T^d$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- DPLL algorithm
  - Propagate : $B \rightarrow$ false
  - Propagate : $A \rightarrow$ true
  - Decide : $C \rightarrow$ true

$\Rightarrow$ Input is 

by interpretation where
$\{A \rightarrow T, B \rightarrow \bot, C \rightarrow T\}$

Context

$B \rightarrow \bot$
$A \rightarrow T$
$C \rightarrow T^d$

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

Context

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : $A \rightarrow$ true

Context

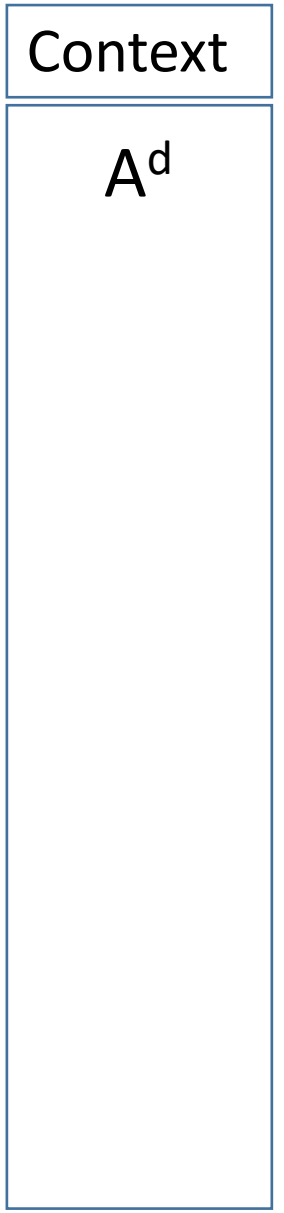
$A \rightarrow T^{d}$

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

Alternatively, can view context as set of literals

- DPLL algorithm
  - Decide : A $\rightarrow$ true



DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true

Context

A^d

B

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true
  - Propagate : C $\rightarrow$ false

Context
A^d
B
$\neg C$

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

$\Rightarrow$ Conflicting clause!
(all literals are false)

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true
  - Propagate : C $\rightarrow$ false

Context
A^d
B
$\neg C$

DPLL

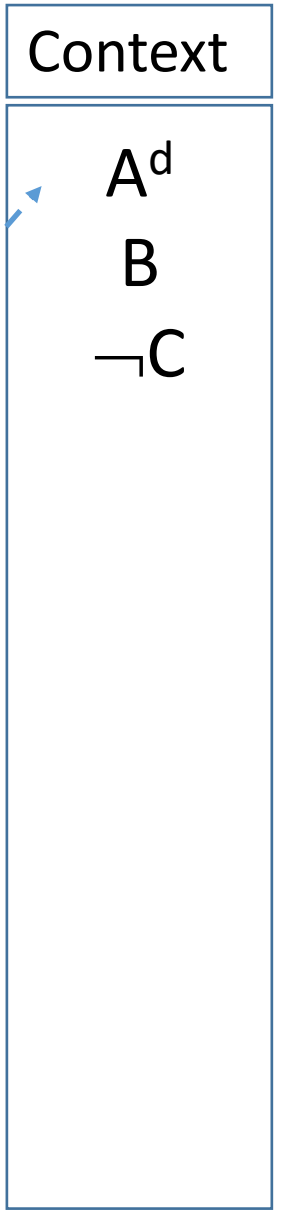
$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true
  - Propagate : C $\rightarrow$ false

$\Rightarrow$ Conflicting clause!

(all literals are false)

…*backtrack* on a decision



DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : $A \rightarrow$ false

Context

$\neg A$

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : A $\rightarrow$ false
  - Propagate : D $\rightarrow$ true

Context

$\neg A$
D

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : A $\rightarrow$ false
  - Propagate : D $\rightarrow$ true
  - Decide : B $\rightarrow$ false

Context

$\neg A$

D

B^d

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : $A \rightarrow$ false
  - Propagate : $D \rightarrow$ true
  - Decide : $B \rightarrow$ false

$\Rightarrow$ Input is 

by interpretation where
$\{A \rightarrow \bot, B \rightarrow \bot, D \rightarrow T\}$

Context

$\neg A$
D
B^d

DPLL

- Important optimizations:
  - Two watched literals
  - Non-chronological backtracking
  - Conflict-driven clause learning (CDCL)
  - Decision heuristics
  - Preprocessing / in-processing

SAT

- Using an encoding of problems into propositional logic:
  - **Pros** : Decidable, very efficient CDCL-based SAT solvers available
  - **Cons** : Not expressive, may require exponentially large encoding

    $\Rightarrow$ Motivation for Satisfiability *Modulo Theories*

SMT solvers handle formulas like:

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

SMT solvers handle formulas like:

$$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$$

- …using DPLL(**T**) algorithm for satisfiability modulo T
  - Extends DPLL algorithm to incorporate reasoning about a theory **T**
  - Combines:
    - Off-the-shelf CDCL-based SAT solver
    - *Theory Solver for T*

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

- DPLL(LIA) algorithm

  Invoke DPLL(T) for theory T = LIA (linear integer arithmetic)

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm

Context

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false

Context
$\neg x+y>0$

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true

Context
$\neg x+y>0$
$x+1>0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true

$\Rightarrow$ ***Unlike propositional SAT case, we must check T-satisfiability of context***

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }

Context is LIA-unsatisfiable!
$\Rightarrow$ one of $x+1>0$, $x<0$ must be false

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }
    - Add *theory lemma* ($\neg x+1>0 \vee \neg x<0$)

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$

$(\neg x+1>0 \vee \neg x<0) \Rightarrow$ Conflicting clause!

…*backtrack* on a decision

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }
    - Add *theory lemma* ($\neg x+1>0 \vee \neg x<0$)

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true

Context
$\neg x+y>0$
$x+1>0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - *Propagate* : $x<0 \rightarrow$ false

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }

Context is LIA-unsatisfiable!
$\Rightarrow$ one of $\neg x+y>0$, $x+y>4$ must be false

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0) \wedge (x+y>0 \vee \neg x+y>4)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }
    - Add *theory lemma* ($x+y>0 \vee \neg x+y>4$)

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0) \wedge (x+y>0 \vee \neg x+y>4)$

$\Rightarrow$ Conflicting clause!

…no decision to backtrack

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }
    - Add *theory lemma* ($x+y>0 \vee \neg x+y>4$)

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

$$(\ x+1>0 \vee x+y>0\ ) \wedge (\ x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$$
$$(\neg x+1>0 \vee \neg x<0\ ) \wedge (\ x+y>0 \vee \neg x+y>4\ )$$

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Propagate : x<0 → false
  - Propagate : x+y>4 → true
  - Invoke theory solver for LIA on: { x+1>0, ¬x+y>0, ¬x<0, x+y>4 }
    - Add *theory lemma* (x+y>0 ∨ ¬x+y>4)

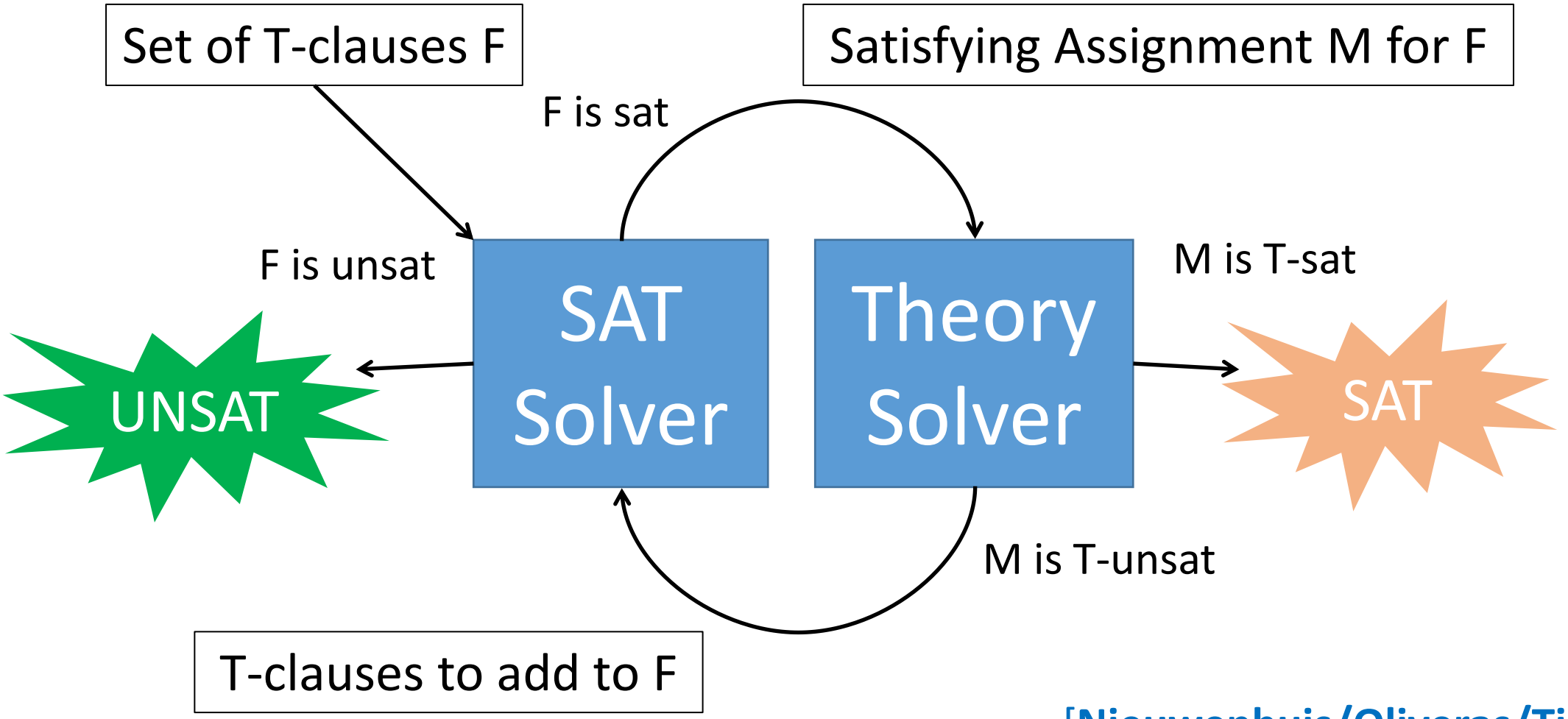
⇒ Conflicting clause!

…no decision to backtrack

⇒ *Input is* LIA-unsat

Context
¬x+y>0
x+1>0
¬x<0
x+y>4

DPLL(T)



[Nieuwenhuis/Oliveras/Tinelli 2006]

Encoding in *.smt2 format

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (or (> (+ x 1) 0) (> (+ x y) 0)))
(assert (or (< x 0) (> (+ x y) 4)))
(assert (not (> (+ x y) 0)))
(check-sat)
```

SMT-LIB format, resources: http://smtlib.cs.uiowa.edu/

Design of DPLL(T) Theory Solvers

- A DPLL(T) theory solver:
  - Should produce models when M is T-satisfiable
  - Should produce T-conflicts of minimal size when M is T-unsatisfiable
  - Should be designed to work *incrementally*
    - M is constantly being appended to/backtracked upon
  - Should cooperate with other theory solvers when combining theories

DPLL(T) Theory Solvers : Examples

- SMT solvers incorporate:
  - Theory solvers that are *decision procedures* for e.g.:
    - Theory of Equality and Uninterpreted Functions (EUF)
    - Theory of Linear Integer/Real Arithmetic
    - Theory of Arrays
    - Theory of Bit Vectors
    - Theory of Inductive Datatypes
    - …and many others
  - Theory solvers that are *incomplete procedures* for e.g.:
    - Theory of Non-Linear Integer Arithmetic
    - Theory of Strings + Length constraints

DPLL(T) Theory Solvers : Examples

- SMT solvers incorporate:
  - Theory solvers that are *decision procedures* for e.g.:
    - Theory of Equality and Uninterpreted Functions (EUF)
    - Theory of Linear Integer/Real Arithmetic
    - Theory of Arrays
    - Theory of Bit Vectors
    - **Theory of Inductive Datatypes**
    - …and many others

    } Focus of the next part
  - Theory solvers that are *incomplete procedures* for e.g.:
    - Theory of Non-Linear Integer Arithmetic
    - Theory of Strings + Length constraints

Theory of Inductive Datatypes : Example

```
ClrList := cons( head : Clr, tail : ClrList ) | nil
Clr := red | green | blue
```

Theory of Inductive Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

- Theory of Inductive Datatypes (DT)
  1. Terms with different constructors are distinct
     - red $\neq$ green
  2. Constructors are injective
     - If cons(c_1, l_1) = cons(c_2, l_2), then $c_1 = c_2$ and $l_1 = l_2$
  3. Terms of a datatype must have one of its constructors as its topmost symbol
     - Each c is such that c = red or c = green or c = blue
  4. Selectors access subfields
     - head(cons(c, l)) = c
  5. Terms do not contain themselves as subterms
     - l $\neq$ cons(c, l)

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$

- DPLL(DT) algorithm

Context

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$

Context

cons(x,nil)=cons(y,z)
y=green

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$

Context

cons(x,nil)=cons(y,z)
y=green
$x=red^d$

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Decide : x=red $\rightarrow$ true

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$

Context

$cons(x,nil)=cons(y,z)$
$y=green$
$x=red^d$

- DPLL(DT) algorithm
  - Propagate : $cons(x,nil)=cons(y,z) \rightarrow true$
  - Propagate : $y=green \rightarrow true$
  - Decide : $x=red \rightarrow true$
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$

Context

$cons(x,nil)=cons(y,z)$
$y=green$
$x=red^d$

- DPLL(DT) algorithm
  - Propagate : $cons(x,nil)=cons(y,z) \rightarrow$ true
  - Propagate : $y=green \rightarrow$ true
  - Decide : $x=red \rightarrow$ true
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}

    $\Rightarrow$DT-unsatisfiable

    Since cons(x, nil) = cons(y, nil), it must be that x = y,
    but x = red and y = green and red $\neq$ green

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$
$(\neg cons(x,nil)=cons(y,z) \vee \neg\ y=green \vee \neg\ x=red)$

- DPLL(DT) algorithm
  - Propagate : $cons(x,nil)=cons(y,z) \rightarrow true$
  - Propagate : $y=green \rightarrow true$
  - Decide : $x=red \rightarrow true$
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}
    $\Rightarrow$ …add theory lemma

Context

cons(x,nil)=cons(y,z)
y=green
$x=red^d$

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$
$(\neg cons(x,nil)=cons(y,z) \vee \neg\ y=green \vee \neg\ x=red)$

$\Rightarrow$ Conflicting clause!

…*backtrack* on a decision

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Decide : x=red $\rightarrow$ true
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}

    $\Rightarrow$ …add theory lemma

Context

cons(x,nil)=cons(y,z)
y=green
$x=red^d$

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true

Context

cons(x,nil)=cons(y,z)
y=green

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, $\neg$x=y}

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)
($\neg$cons(x,nil)=cons(y,z) $\vee$ x=y)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, $\neg$x=y}

    $\Rightarrow$ DT-unsatisfiable, add theory lemma

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)
($\neg$cons(x,nil)=cons(y,z) $\vee$ x=y) $\Rightarrow$ Conflicting clause!

…no decisions

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, $\neg$x=y}

    $\Rightarrow$ DT-unsatisfiable, add theory lemma

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)
($\neg$cons(x,nil)=cons(y,z) $\vee$ x=y) $\Rightarrow$ Conflicting clause!

…no decisions

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, $\neg$x=y}

$\Rightarrow$ Input is DT-unsat

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Encoding in *.smt2

```
(set-logic QF_DT)
(declare-datatypes ((ClrList 0) (Clr 0))(
 ((cons (head Clr) (tail ClrList)) (nil))
 ((red) (green) (blue))))
(declare-fun x () Clr)
(declare-fun y () Clr)
(declare-fun z () ClrList)
(assert (= (cons x nil) (cons y z))
(assert (or (= x red) (not (= x y))))
(assert (= y green))
(check-sat)
```

Theory Combination

- What if we have:

$$\text{IntList} := \text{cons}(\ \text{head} : \text{Int},\ \text{tail} : \text{IntList}\ ) \mid \text{nil}$$

- Example input:

$$(\ \text{head}(\ x\ ) + 3 = y \vee x = \text{cons}(\ y+1,\ \text{nil}\ )\ ) \wedge \text{head}(\ x\ ) > y+1$$

$\Rightarrow$Requires reasoning about datatypes *and* integers!

Theory Combination

- What if we have:

IntList := cons(head : Int, tail : IntList) | nil

- Example input:

$$(\text{head}(x)+3 = y \vee x = \text{cons}(y+1, \text{nil})) \wedge \text{head}(x) > y+1$$

- Idea:
  - Use DPLL(LIA+DT): find satisfying assignments $M = M_{LIA} \cup M_{DT}$
    - Use existing solver for LIA to check if M_{LIA} is LIA-satisfiable
    - Use existing solver for DT to check if M_{DT} is DT-satisfiable

Do not need to write a new theory solver for LIA+DT

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$$(\text{head}(x)+3 = y \vee x = \text{cons}(y+1, \text{nil})) \wedge \text{head}(x) > y+1$$

- DPLL(LIA+DT) algorithm

Context

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

(u_1 +3 = y $\vee$ x = cons(u_2 , nil)) $\wedge$ u_1 > y+1 $\wedge$
u_1=head(x) $\wedge$ u_2=y+1

Context

- DPLL(LIA+DT) algorithm
  $\Rightarrow$First, purify the input
  - Introduce *shared variables* u_1, u_2

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

Context

- DPLL(LIA+DT) algorithm

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$u_1+3=y^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x)\}$ … DT-satisfiable

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$u_1+3=y^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, u_1+3=y\}$

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$u_1+3=y^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, u_1+3=y\}$…LIA-unsatisfiable
    $\Rightarrow$ Add theory lemma

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$u_1+3=y^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1 \wedge (\neg u_1>y+1 \vee \neg u_1+3=y)$

$\Rightarrow$ Conflicting clause!

…*backtrack* on a decision

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, u_1+3=y\}$…LIA-unsatisfiable

    $\Rightarrow$ Add theory lemma

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$u_1+3=y^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable

  $\Rightarrow$Theory solvers must agree on shared variables u_1, u_2

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, $u_1=u_2$
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, $u_1=u_2$
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable, $u_1 \neq u_2$

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, **$u_1=u_2$**
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable, **$u_1 \neq u_2$**

  $\Rightarrow$ *Theory solvers do not agree on $u_1=u_2$!*

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, $u_1=u_2$
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable, $u_1 \neq u_2$

  $\Rightarrow$ *Theory solvers do not agree on $u_1=u_2$ … add splitting lemma for u_1, u_2*

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Decide : $u_1=u_2 \rightarrow$ true

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$u_1=u_2^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Decide : $u_1=u_2 \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), u_1=u_2\}$ … DT-satisfiable

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$u_1=u_2^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1 \wedge (\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee \neg u_2=y+1 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Decide : $u_1=u_2 \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), u_1=u_2\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y, u_1=u_2\}$ … LIA-unsatisfiable

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$\neg u_1+3=y$

$x=cons(u_2, nil)$

$u_1=u_2^{d}$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2) \wedge (\neg u_1=head(x) \vee \neg x=cons(u_2, nil) \vee u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false
  - Invoke DT solver on {$u_1=head(x)$, $x=cons(u_2, nil)$, $\neg u_1=u_2$} …DT-unsat

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2) \wedge (\neg u_1=head(x) \vee \neg x=cons(u_2, nil) \vee u_1=u_2)$

$\Rightarrow$ Conflicting clause!

…no decisions

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), \neg u_1=u_2\}$ …DT-unsat

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2) \wedge (\neg u_1=head(x) \vee \neg x=cons(u_2, nil) \vee u_1=u_2)$

$\Rightarrow$ Conflicting clause!

…no decisions

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), \neg u_1=u_2\}$ …DT-unsat

$\Rightarrow$ Input is LIA+DT-unsat

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2^d$

Encoding in *.smt2

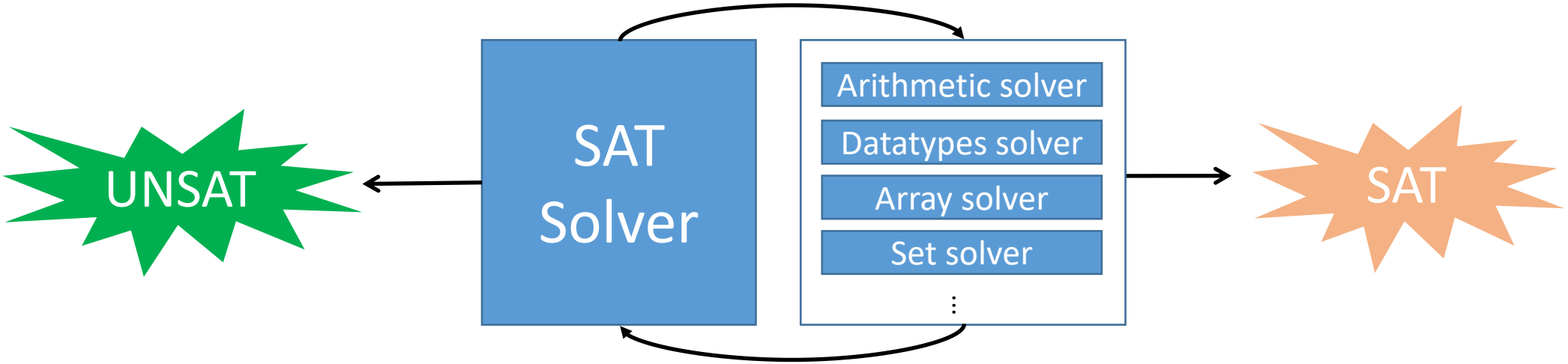
```
(set-logic QF_DTLIA)
(declare-datatypes ((IntList 0))(
 ((cons (head Int) (tail IntList)) (nil))))
(declare-fun x () IntList)
(declare-fun y () Int)
(assert (= (+ (head x) 3) y))
(assert (= x (cons (+ y 1) nil)))
(assert (> (head x) (+ y 1)))
(check-sat)
```

DPLL(T) : Theory Combination

- Nelson-Oppen Theory Combination
  - SMT solvers use preexisting theory solvers for combined theories $T_1 + \ldots + T_n$
  - Partition and distribute context M to T_1-solver, …, T_n-solver
    - If any T_i-solver says “unsat”, then M is unsatisfiable
    - If each T_i-solver says “sat”, then solvers must agree on shared variables
  - Requires theory solvers to:
    - Have disjoint signatures
      - E.g. arithmetic has functions { +, <, 0, 1, … }, datatypes has functions { cons, head, tail, … }
    - Know equalities/disequalities between shared variables
      - E.g. are $u_1 = u_2$ equal?
    - Theories agree on cardinalities for shared types
      - E.g. LIA and DT may agree that Int has infinite cardinality

DPLL(T) : Summary

- SMT solvers use
  - DPLL(T) algorithm for theory T, which uses:
    - Off-the-shelf SAT solver
    - Theory solver(s) for T
  - Nelson-Oppen theory combination for combined theories $T_1 + T_2$, which uses:
    - Existing theory solvers for T_1 and T_2, combines them using a generic method



Examples

Contract-Based Verification

```
@precondition: P1[ xin,yin ]
void f( int& x, int& y )
{
      ...
}
@ensures: P2[ xin,yin,xout,yout]
```

Property P_1 should hold for all inputs x_{in}, y_{in} to function f

Property P_2 is guaranteed to hold for x_{out}, y_{out}
(the state of x, y after calling f)

Contract-Based Verification

```
   @precondition: x_in>y_in
0  void swap(int& x, int& y)
   {
1        x := x + y;
2        y := x - y;
3        x := x - y;
   }
   @ensures: x_out=y_in ∧y_out=x_in ?
```

Contract-Based Verification

```
   @precondition: x_in>y_in
0  void swap(int& x, int& y)
   {
1      x := x + y;
2      y := x - y;
3      x := x - y;
   }
   @ensures: x_out=y_in ∧y_out=x_in
```

```
(set-logic QF_LIA)
(declare-fun xin () Int) (declare-fun yin () Int)
(assert (> xin yin))

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin) (= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 (+ x0 y0)) (= y1 y0)))

(declare-fun x2 () Int) (declare-fun y2 () Int)
(assert (and (= x2 x1) (= y2 (- x1 y1))))

(declare-fun x3 () Int) (declare-fun y3 () Int)
(assert (and (= x3 (- x2 y2)) (= y3 y2))

(declare-fun xout () Int) (declare-fun yout () Int)
(assert (and (= xout x3) (= yout y3)))

(assert (not (and (= xout yin) (= yout xin))))

(check-sat)
```

ex-swap.smt2

Contract-Based Verification

```
@precondition: x_in>y_in
0 void swap(int& x, int& y)
  {
1       x := x + y;
2       y := x - y;
3       x := x - y;
  }
@ens
```

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-swap.smt2
unsat
```

```
(set-logic QF_LIA)
(declare-fun xin () Int) (declare-fun yin () Int)
(assert (> xin yin))

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin) (= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 (+ x0 y0)) (= y1 y0)))

(declare-fun x2 () Int) (declare-fun y2 () Int)
(assert (and (= x2 x1) (= y2 (- x1 y1)))

(declare-fun x3 () Int) (declare-fun y3 () Int)
(assert (and (= x3 (- x2 y2)) (= y3 y2))

yout () Int)

xin))))

(check-sat)
```

ex-swap.smt2

Verification: Unsat Cores

```
   @precondition: x_in > y_in
0  void swap(int& x, int& y)
   {
1       x := x + y;
2       y := x - y;
3       x := x - y;
   }
   @ensures: x_out = y_in ∧ y_out = x_in
```

Is this necessary?

Verification: Unsat Cores

```
   @precondition: x_in>y_in
0  void swap(int& x, int& y)
   {
1        x := x + y;
2        y := x - y;
3        x := x - y;
   }
   @ensures: x_out=y_in ∧y_out=x_in
```

```
(set-logic QF_LIA)
(set-option :produce-unsat-cores true)
(declare-fun xin () Int) (declare-fun yin () Int)
(assert (> xin yin))

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin) (= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 (+ x0 y0)) (= y1 y0)))

(declare-fun x2 () Int) (declare-fun y2 () Int)
(assert (and (= x2 x1) (= y2 (- x1 y1))))

(declare-fun x3 () Int) (declare-fun y3 () Int)
(assert (and (= x3 (- x2 y2)) (= y3 y2))

(declare-fun xout () Int) (declare-fun yout () Int)
(assert (and (= xout x3) (= yout y3)))

(assert (not (and (= xout yin) (= yout xin))))

(check-sat)
(get-unsat-core)
```

ex-swap-unsat-core.smt2

Verification: Unsat Cores

```
@precondition: x_in>y_in
0 void swap(int& x, int& y)
  {
1       x :=
2       y :=
3       x :=
  }
  @ensures:
```

```
(set-logic QF_LIA)
(set-option :produce-unsat-cores true)
(declare-fun xin () Int) (declare-fun yin () Int)
(assert (> xin yin))

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin) (= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 (+ x0 y0)) (= y1 y0)))

(declare-fun x2 () Int) (declare-fun y2 () Int)
```

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-swap-unsat-core.smt2 --dump-unsat-cores-full
unsat
(
(= x3 (- x2 y2))
(= xout x3)
(= x2 x1)
(= yout y3)
(not (and (= xout yin) (= yout xin)))
(= y0 yin)
(= x0 xin)
(= y2 (- x1 y1))
(= x1 (+ x0 y0))
(= y3 y2)
(= y1 y0)
)
```

- **Does not contain (> xin yin)!**

ex-swap-unsat-core.smt2

Verification: Unsat Cores

```
   @precondition: x_in>y_in
0  void swap(int& x, int& y)
   {
1      x := x + y;
2      y := x - y;
3      x := x - y;
   }
   @ensures: x_out=y_in ∧y_out=x_in
```

Not necessary

$x_{in}>y_{in}$ is not in the unsatisfiable core in the *proof* of $\mathbf{x_{out}=y_{in} \wedge y_{out}=x_{in}}$

$\Rightarrow$ *precondition is not necessary to show properties of* `swap`

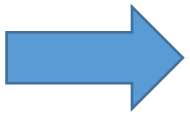
Counterexamples

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x - y;
      x := x - y;
}
@ensures: x_out=y_in ∧y_out=x_in
```

```
0  void setMax(int& x, int& y)
   {
1        if( y>x ){
           swap( x, y );
2        }
   }
   @ensures: x_out>y_out ?
```

Counterexamples

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x - y;
      x := x - y;
}
@ensures: x_out=y_in ∧y_out=x_in
```



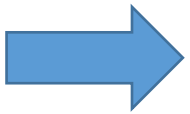
```
(set-logic QF_LIA)
(set-option :produce-models true)
(declare-fun xin () Int)(declare-fun yin () Int)

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin)(= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 x0) (= y1 y0)))

(declare-fun x2 () Int) (declare-fun y2 () Int)
; by post-condition of swap
(assert (ite (> y0 x0) (and (= x2 y1) (= y2 x1))
                       (and (= x2 x1) (= y2 y1))))

(declare-fun xout () Int) (declare-fun yout () Int)
(assert (and (= xout x2)(= yout y2)))

(assert (not (> xout yout)))

(check-sat)
(get-model)
```

```
0  void setMax(int& x, int& y)
   {
1        if( y>x ){
           swap( x, y );
2        }
   }
   @ensures: x_out>y_out
```

Counterexamples

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x - y;
      x := x - y;
}
@ensures: x_out
```

```
0  void setMax(i
   {
1        if( y>x
            swap(
2        }
   }
   @ensures: x_out
```



```
(set-logic QF_LIA)
(set-option :produce-models true)
(declare-fun xin () Int)(declare-fun yin () Int)

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin)(= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 x0) (= y1 y0)))
```

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-swap-bug.smt2
sat
(model
(define-fun xin () Int 0)
(define-fun yin () Int 0)
(define-fun x0 () Int 0)
(define-fun y0 () Int 0)
(define-fun x1 () Int 0)
(define-fun y1 () Int 0)
(define-fun x2 () Int 0)
(define-fun y2 () Int 0)
(define-fun xout () Int 0)
(define-fun yout () Int 0)
)
```

Counterexamples

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x - y;
      x := x - y;
}
@ensures: x_out=y_in ∧y_out=x_in
```

```
0  void setMax(int& x, int& y)
   {
1        if( y>x ){
           swap( x, y );
2        }
   }
   @ensures: x_out >y_out
```

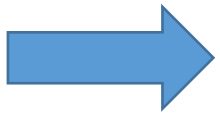
…when $x_{in}=0$ and $y_{in}=0$

Contract-Based Verification

```
   @precondition: x_in>5
0  void foo(int& x, int& y)
   {
1        if( x*y==3 ){
           x=-1;
         }
2  }
   @ensures: x_out>5 ?
```

Contract-Based Verification

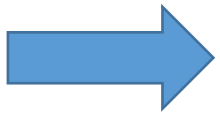
```
@precondition: x_in>5
0  void foo(int& x, int& y)
   {
1       if( x*y==3 ){
          x=-1;
        }
2  }
@ensures: x_out>5
```



```
(set-logic QF_NIA)
(declare-fun xin () Int) (declare-fun yin () Int)
(assert (> xin 5))

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin)(= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 x0) (= y1 y0)))

(declare-fun x2 () Int) (declare-fun y2 () Int)
(assert (ite (= (* y1 x1) 3)
  (and (= x2 (- 1)) (= y2 y1))
  (and (= x2 x1) (= y2 y1))))

(declare-fun xout () Int) (declare-fun yout () Int)
(assert (and (= xout x2) (= yout y2)))

(assert (not (> xout 5)))

(check-sat)
```

Contract-Based Verification

```
@precondition: x_in>5
0  void foo(int& x, int& y)
   {
1       if( x*y==3 ){
           x=-1;
        }
2  }
@ensures: x_out>5
```



```
(set-logic QF_NIA)
(declare-fun xin () Int) (declare-fun yin () Int)
(assert (> xin 5))

(declare-fun x0 () Int) (declare-fun y0 () Int)
(assert (and (= x0 xin)(= y0 yin)))

(declare-fun x1 () Int) (declare-fun y1 () Int)
(assert (and (= x1 x0) (= y1 y0)))

(declare-fun x2 () Int) (declare-fun y2 () Int)
(assert (ite (= (* y1 x1) 3)
  (and (= x2 (- 1)) (= y2 y1))
  (and (= x2 x1) (= y2 y1))))

(declare-fun xout () Int) (declare-fun yout () Int)
                                  = yout y2)))


(check-sat)
```

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-nia.smt2
unsat
```

Contract-Based Verification

```
   @precondition: x_in>5
0  void foo(int& x, int& y)
   {
1        if( x*y==3 ){
           x=-1;
         }
2  }
   @ensures: x_out>5
```

…using heuristic techniques for non-linear arithmetic

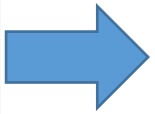
(incomplete, can work often in practice)

Contract-Based Verification

```
   @precondition: res_in==0
0  void cubes(int a, int b, int c, int& res)
   {
1         if( (a*a*a)+(b*b*b)+(c*c*c)==33 ){
            res = 1;
          }
2  }
   @ensures: res_out==0 ?
```

Contract-Based Verification

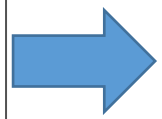
```
@precondition: res_in==0
0 void cubes(int a, int b, int c, int& res)
  {
1       if( (a*a*a)+(b*b*b)+(c*c*c)==33 ){
          res = 1;
        }
2 }
@ensures: res_out==0
```



```
(set-logic QF_NIA)
(declare-fun a () Int) (declare-fun b () Int)
(declare-fun c () Int) (declare-fun resin () Int)
(assert (= resin 0))

(declare-fun res0 () Int)
(assert (= res0 resin))

(declare-fun res1 () Int)
(assert (= res1 res0))

(declare-fun res2 () Int)
(assert (ite (= (+ (* a a a) (* b b b) (* c c c)) 33)
             (= res2 1)
             (= res2 res1)))


(declare-fun resout () Int)
(assert (= resout res2))

(assert (not (= resout 0)))

(check-sat)
```

Contract-Based Verification

```
@precondition: res_in==0
0 void cubes(int a, int b, int c, int& res)
  {
1       if( (a*a*a)+(b*b*b)+(c*c*c)==33 ){
          res = 1;
        }
2 }
@ensures: res_out==0
```



```
(set-logic QF_NIA)
(declare-fun a () Int) (declare-fun b () Int)
(declare-fun c () Int) (declare-fun resin () Int)
(assert (= resin 0))

(declare-fun res0 () Int)
(assert (= res0 resin))

(declare-fun res1 () Int)
(assert (= res1 res0))

(declare-fun res2 () Int)
(assert (ite (= (+ (* a a a) (* b b b) (* c c c)) 33)
             (= res2 1)
             (= res2 res1)))

(check-sat)
```

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-nia-hard.smt2
unknown

C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-nia-hard.smt2 --nl-ext-tplanes
^C
```

Contract-Based Verification

```
   @precondition: res_in==0
0  void cubes(int a, int b, int c, int& res)
   {
1         if( (a*a*a)+(b*b*b)+(c*c*c)==33 ){
            res = 1;
          }
2  }
   @ensures: res_out==0 ?
```

…the SMT solver will (typically) not solve open problems in mathematics!

Contract-Based Verification

```
@precondition: res_in==0
0 void cubes(int a, int b, int c, int& res)
  {
1       if( (a*a*a)+(b*b*b)+(c*c*c)==33 ){
          res = 1;
        }
2 }
@ensures: res_out==0 ?
```

…the SMT solver will (typically) not solve open problems in mathematics!

ABSTRACTIONS BLOG

Sum-of-Three-Cubes Problem Solved for 'Stubborn' Number 33

37 |

A number theorist with programming prowess has found a solution to 33 = $x^3 + y^3 + z^3$, a much-studied equation that went unsolved for 64 years.

$(8{,}866{,}128{,}975{,}287{,}528)^3 + (-8{,}778{,}405{,}442{,}862{,}239)^3 + (-2{,}736{,}111{,}468{,}807{,}040)^3 = 33$

Decision Procedures

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

Constraints Supported by SMT Solvers

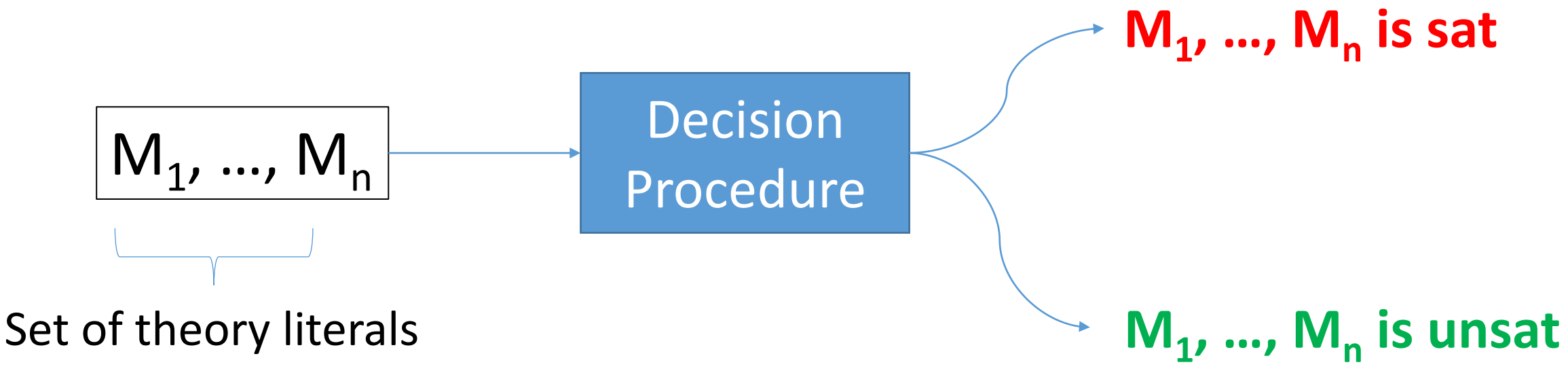
- SMT solvers support:
  - Arbitrary Boolean combinations of theory constraints
  - Examples of supported *theories*:
    - Uninterpreted functions: `f(a)=g(b,c)`
    - Linear real/integer arithmetic: `a≥b+2*c+3`
    - Arrays: `select(A,i)=select(store(A,i+1,3),i)`
    - Bit-vectors: `bvule(x,#xFF)`
    - 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]**
    - …

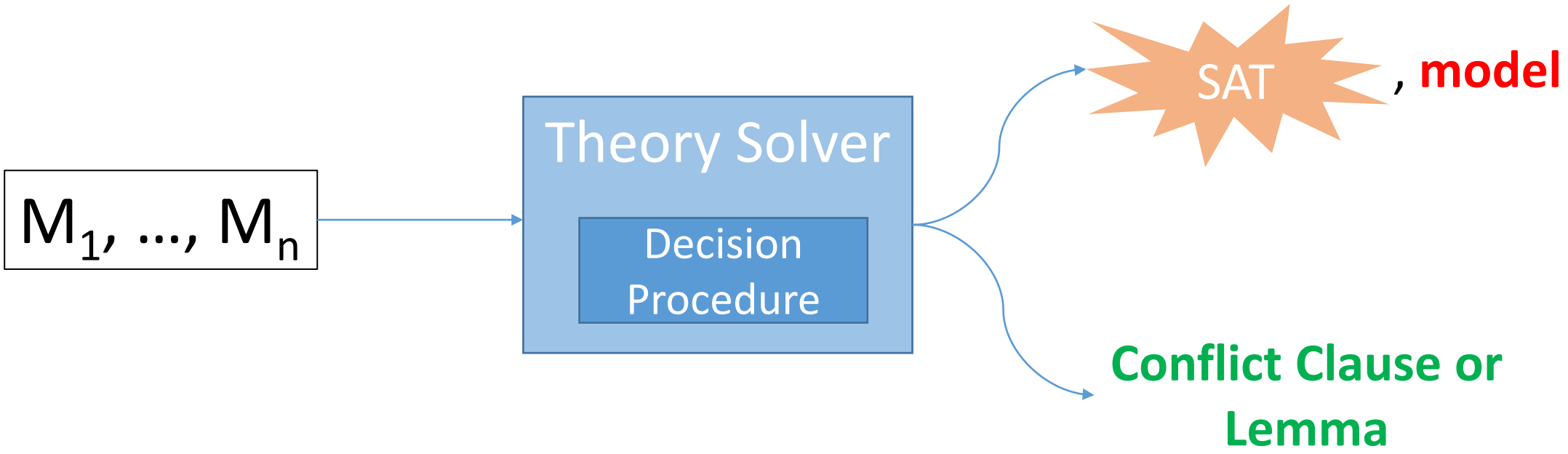
Decision Procedures in DPLL(T)

- A *decision procedure* has the following interface:



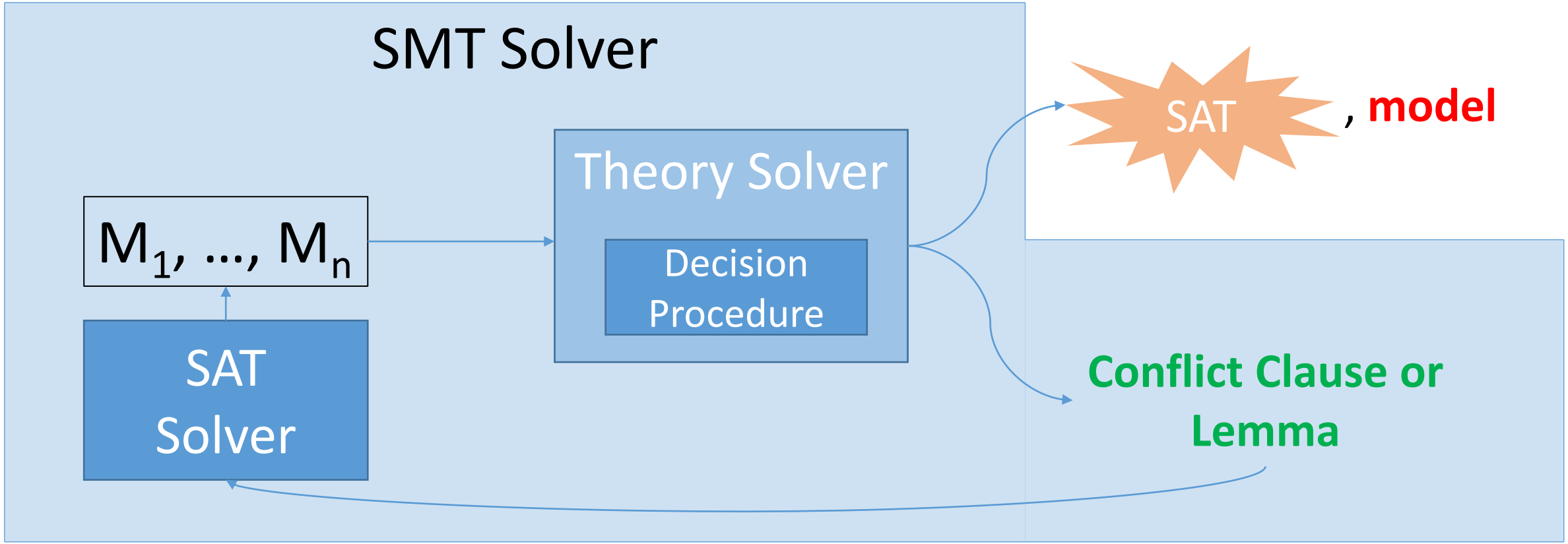
Decision Procedures in DPLL(T)

- 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

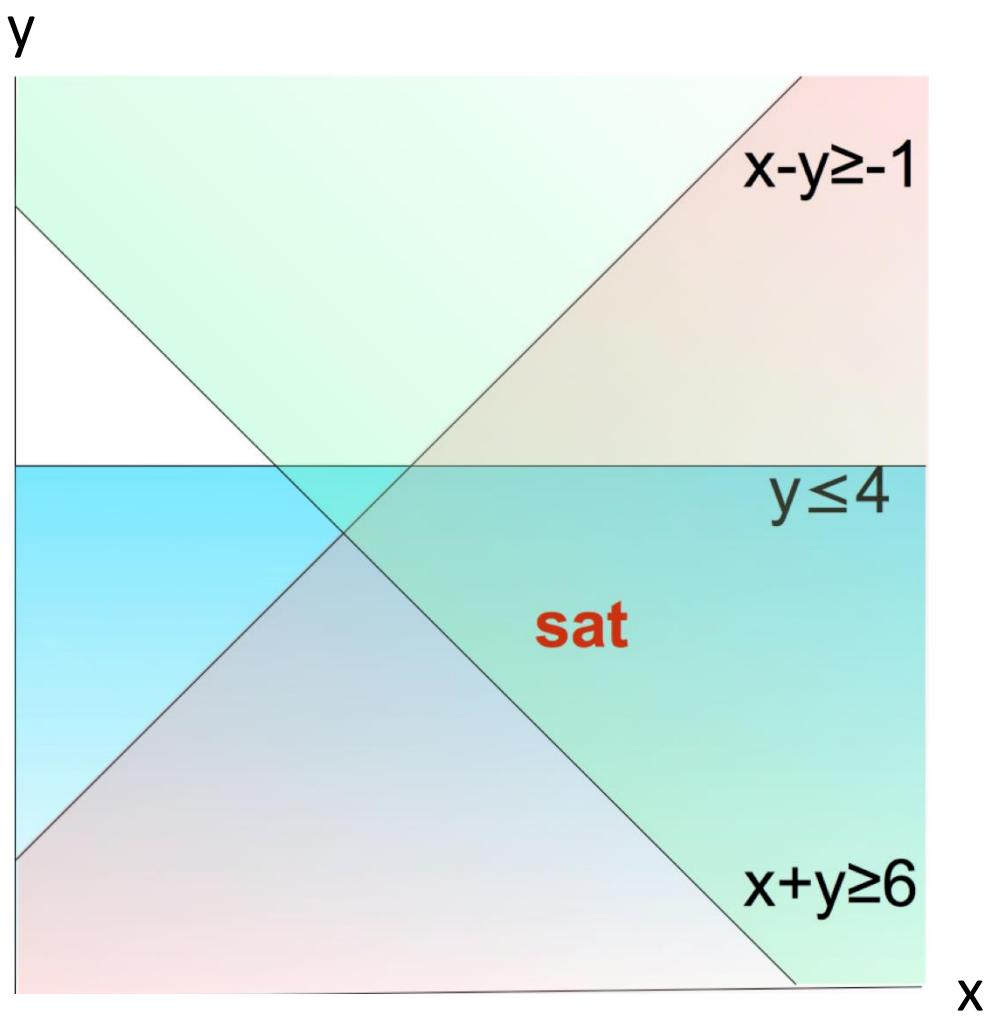
$$x : \text{Real}; \; y : \text{Real};$$
$$x - y \geq -1, \;\; y \leq 4, \;\; x + y \geq 6$$

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

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

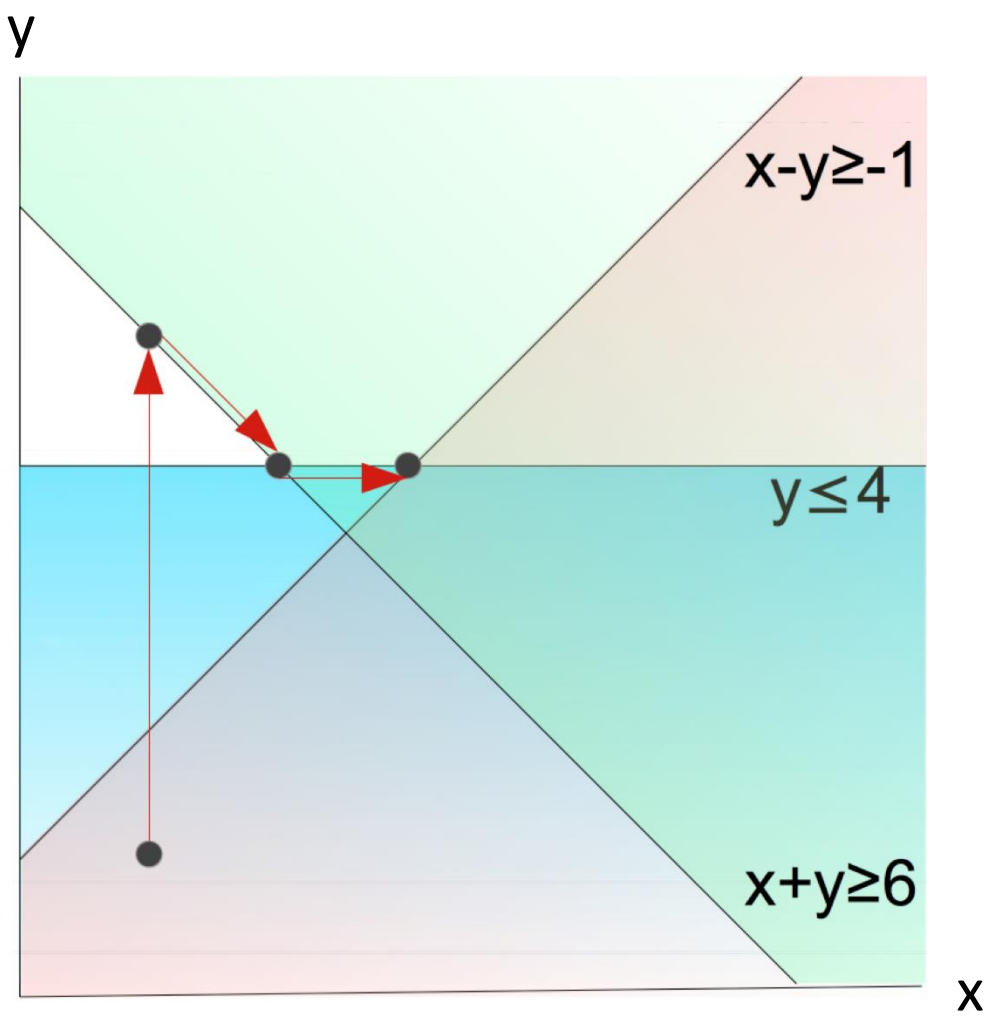
Simplex Search

$x - y \geq -1 \wedge y \leq 4 \wedge x + y \geq 6$



Is an intersection of half planes empty?

Simplex Search



$x - y \geq -1 \wedge y \leq 4 \wedge x + y \geq 6$

$x = 1, y = 1$?

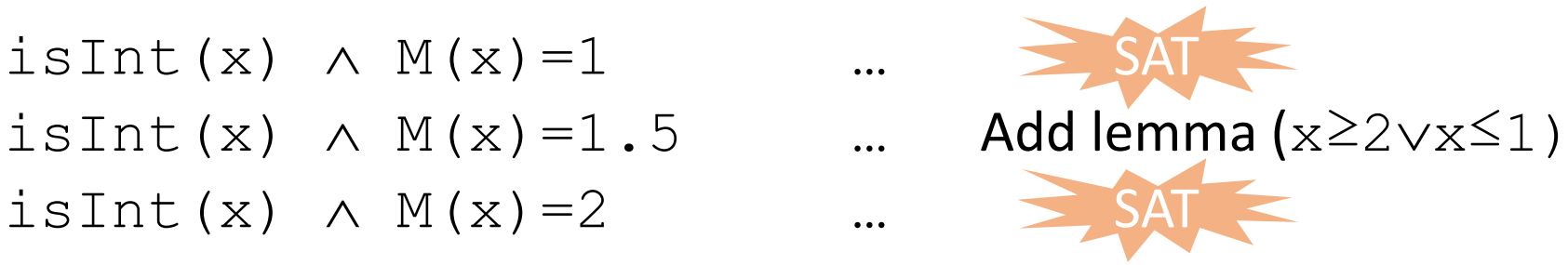
$x = 1, y = 5$?

$x = 2, y = 4$?

$x = 3, y = 4$ SAT

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 **[Griggio 2012, Jovanovic/de Moura 2013, Bromberger 2018]**



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

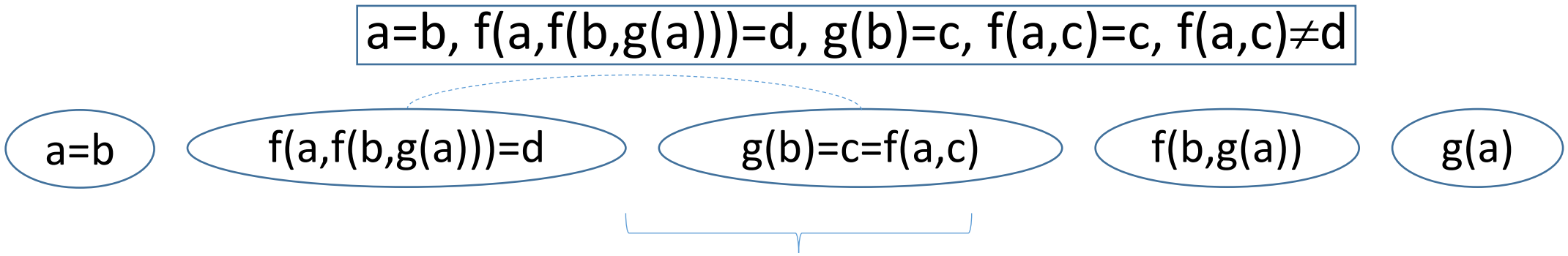
$a,b,c,d : U$

$g : U \rightarrow U$

$f : U \times U \rightarrow U$

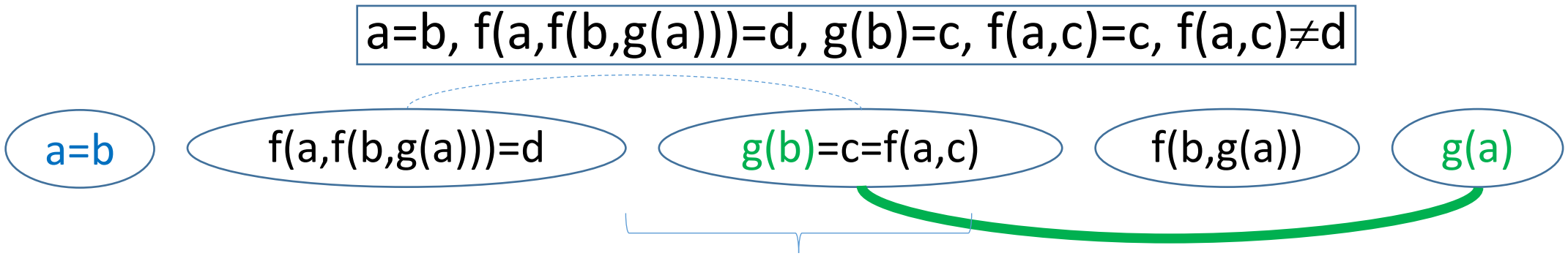
$\Rightarrow$ UF are useful for abstracting processes, other symbols not natively supported by solver

Congruence Closure



Compute *equivalence classes* of all (sub)terms

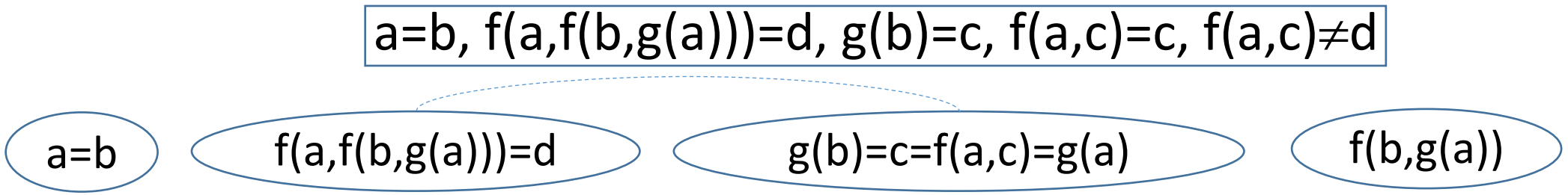
Congruence Closure



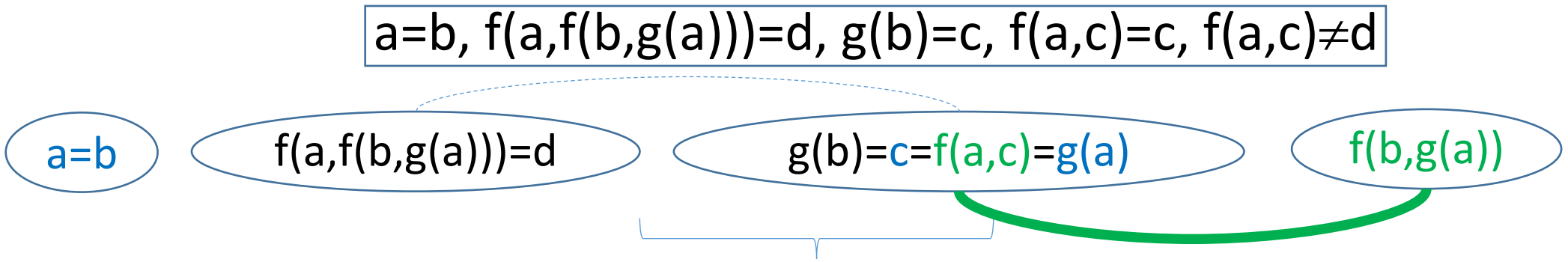
…merge congruent terms

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

Congruence Closure



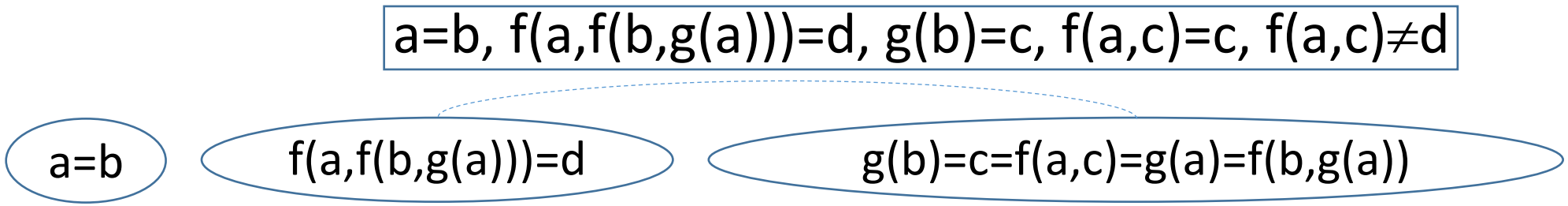
Congruence Closure



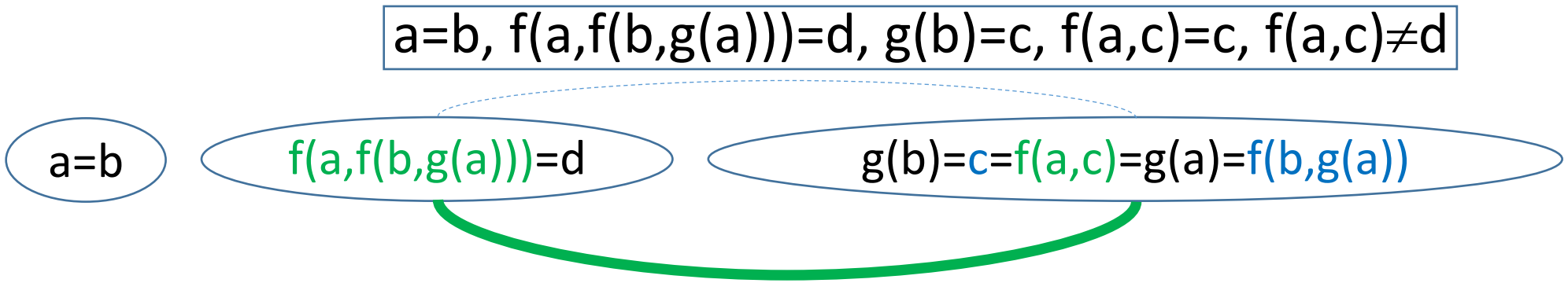
…merge congruent terms

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

Congruence Closure



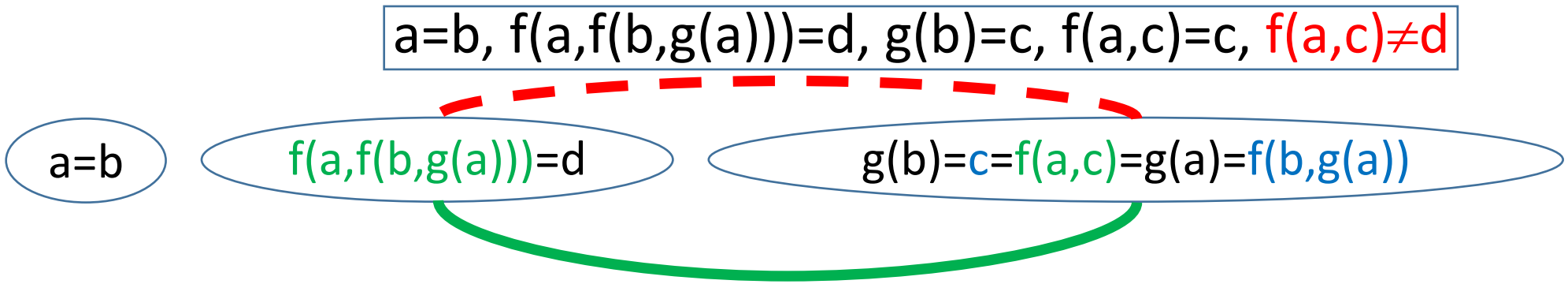
Congruence Closure



…merge congruent terms

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

Congruence Closure

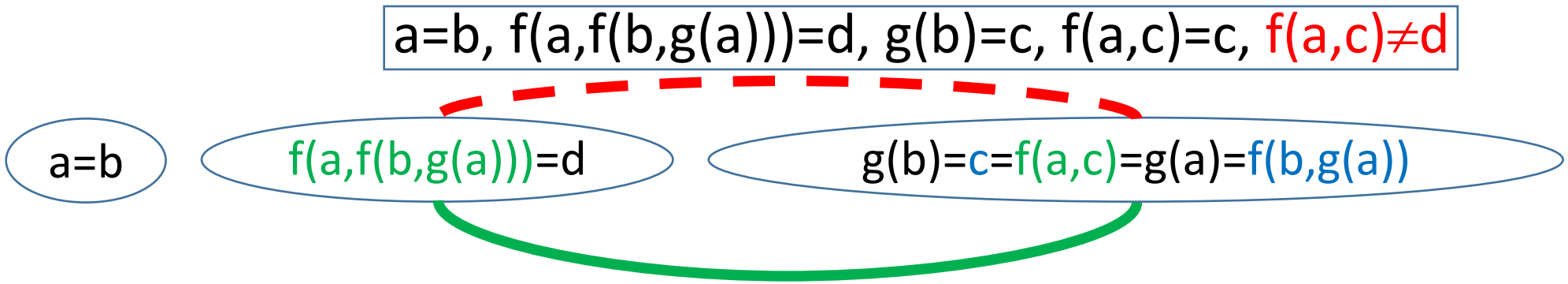


…merge congruent terms

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

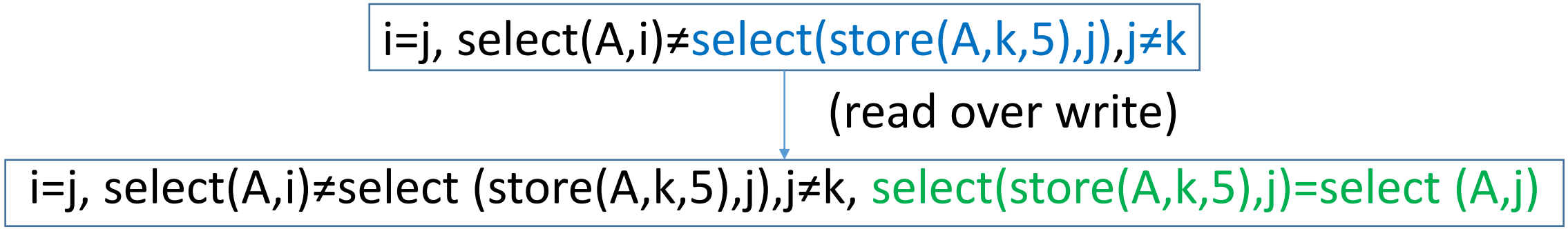
(select a i) : Element at index i of Array a

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

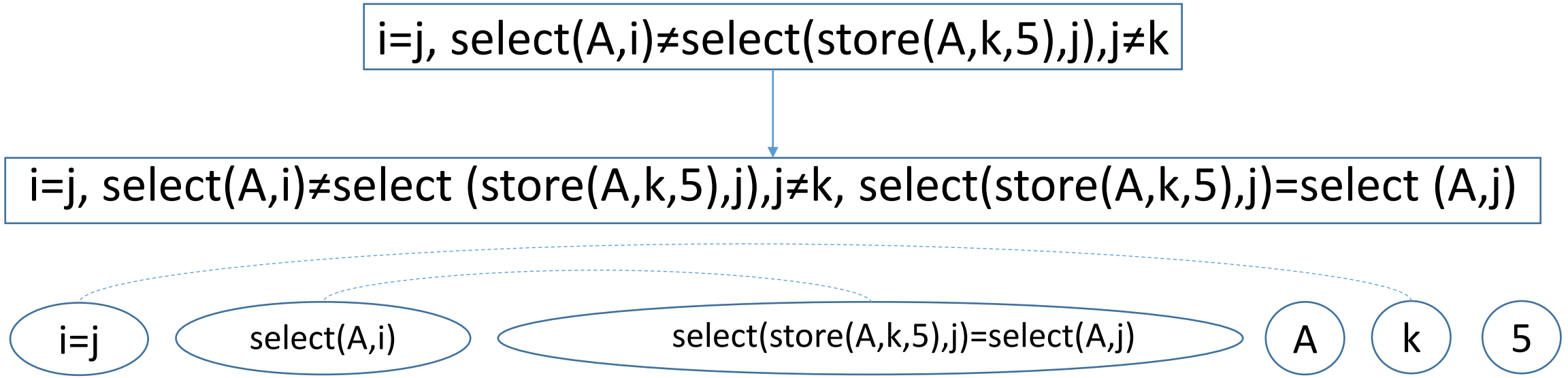
Procedure for Arrays with Extensionality

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

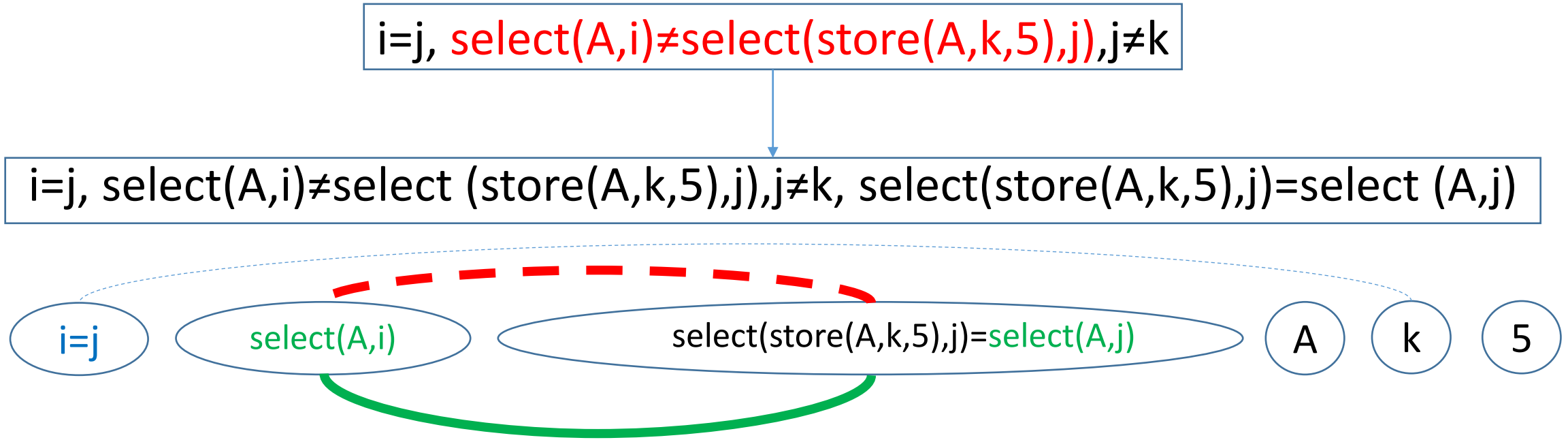
Procedure for Arrays with Extensionality



Procedure for Arrays with Extensionality



Procedure for Arrays with Extensionality



…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

$\Rightarrow$ 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

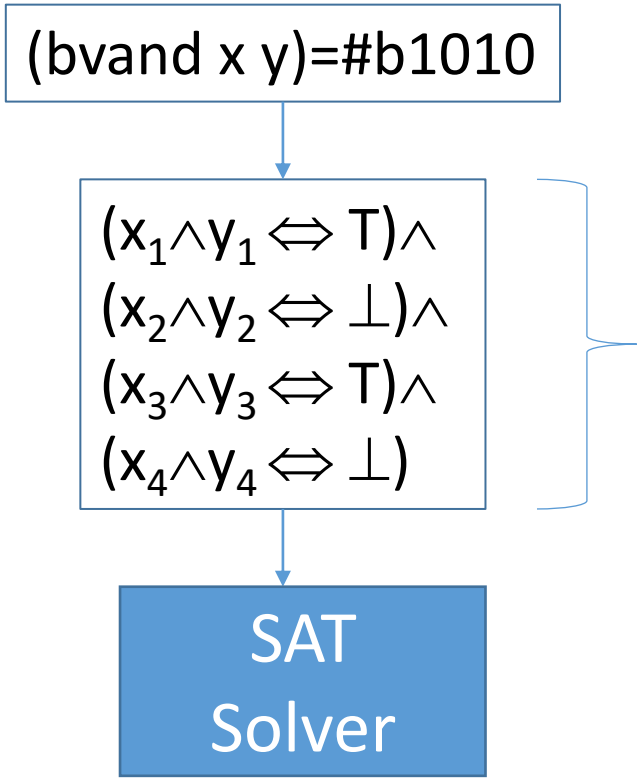
  $\Rightarrow$Eager reduction to propositional satisfiability

- For example:

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

Solving Bit Vectors

- For example:



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

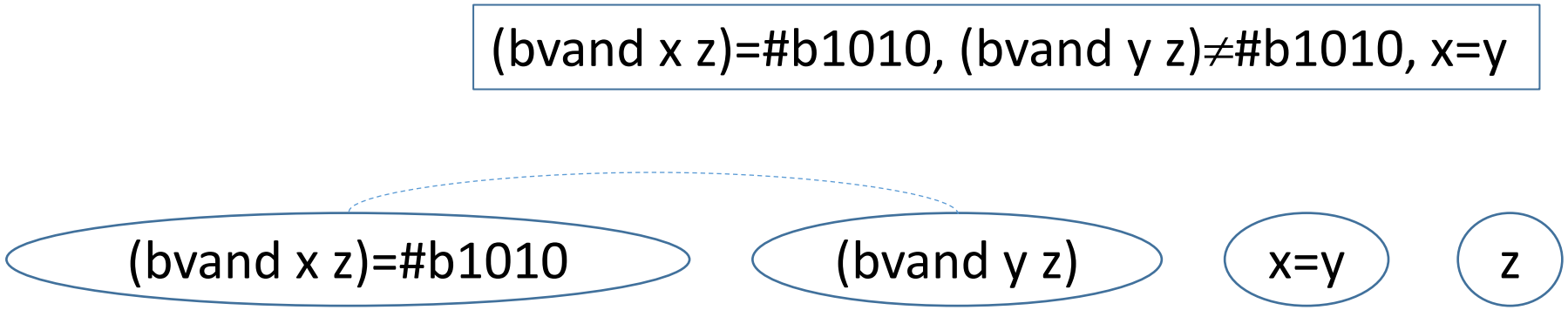
Solving Bit Vectors

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

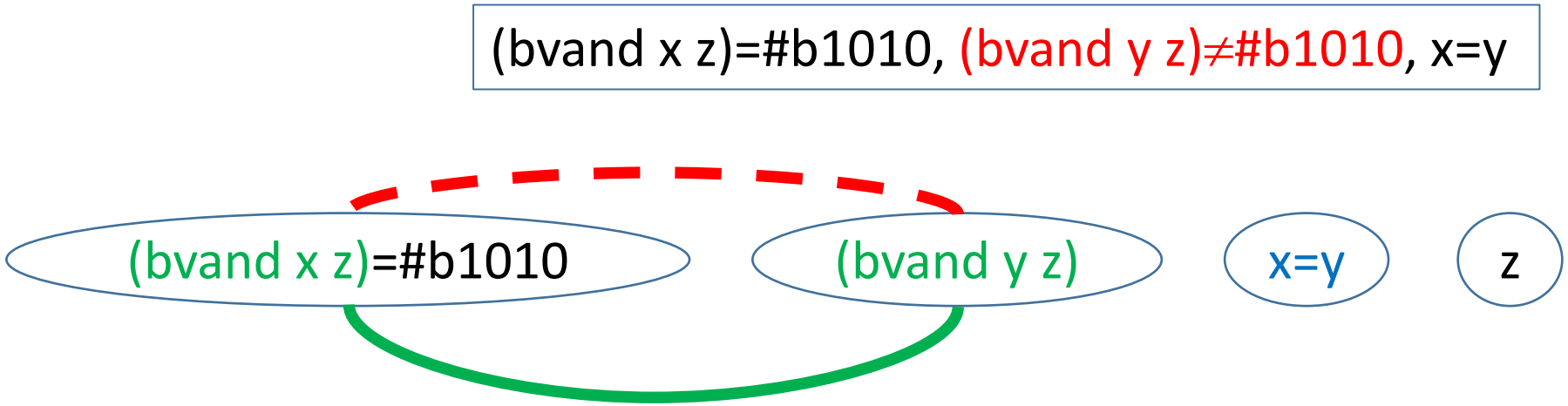
(bvand x z)=#b1010, (bvand y z)≠#b1010, x=y

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

Solving Bit Vectors



Solving Bit Vectors



…merge congruent terms

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

$\Rightarrow$ ***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

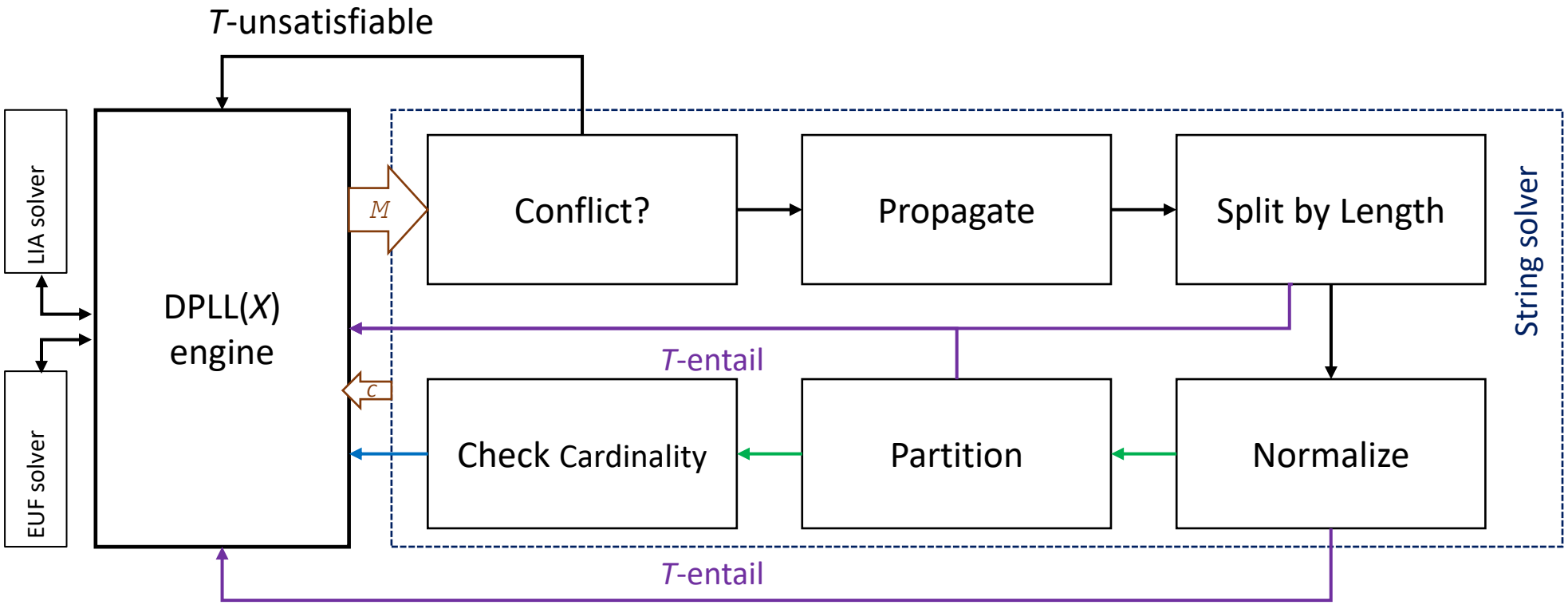
- Equalities and disequalities between *string terms* :
  - Variables: x
  - String constants: “abc”
  - Concatenation: $x \cdot \text{“abc”}$
  - Length: $|x|$
- *Linear arithmetic* constraints: $|x|+4>|y|$

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

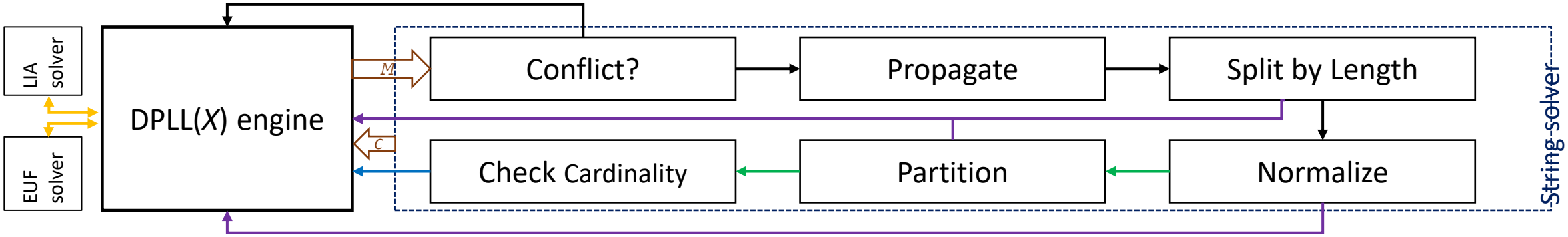
Procedures in **[Abdulla et al CAV2014, Liang et al CAV2014]**

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

General String Solver Architecture



DPLL(*T*): Find Satisfying Assignment



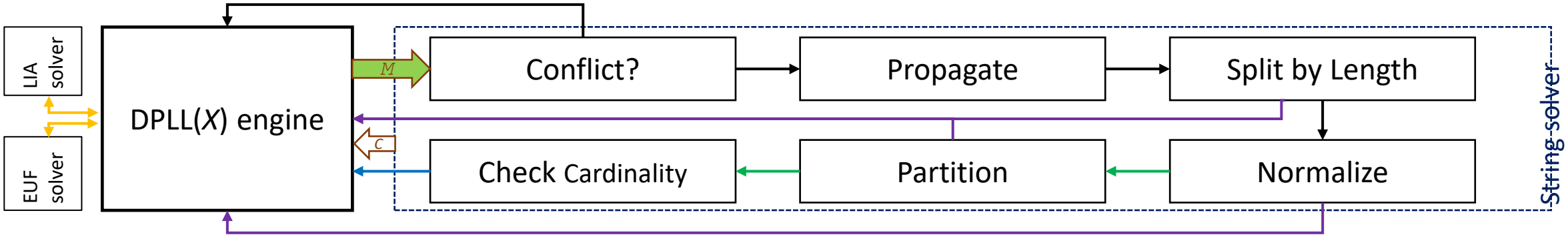
$$|x| > |y|$$
$$(x \cdot z = y \cdot w \cdot \text{“ab”} \vee x = y)$$

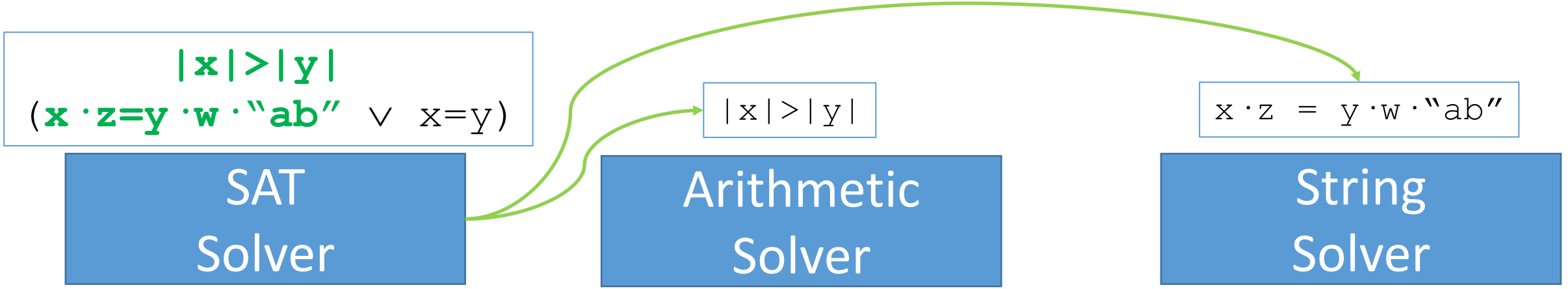
SAT Solver

Arithmetic Solver

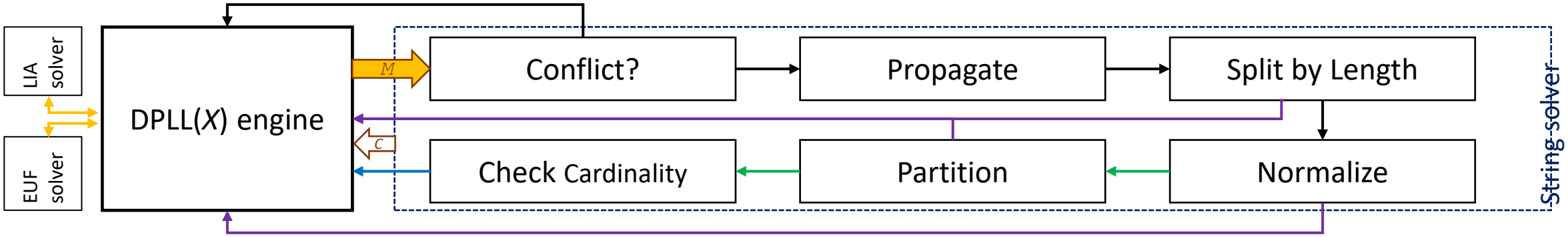
String Solver

DPLL(*T*): Find Satisfying Assignment





DPLL(*T*): Find Satisfying Assignment



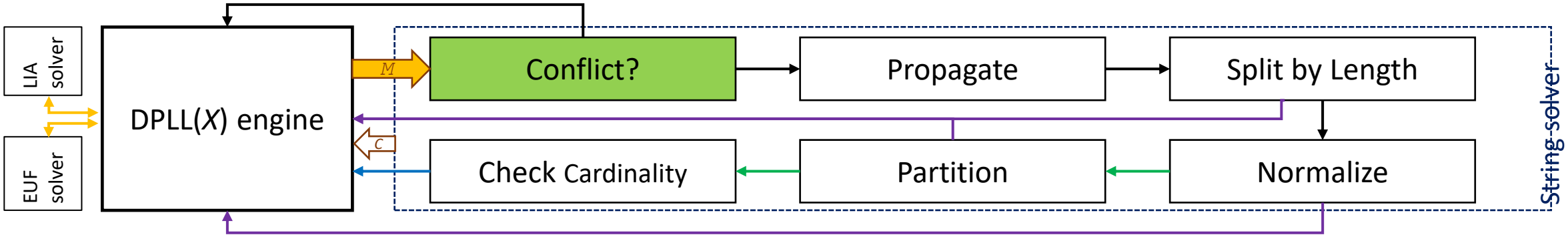
`|x| > |y|`

Arithmetic Solver

`x·z = y·w·"ab"`

String Solver

Conflict Checking



`|x| > |y|`

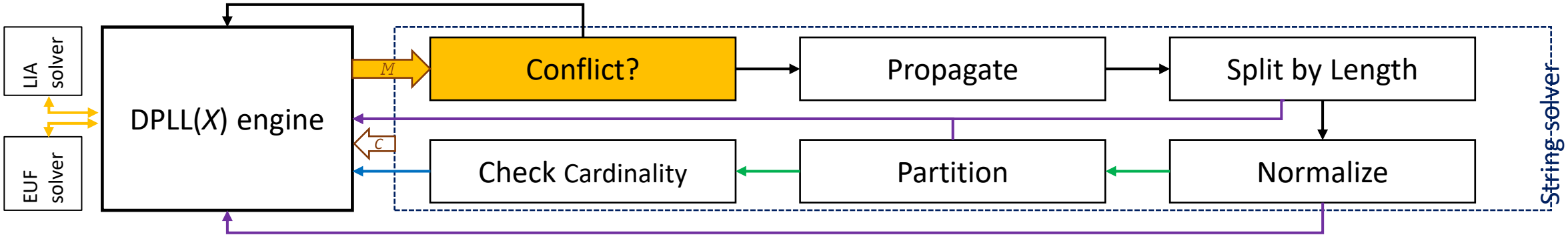
Arithmetic Solver

`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



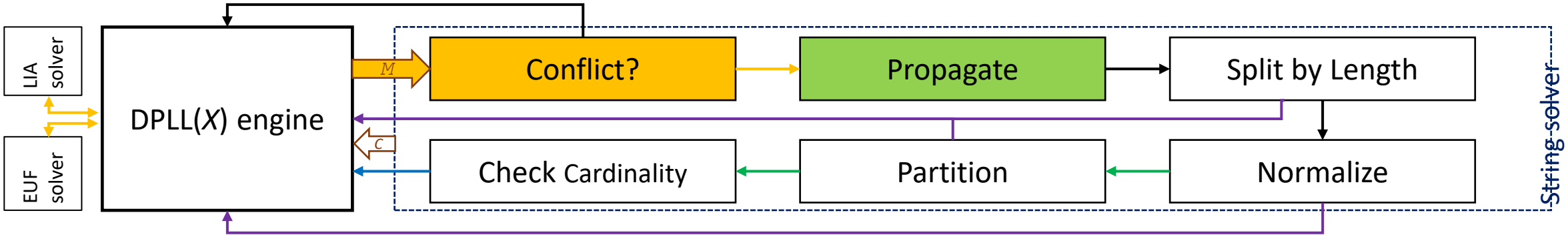
`|x| > |y|`

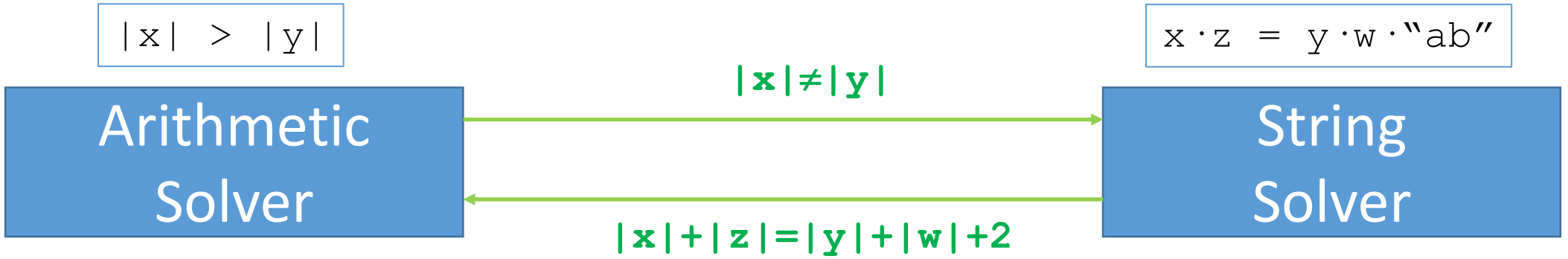
Arithmetic Solver

`x·z = y·w·"ab"`

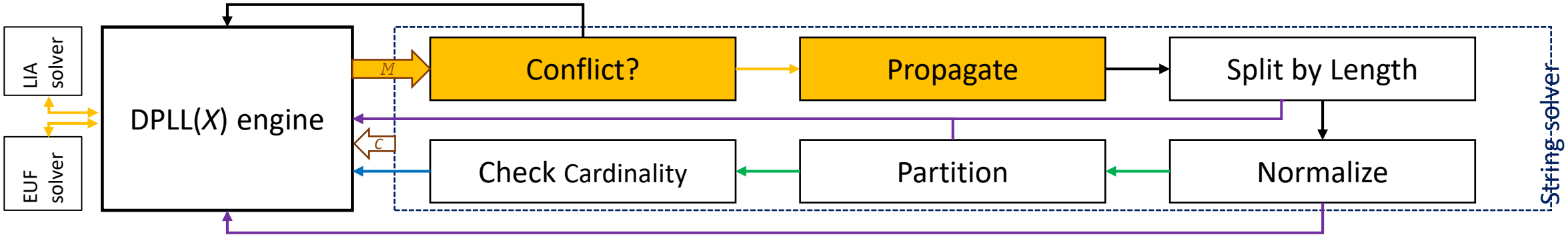
String Solver

Shared Term Propagation





Shared Term Propagation



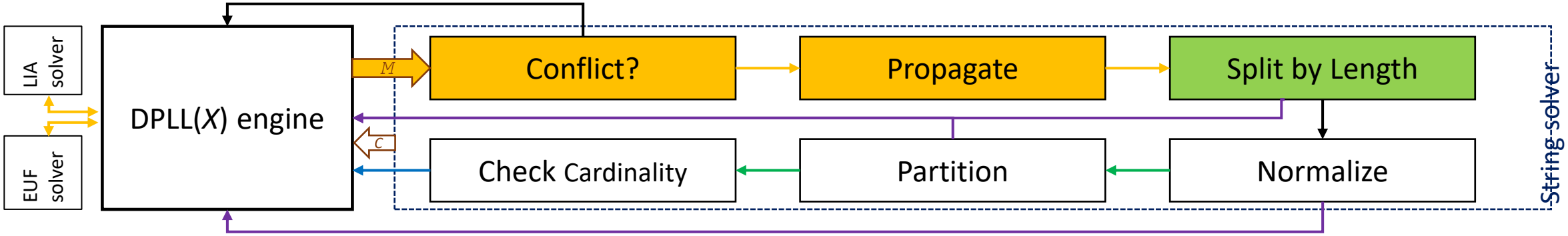
$|x| + |z| = |y| + |w| + 2$
$|x| > |y|$

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

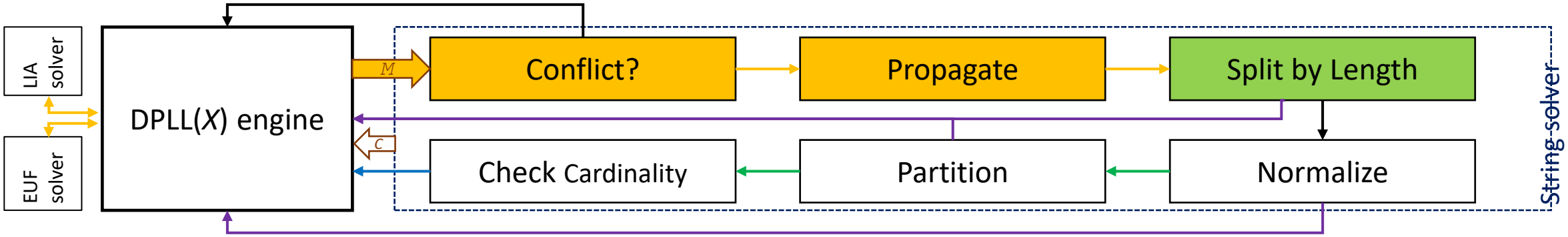
$|x| = 0$
$|y| = 0$
…

$|x| = 0$
$|y| > 0$
…

$|x| > 0$
$|y| = 0$
…

$|x| > 0$
$|y| > 0$
…

Length Splitting



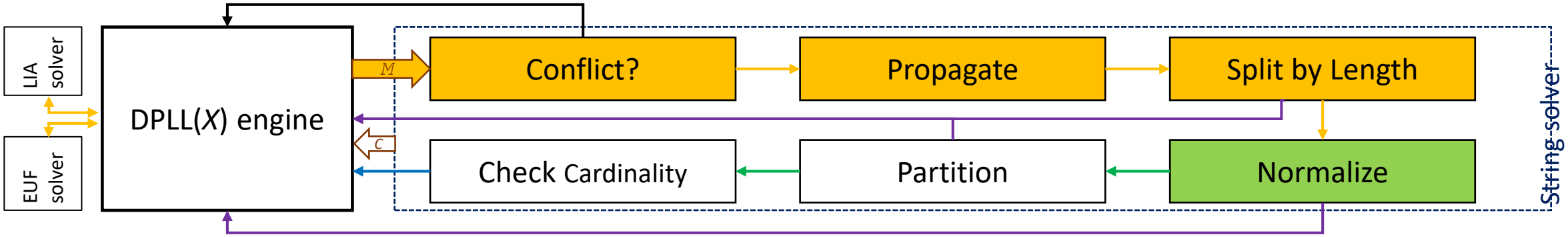
$$|x| + |z| = |y| + |w| + 2$$
$$|x| > |y|$$
$$|x| > 0$$
$$|y| > 0$$

Arithmetic Solver

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

String Solver

Normalization



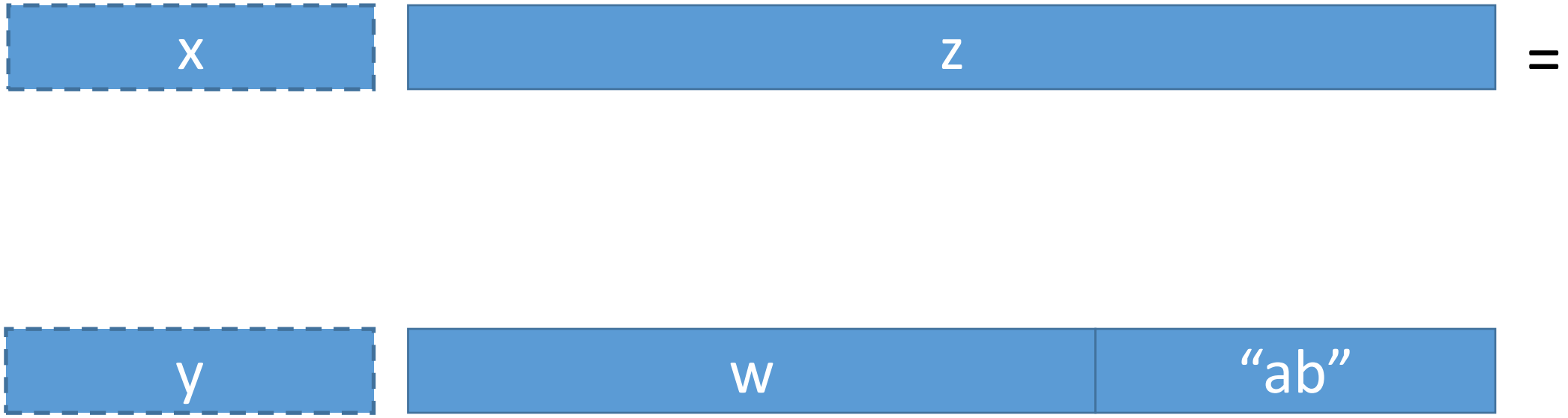
$$|x| \neq |y|$$
$$x \cdot z = y \cdot w \cdot \text{“ab”}$$

String Solver

Normalize Equalities

S { **x** ·z=**y** ·w ·“ab”

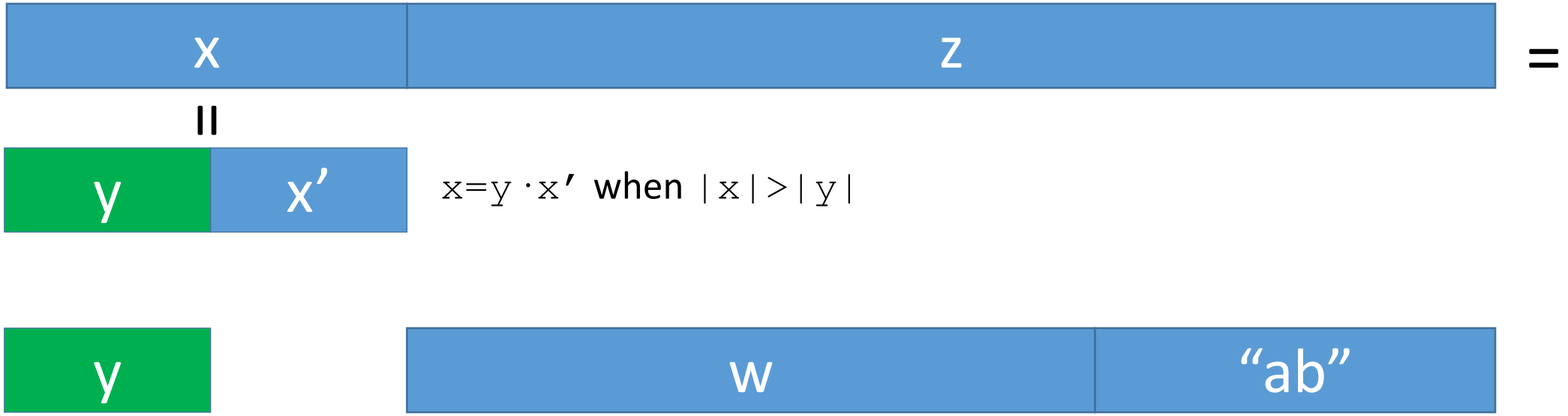
To handle string equalities with prefix **x** and **y** must consider 3 cases…



Normalize Equalities

S { x·z=y·w·"ab"
x=y·x'

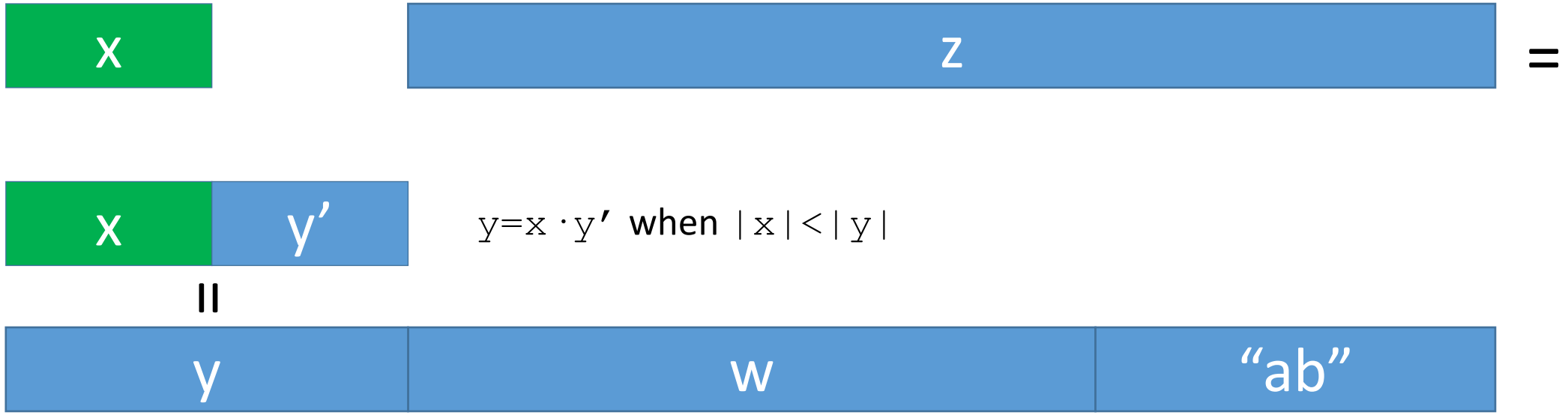
L { |x|>|y|



Normalize Equalities

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

L { $|x| < |y|$ }



Normalize Equalities

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

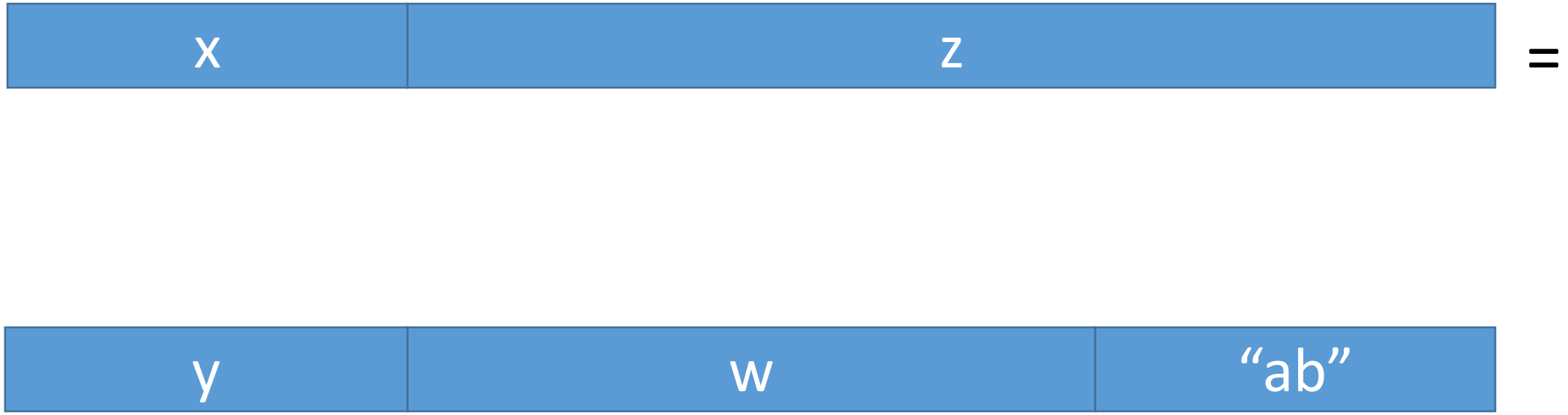
L { $|x| = |y|$ }

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

Normalize Equalities

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

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

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

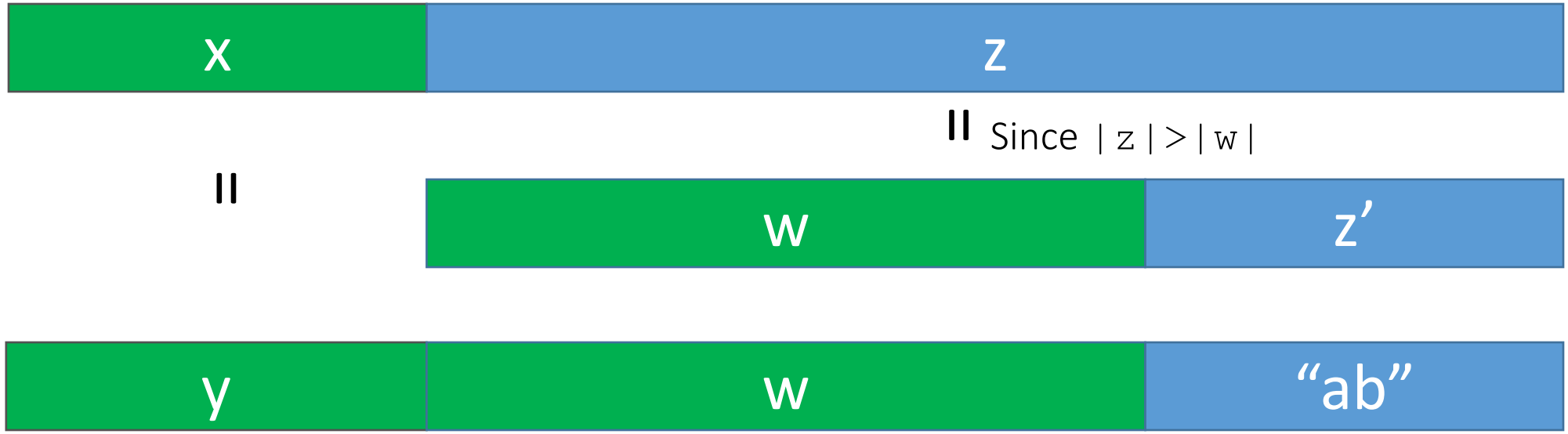
$x = y$

$z = w \cdot z'$

L

$|x| = |y|$

$|z| > |w|$



Normalize Equalities

S

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

$x = y$

$z = w \cdot z'$

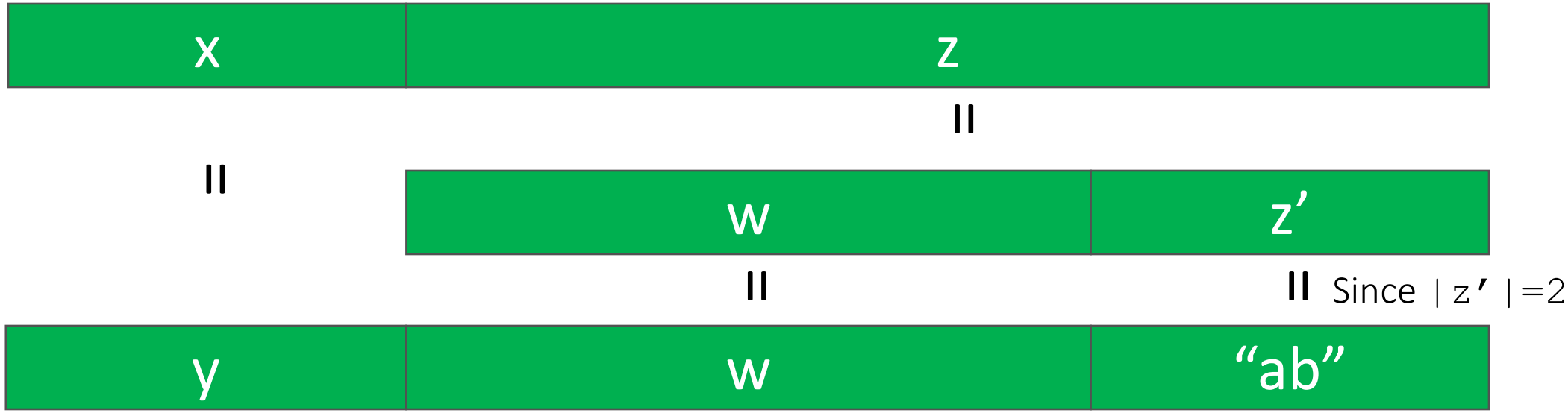
$z' = \text{“ab”}$

L

$|x| = |y|$

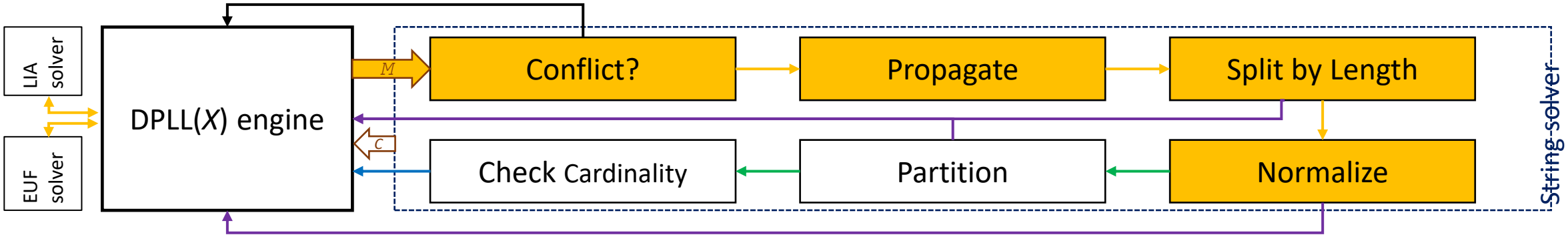
$|z| > |w|$

$|z'| = 2$



$\Rightarrow S \cup L$ is satisfiable

Normalization



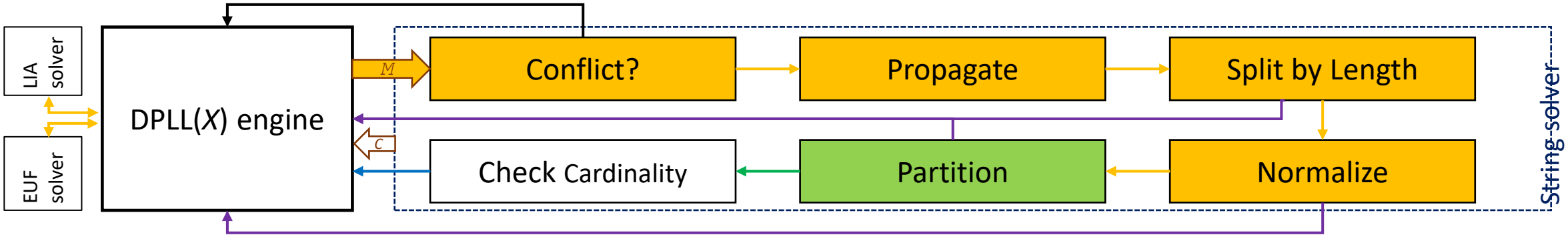
S

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

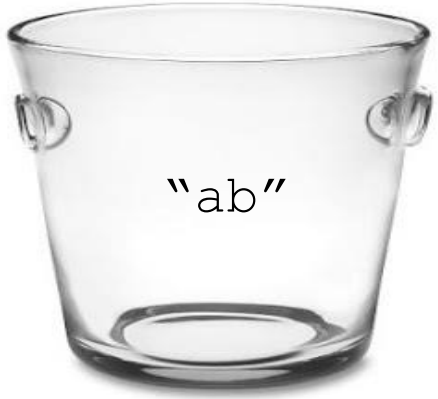
L

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

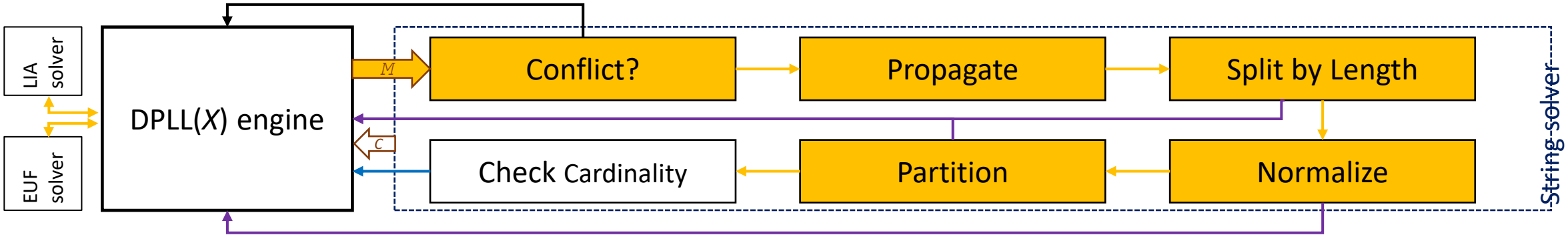
Partition



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



Check Cardinality of Σ

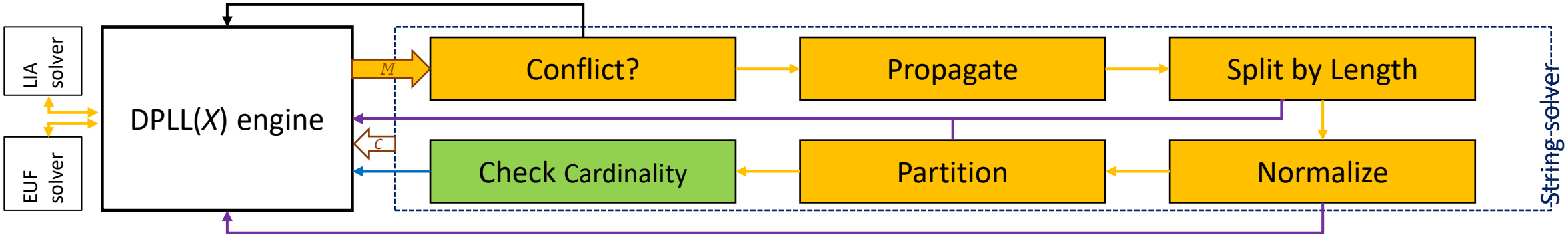


What if, for strings $s_1, s_2, \ldots, s_{256}, s_{257}$:

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

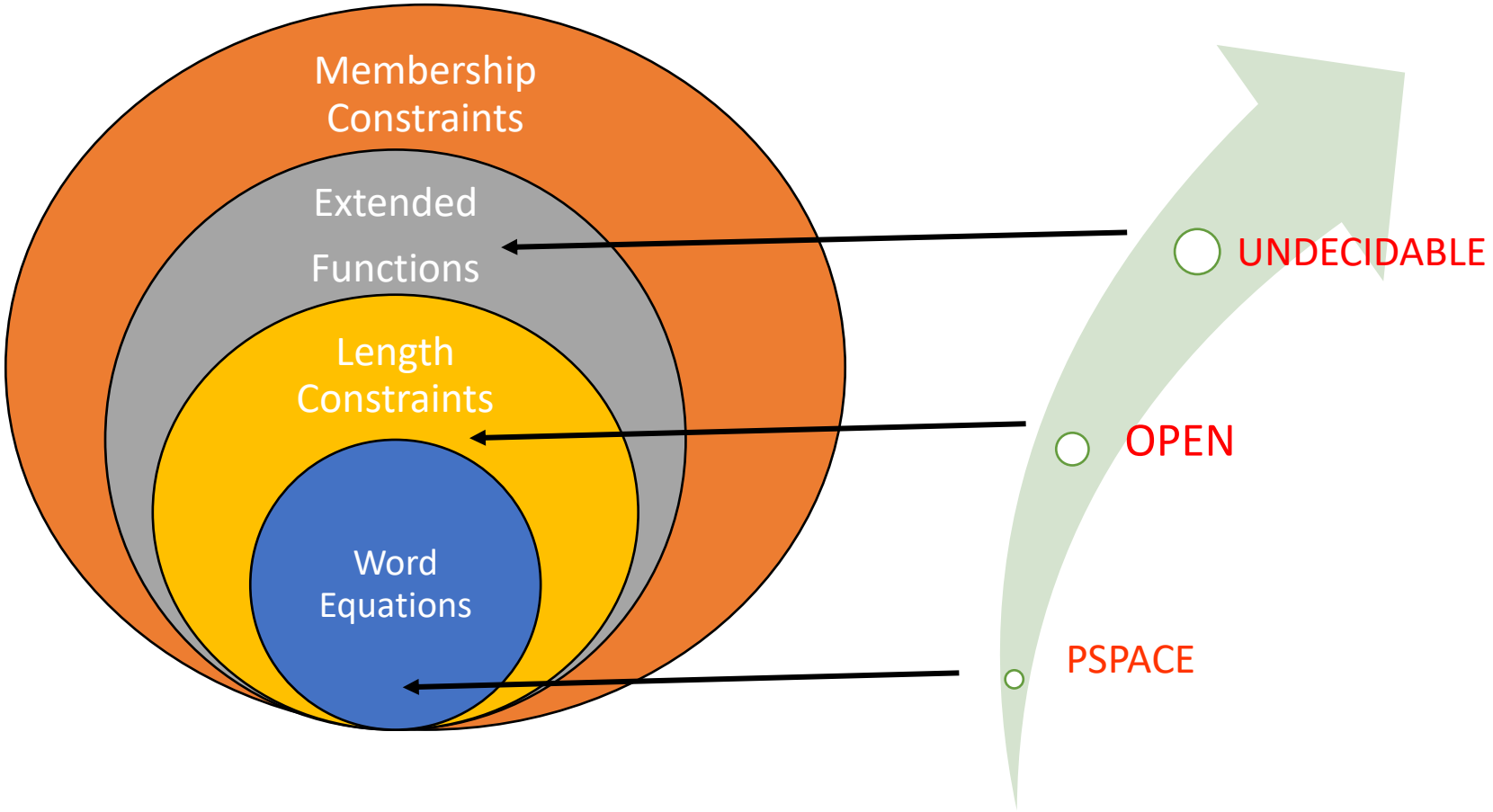


- May be unsatisfiable since Σ is finite
- For instance, if:
  - Σ is a finite alphabet of 256 characters, and
  - $S \cup L$ entails that 257 distinct strings of length 1 exist

  $\Rightarrow$ 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
  - *…Regular expresions*

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: $\neg$`contains(substr(x,0,3),“a”)` $\wedge$ $0 \leq$ `indexof(x,“ab”,0)` < 4

How do we handle Extended String Constraints?

```
¬contains(x,“a”)
```

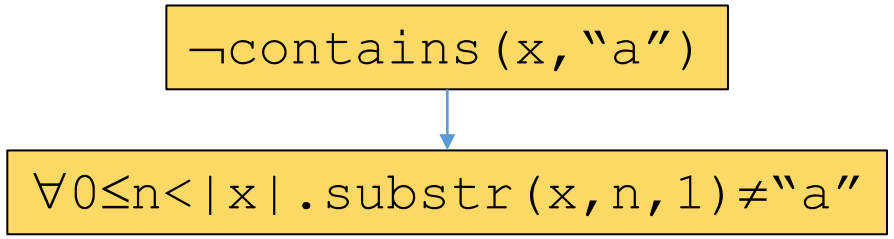
How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

`¬contains(x,“a”)`

How do we handle Extended String Constraints?

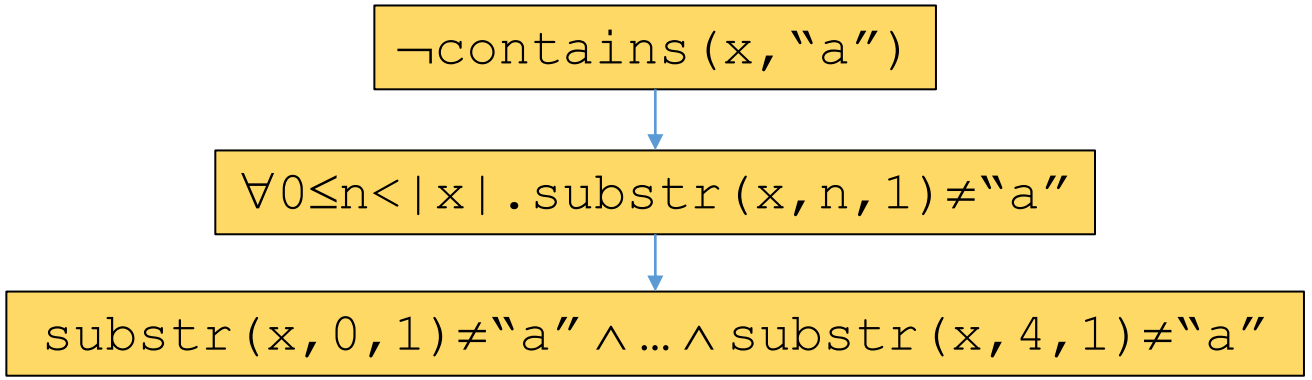
- Naively, by reduction to basic constraints + bounded ∀



Expand `contains`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

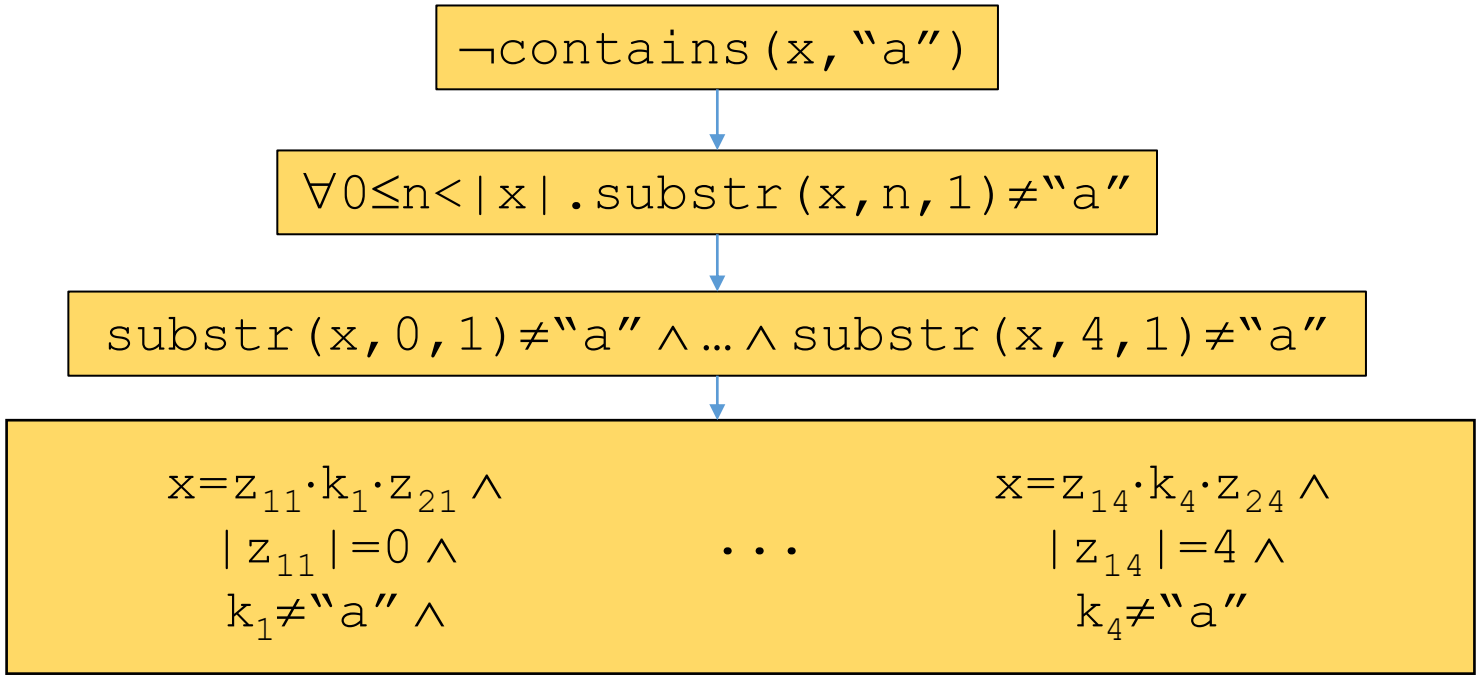


Expand `contains`

Assuming bound `|x|≤5`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀



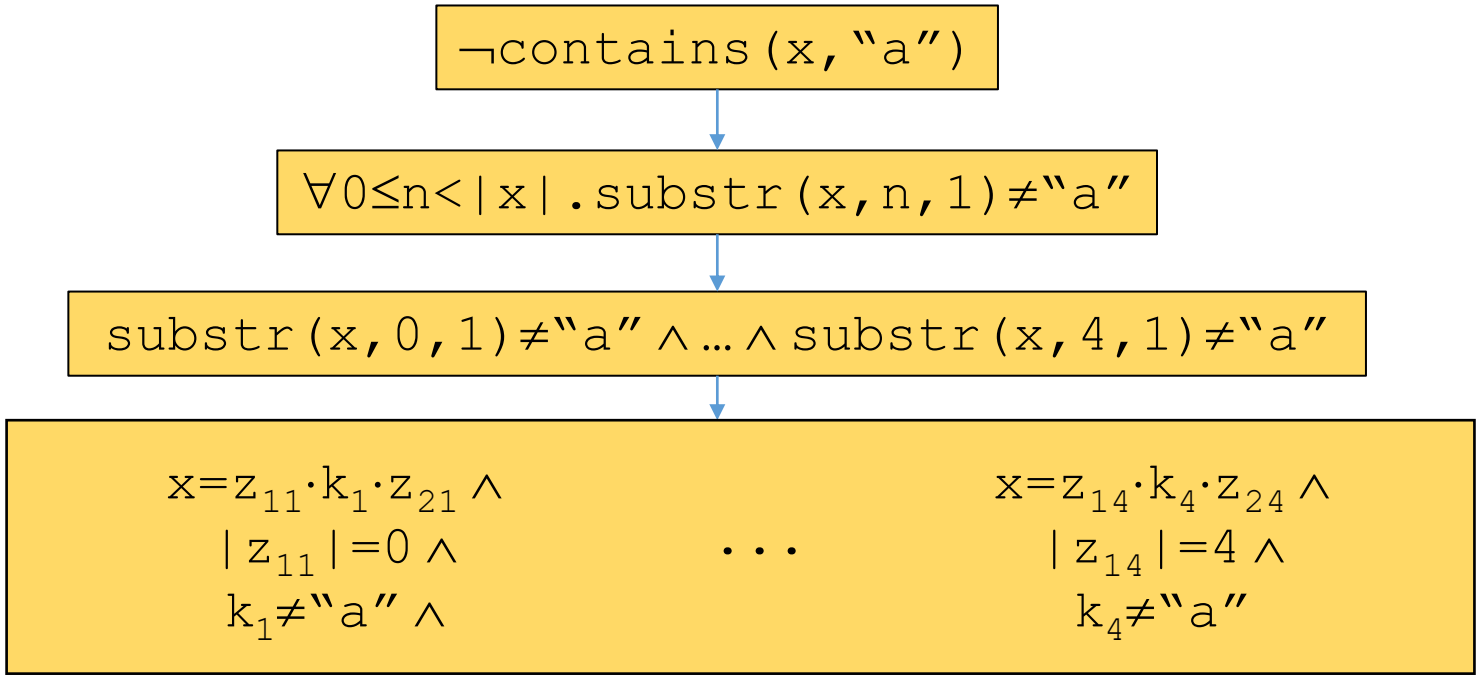
Expand `contains`

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]

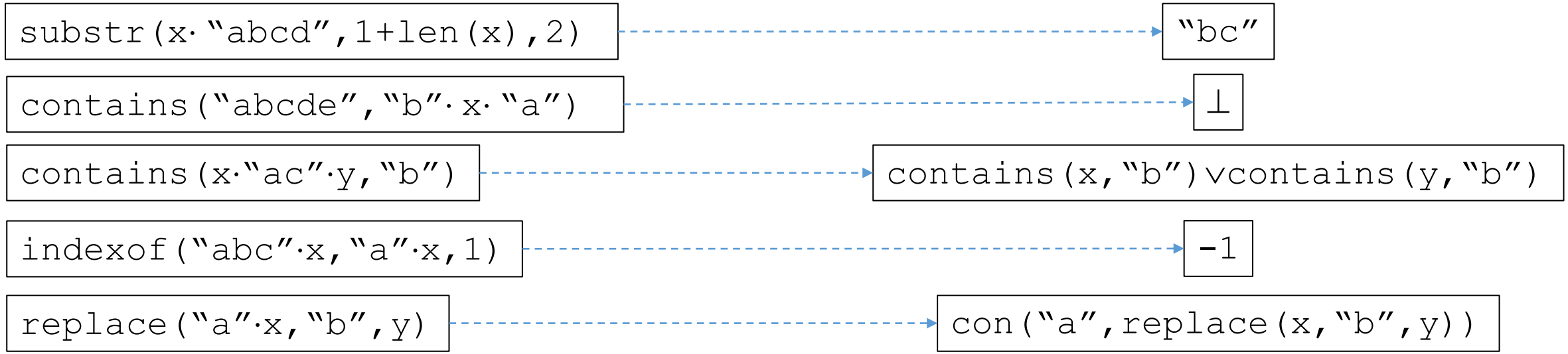
Extended String Constraints

- Context-Dependent Simplifications **[Reynolds et al CAV 2017]**
  - Leverage equalities that hold in current context for simplifying constraints
    - E.g. contains(x,y) simplifies to true in context where y=“”.
- Aggressive Rewrite Rules **[Reynolds et al CAV 2018]**
  - Use high-level reasoning to simplify constraints
- Efficient Encodings of Reductions
  - Reuse variables in reductions of extended functions

Many Simplification Rules for Strings

- Unlike arithmetic:

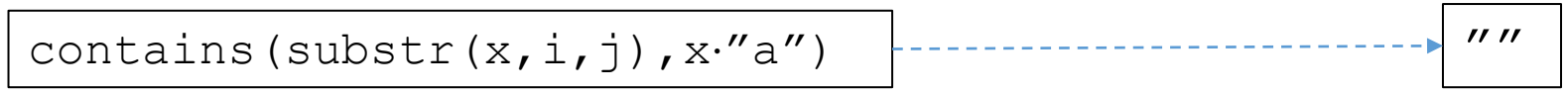
…simplification rules for strings are highly non-trivial:



Simplification based on High-Level Abstractions

[Reynolds/Noetzli/Tinelli/Barrett CAV 19]

- Rules based on high-level abstractions
  - When viewing strings as #characters (e.g. reasoning about their length):



…since the second argument is longer than the first

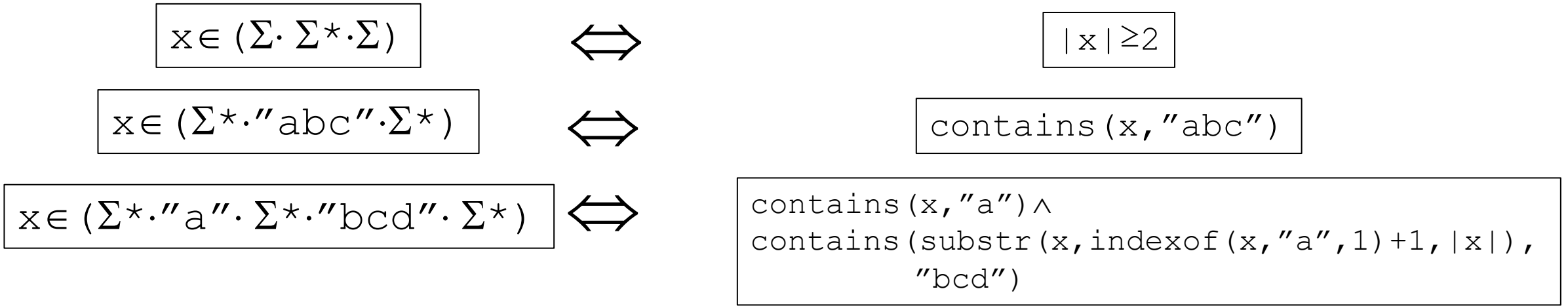
  - When considering the containment relationship between strings:

  - When viewing strings as multisets of characters:

…since LHS contains at least 1 more occurrences of "a"

Regular Expression Elimination

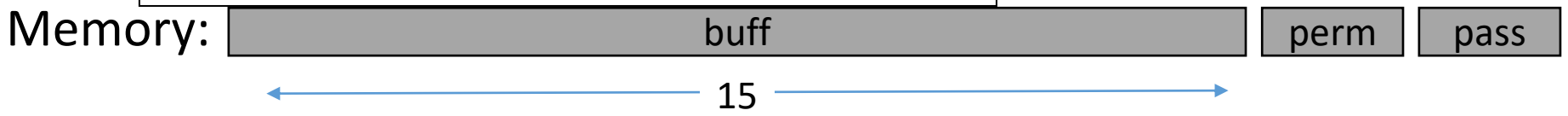
- **Idea:** reduce RE to extended string constraints
  - Possible for many regular expression memberships that occur in practice:



- CVC4 supports aggressive elimination techniques for RE like those above
  - Utilize existing support for extended functions

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' );
   }
```



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' );
   }
```

```
(set-logic QF_SLIA)
(declare-fun mem () String)
(declare-fun buff () String)
(declare-fun permission () String)
(declare-fun pass () String)
(assert (= mem "000000000000000NN"))
(assert (= buff (str.substr mem 0 15)))
(assert (= permission (str.substr mem 15 1)))
(assert (= pass (str.substr mem 16 1)))
(declare-fun input () String)
(declare-fun mem1 () String)
(declare-fun buff1 () String)
(declare-fun permission1 () String)
(declare-fun pass1 () String)
(assert (= (str.len mem1) 17))
(assert (= mem1 (str.++ (str.substr input 0 17)
                        (str.substr mem (str.len input) (str.len mem)))))
(assert (= buff1 (str.substr mem1 0 15)))
(assert (= permission1 (str.substr mem1 15 1)))
(assert (= pass1 (str.substr mem1 16 1)))
(declare-fun mem2 () String)
(declare-fun buff2 () String)
(declare-fun permission2 () String)
(declare-fun pass2 () String)
(assert (str.prefixof "PASSWORD\0" buff1))
(assert (= (str.len mem2) 17))
(assert (= mem2 (str.++ (str.substr mem1 0 15) "Y" "Y")))
(assert (= buff2 (str.substr mem2 0 15)))
(assert (= permission2 (str.substr mem2 15 1)))
(assert (= pass2 (str.substr mem2 16 1)))
(assert (= permission2 "Y"))
(assert (not (= pass2 "Y")))
(check-sat)
```

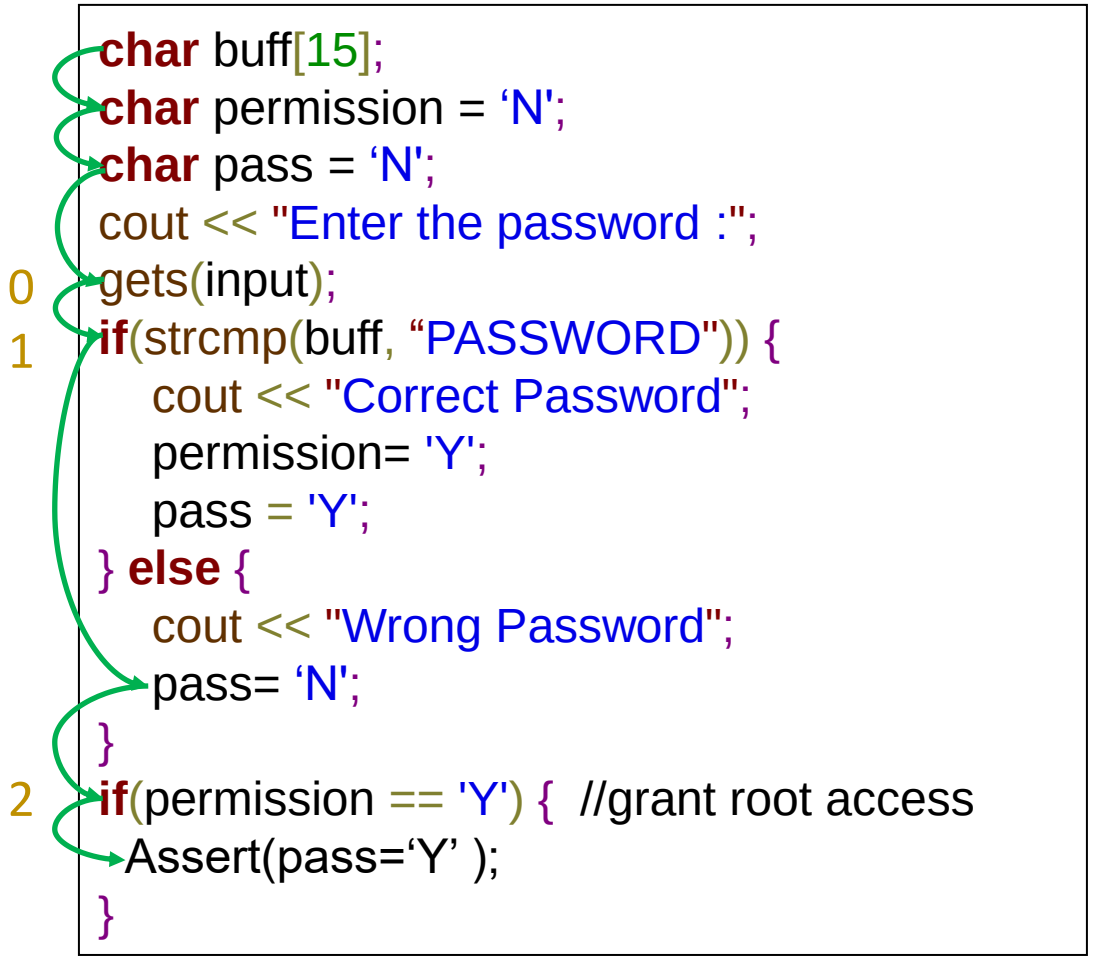
Symbolic Execution

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

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-sym-exec-path1.smt2
unsat
```

```
(set-logic QF_SLIA)
(declare-fun mem () String)
(declare-fun buff () String)
(declare-fun permission () String)
(declare-fun pass () String)
(assert (= mem "000000000000000NN"))
(assert (= buff (str.substr mem 0 15)))
(assert (= permission (str.substr mem 15 1)))
(assert (= pass (str.substr mem 16 1)))
(declare-fun input () String)
(declare-fun mem1 () String)
(declare-fun buff1 () String)
(declare-fun permission1 () String)
(declare-fun pass1 () String)
(assert (= (str.len mem1) 17))
(assert (= mem1 (str.++ (str.substr input 0 17)
                        (str.substr mem (str.len input) (str.len mem)))))
(assert (= buff1 (str.substr mem1 0 15)))
(declare-fun permission2 () String)
(declare-fun pass2 () String)
(assert (str.prefixof "PASSWORD\0" buff1))
(assert (= (str.len mem2) 17))
(assert (= mem2 (str.++ (str.substr mem1 0 15) "Y" "Y")))
(assert (= buff2 (str.substr mem2 0 15)))
(assert (= permission2 (str.substr mem2 15 1)))
(assert (= pass2 (str.substr mem2 16 1)))
(assert (= permission2 "Y"))
(assert (not (= pass2 "Y")))
(check-sat)
```

Symbolic Execution



Symbolic Execution

```
0  char buff[15];
   char permission = 'N';
   char pass = 'N';
   cout << "Enter the password :";
   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' );
   }
```

```
(set-logic QF_SLIA)
(set-option :produce-models true)
(declare-fun mem () String)
(declare-fun buff () String)
(declare-fun permission () String)
(declare-fun pass () String)
(assert (= mem "000000000000000NN"))
(assert (= buff (str.substr mem 0 15)))
(assert (= permission (str.substr mem 15 1)))
(assert (= pass (str.substr mem 16 1)))
(declare-fun input () String)
(declare-fun mem1 () String)
(declare-fun buff1 () String)
(declare-fun permission1 () String)
(declare-fun pass1 () String)
(assert (= (str.len mem1) 17))
(assert (= mem1 (str.++ (str.substr input 0 17)
                        (str.substr mem (str.len input) (str.len mem)))))
(assert (= buff1 (str.substr mem1 0 15)))
(assert (= permission1 (str.substr mem1 15 1)))
(assert (= pass1 (str.substr mem1 16 1)))
(declare-fun mem2 () String)
(declare-fun buff2 () String)
(declare-fun permission2 () String)
(declare-fun pass2 () String)
(assert (not (str.prefixof "PASSWORD\0" buff1)))
(assert (= (str.len mem2) 17))
(assert (= mem2 (str.++ (str.substr mem1 0 16) "N")))
(assert (= buff2 (str.substr mem2 0 15)))
(assert (= permission2 (str.substr mem2 15 1)))
(assert (= pass2 (str.substr mem2 16 1)))
(assert (= permission2 "Y"))
(assert (not (= pass2 "Y")))
(check-sat)
(get-model)
```

Symbolic Execution

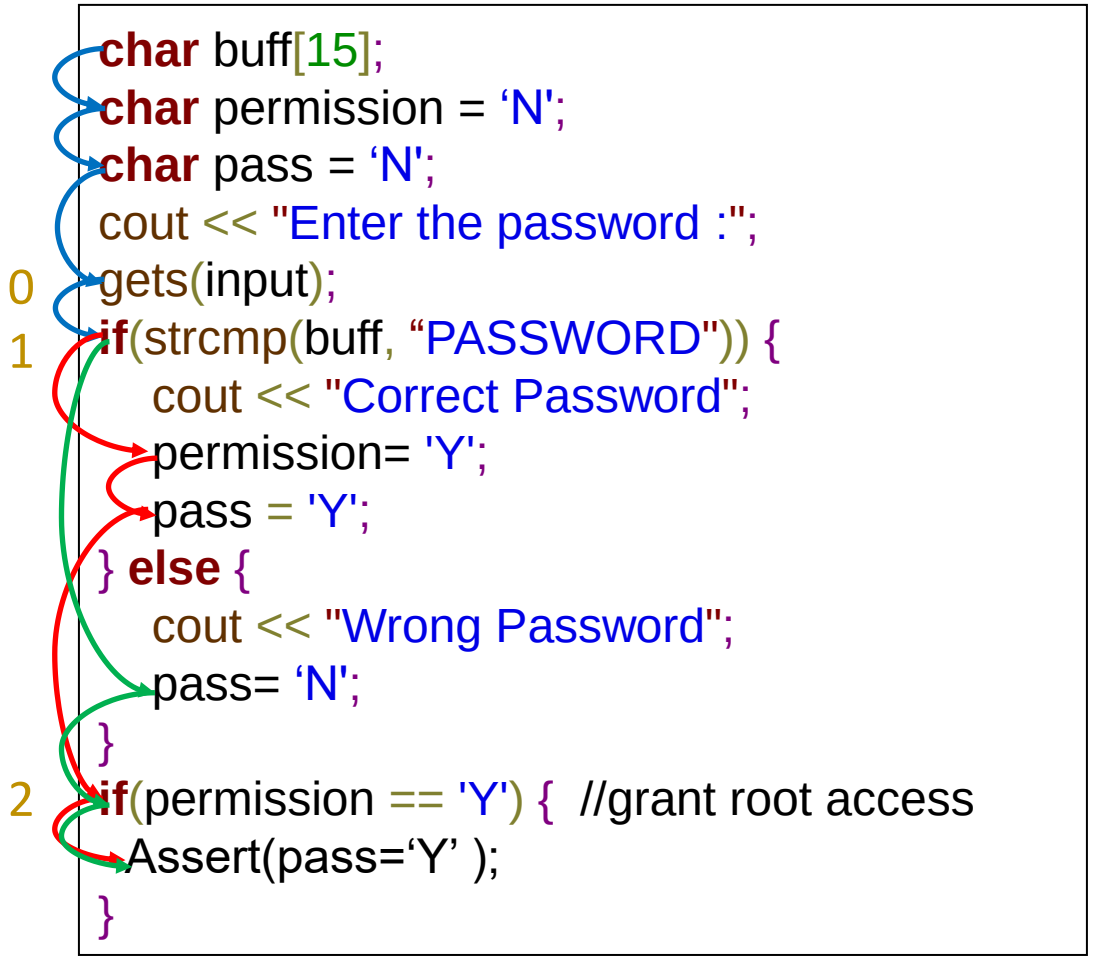
```
char buff[15];
char permission = 'N';
char pass = 'N';
cout << "Enter the password :";
gets(input);
if(strcmp(buff, "PASSWORD")) {
    cout <<
    permiss
    pass = '
} else {
    cout <<
    pass= 'I
}
if(permiss
Assert(pa
}
```

0
1
2

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-sym-exec-path2.smt2
sat
(model
(define-fun mem () String "000000000000000NN")
(define-fun buff () String "000000000000000")
(define-fun permission () String "N")
(define-fun pass () String "N")
(define-fun input () String "AAAAAAAAAAAAAAAYA")
(define-fun mem1 () String "AAAAAAAAAAAAAAAYA")
(define-fun buff1 () String "AAAAAAAAAAAAAAA")
(define-fun permission1 () String "Y")
(define-fun pass1 () String "A")
(define-fun mem2 () String "AAAAAAAAAAAAAAAYN")
(define-fun buff2 () String "AAAAAAAAAAAAAAA")
(define-fun permission2 () String "Y")
(define-fun pass2 () String "N")
)
```

```
(set-logic QF_SLIA)
(set-option :produce-models true)
(declare-fun mem () String)
(declare-fun buff () String)
(declare-fun permission () String)
(declare-fun pass () String)
(assert (= mem "000000000000000NN"))
(assert (= buff (str.substr mem 0 15)))
(assert (= permission (str.substr mem 15 1)))
(assert (= pass (str.substr mem 16 1)))
(declare-fun input () String)
(declare-fun mem1 () String)
(declare-fun buff1 () String)
(declare-fun permission1 () String)
(declare-fun pass1 () String)
(assert (= (str.len mem1) 17))
(assert (= mem1 (str.++ (str.substr input 0 17)
                        (str.substr mem (str.len input) (str.len mem)))))
                                                    .substr mem1 0 15)))
                                                    on1 (str.substr mem1 15 1)))
                                                    r.substr mem1 16 1)))
                                                     () String)
                                                    () String)
                                                    ssion2 () String)
                                                    () String)
                                                    efixof "PASSWORD\0" buff1)))
                                                    mem2) 17))
                                                    r.++ (str.substr mem1 0 16) "N")))
                                                    .substr mem2 0 15)))
                                                    on2 (str.substr mem2 15 1)))
                                                    r.substr mem2 16 1)))
                                                    on2 "Y"))
                                                    2 "Y")))
```

Symbolic Execution

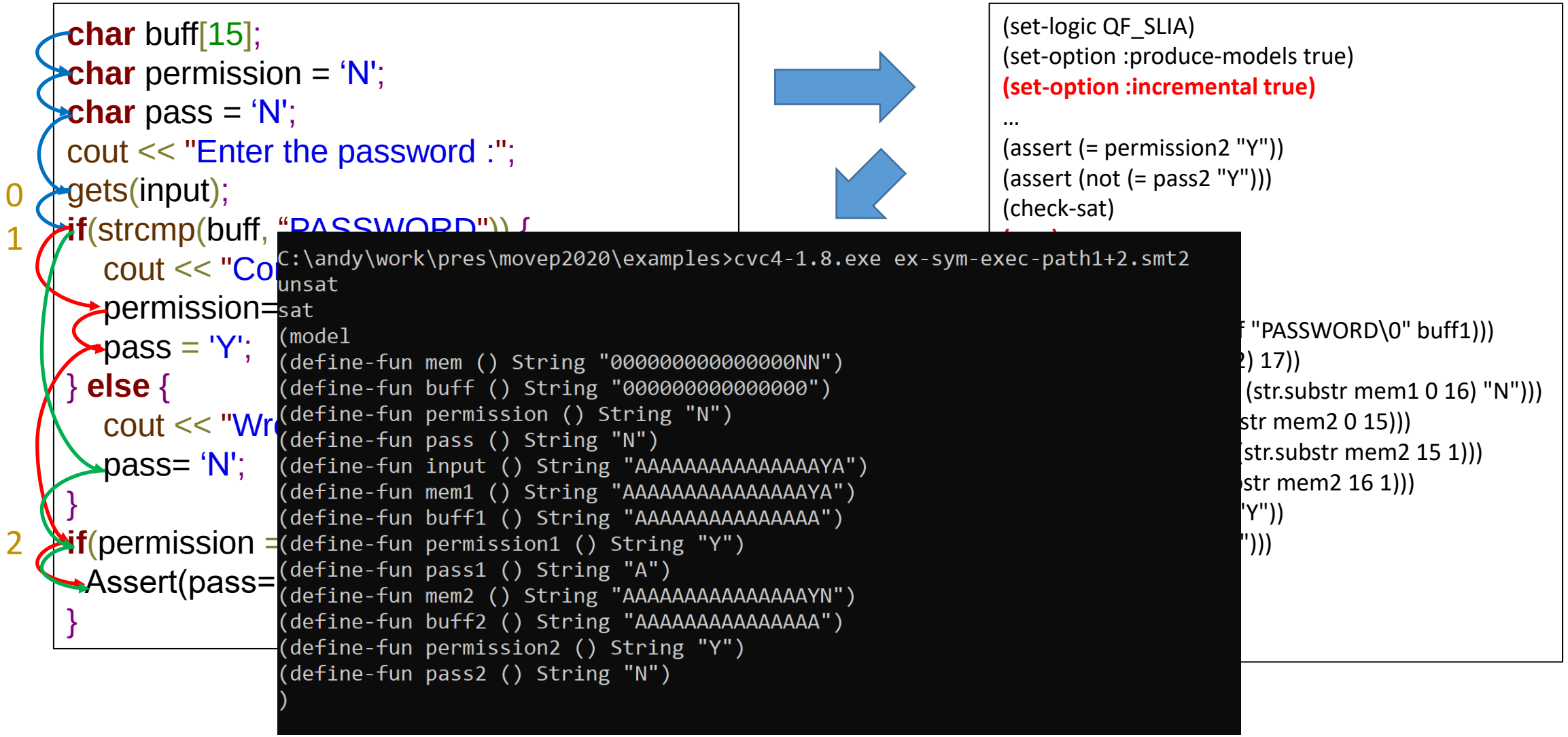


Symbolic Execution

```
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' );
   }
```

```
(set-logic QF_SLIA)
(set-option :produce-models true)
(set-option :incremental true)
…
(assert (= permission2 "Y"))
(assert (not (= pass2 "Y")))
(check-sat)
(pop)

(push)
(assert (not (str.prefixof "PASSWORD\0" buff1)))
(assert (= (str.len mem2) 17))
(assert (= mem2 (str.++ (str.substr mem1 0 16) "N")))
(assert (= buff2 (str.substr mem2 0 15)))
(assert (= permission2 (str.substr mem2 15 1)))
(assert (= pass2 (str.substr mem2 16 1)))
(assert (= permission2 "Y"))
(assert (not (= pass2 "Y")))
(check-sat)
(get-model)
(pop)
```

Symbolic Execution



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"
}
```

…vulnerability detected automatically by the SMT solver

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;
}
```

Loop and execute commands:

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);
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;
}
```

```
(set-logic QF_SLIA)
(set-option :produce-models true)
(set-option :incremental true)
(set-option :strings-exp true)
(declare-fun input () String)(declare-fun data () String)
(assert (= data (str.++ "p" input ";xcvc4;")))
(declare-fun index1 () Int)(declare-fun end1 () Int)
(declare-fun exe1 () String)(declare-fun cmd1 () String)
(assert (= index1 0))
(assert (= end1 (str.indexof data ";" index1)))
(assert (= exe1 (str.substr data index1 1)))
(assert (= cmd1 (str.substr data (+ index1 1) (- end1 (+ index1 1)))))
(assert (< index1 (str.len data)))
(push)
; is assertion violated on this iteration?
(assert (not (= cmd1 "")))
(assert (= (str.substr data index1 1) "x"))
(assert (not (= cmd1 "cvc4")))
(check-sat)
(pop)
(declare-fun index2 () Int)(declare-fun end2 () Int)
(declare-fun exe2 () String)(declare-fun cmd2 () String)
(assert (= index2 (+ end1 1)))
(assert (= end2 (str.indexof data ";" index2)))
(assert (= exe2 (str.substr data index2 1)))
(assert (= cmd2 (str.substr data (+ index2 1) (- end2 (+ index2 1)))))
(assert (< index2 (str.len data)))
(push)
; is assertion violated on this iteration?
(assert (not (= cmd2 "")))
(assert (= (str.substr data index2 1) "x"))
(assert (not (= cmd2 "cvc4")))
(check-sat)
(get-model)
(pop)
```

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,in
      cout << curr_c
    }else if(substr(c
      exec(cmd);
      Assert(cmd==
    }else{
      cout << “Bad c
    }
  }
  index=end+1;
}
```

```
C:\andy\work\pres\movep2020\examples>cvc4-1.8.exe ex-sym-exec-2.smt2
unsat
sat
(model
(define-fun input () String ";xA")
(define-fun data () String "p;xA;xcvc4;")
(define-fun index1 () Int 0)
(define-fun end1 () Int 1)
(define-fun exe1 () String "p")
(define-fun cmd1 () String "")
(define-fun index2 () Int 2)
(define-fun end2 () Int 4)
(define-fun exe2 () String "x")
(define-fun cmd2 () String "A")
)
```

```
(set-logic QF_SLIA)
(set-option :produce-models true)
(set-option :incremental true)
(set-option :strings-exp true)
(declare-fun input () String)(declare-fun data () String)
(assert (= data (str.++ "p" input ";xcvc4;")))
(declare-fun index1 () Int)(declare-fun end1 () Int)
(declare-fun exe1 () String)(declare-fun cmd1 () String)
(assert (= index1 0))
(assert (= end1 (str.indexof data ";" index1)))
(assert (= exe1 (str.substr data index1 1)))
(assert (= cmd1 (str.substr data (+ index1 1) (- end1 (+ index1 1)))))
(assert (< index1 (str.len data)))
(push)
; is assertion violated on this iteration?
(assert (not (= cmd1 "")))
(assert (= (str.substr data index1 1) "x"))
(assert (not (= cmd1 "cvc4")))
```

```
t)(declare-fun end2 () Int)
ng)(declare-fun cmd2 () String)
1 1)))
xof data ";" index2)))
r data index2 1)))
tr data (+ index2 1) (- end2 (+ index2 1)))))
 data)))

 this iteration?
)
a index2 1) "x"))
c4")))
```

Symbolic Execution #2

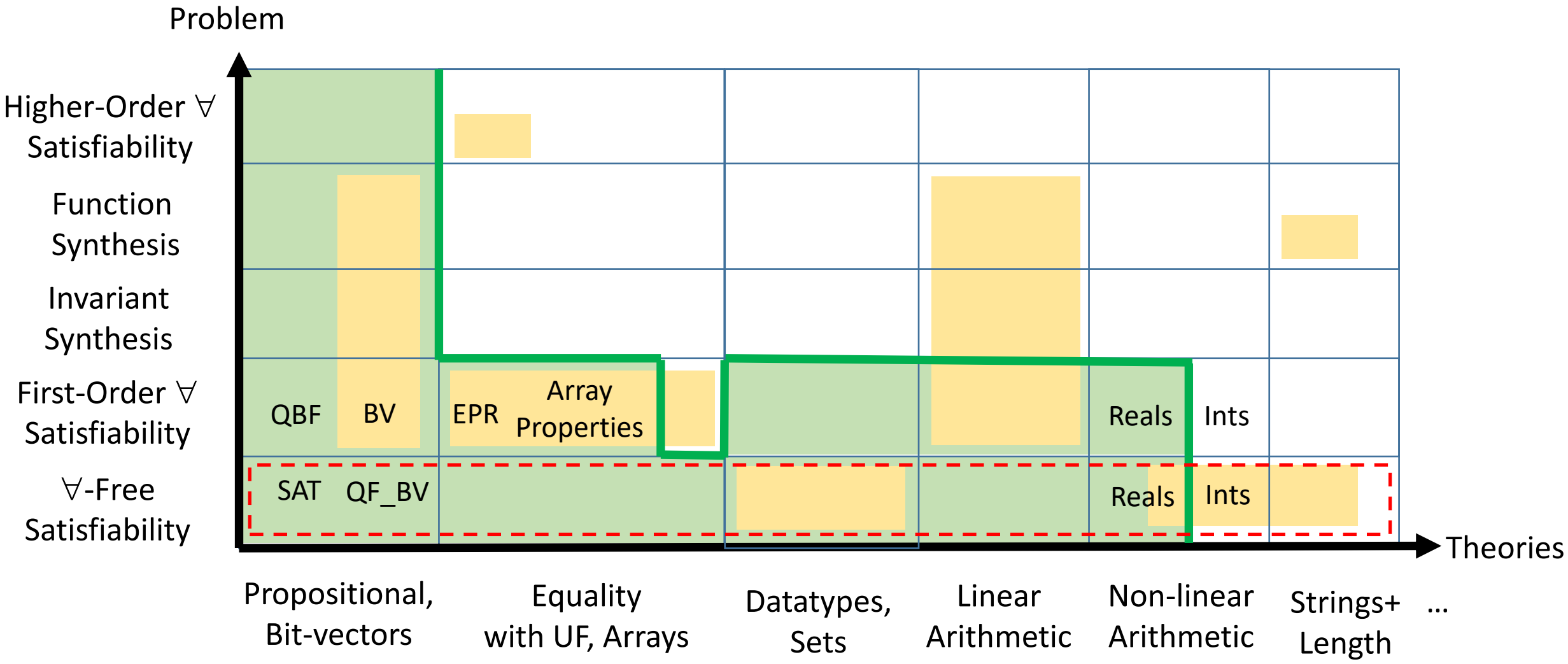
```
cout << "Enter text to print :";
string input;
gets(input);   ← “;xA”
string data= ‘p‘ ++ input ++ ‘;xcvc4;‘;
int index=0;
while(index<str.len(data)){
  int end=indexof(data,’;’,index+1);
  string cmd=substr(data,index+1,end-(index+1));
  if(cmd!=“”){
    if(substr(data,index,1)=‘p’){
      cout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

Theories Not Covered:

- Non-linear Arithmetic **[Franzel et al 2007, Jovanovic/de Moura 2011, …]**
- Floating-point **[Brain et al 2012]**
- Co-inductive datatypes (e.g. streams) **[Reynolds/Blanchette 2015]**
- Separation logic
- Even more in development: Graphs, Heaps, Multisets, Tables, Posits, …

$\Rightarrow$ *Quantifiers* $\forall\exists$

Ongoing work in SMT



Quantified formulas ∀ in SMT

- Are of importance to applications:
  - Automated theorem proving:
    - Background axioms {∀x.g(e,x)=g(x,e)=x, ∀x.g(x,g(y,z))=g(g(x,y),x),∀x.g(x,i(x))=e}
  - Software verification:
    - Unfolding ∀x.foo(x)=bar(x+1), code contracts ∀x.pre(x)⇒post(f(x))
    - Frame axioms ∀x.x≠t ⇒ A'(x)=A(x)
  - Function Synthesis: ∀i:input.∃o:output.R[o,i]
  - Planning: ∃p:plan.∀t:time.F[P,t]
  - Knowledge representation: ∀xy:Person.sibling(x,y)⇒mother(x)=mother(y)

Quantified formulas ∀ in SMT

- Are of importance to applications:
  - Automated theorem proving:
    - Background axioms {∀x.g(e,x)=g(x,e)=x, ∀x.g(x,g(y,z))=g(g(x,y),x),∀x.g(x,i(x))=e}
  - Software verification:
    - Unfolding ∀x.foo(x)=bar(x+1), code contracts ∀x.pre(x)⇒post(f(x))
    - Frame axioms ∀x.x≠t ⇒ A'(x)=A(x)
  - Function Synthesis: ∀i:input.∃o:output.R[o,i]
  - Planning: ∃p:plan.∀t:time.F[P,t]
  - Knowledge representation: ∀xy:Person.sibling(x,y)⇒mother(x)=mother(y)
- Are very challenging in theory:
  - Establishing T-satisfiability of formulas with ∀ is generally undecidable

Quantified formulas ∀ in SMT

- Are of importance to applications:
  - Automated theorem proving:
    - Background axioms {∀x.g(e,x)=g(x,e)=x, ∀x.g(x,g(y,z))=g(g(x,y),x),∀x.g(x,i(x))=e}
  - Software verification:
    - Unfolding ∀x.foo(x)=bar(x+1), code contracts ∀x.pre(x)⇒post(f(x))
    - Frame axioms ∀x.x≠t ⇒ A'(x)=A(x)
  - Function Synthesis: ∀i:input.∃o:output.R[o,i]
  - Planning: ∃p:plan.∀t:time.F[P,t]
  - Knowledge representation: ∀xy:Person.sibling(x,y)⇒mother(x)=mother(y)
- Are very challenging in theory:
  - Establishing T-satisfiability of formulas with ∀ is generally undecidable
- Can be handled well in practice:
  - Efficient decision procedures for decidable fragments
  - Heuristic techniques have high success rates in the general case

Solvers for ∀

- First order theorem provers focus on ∀ reasoning
  …but have been extended in the past decade to theory reasoning:
  - **Vampire, E, SPASS**
    - First-order resolution + superposition **[Robinson 65, Nieuwenhuis/Rubio 99, Prevosto/Waldman 06]**
    - AVATAR **[Voronkov 14, Reger et al 15]**
  - **iProver**
    - InstGen calculus **[Ganzinger/Korovin 03]**
  - **Princess, Beagle**, …
- SMT solvers focus mostly on quantifier-free theory reasoning
  …but have been extended in the past decade to ∀ reasoning:
  - **Z3, CVC4, VeriT, Alt-Ergo**
    - Some superposition-based **[deMoura et al 09]**
    - Mostly instantiation-based **[Detlefs et al 05, deMoura et al 07, Ge et al 07, …]**

Solvers for ∀

- First order theorem provers focus on ∀ reasoning
  …but have been extended in the past decade to theory reasoning:
  - **Vampire, E, SPASS**
    - First-order resolution + superposition **[Robinson 65, Nieuwenhuis/Rubio 99, Prevosto/Waldman 06]**
    - AVATAR **[Voronkov 14, Reger et al 15]**
  - **iProver**
    - InstGen calculus **[Ganzinger/Korovin 03]**
  - **Princess, Beagle**, …
- SMT solvers focus mostly on quantifier-free theory reasoning
  …but have been extended in the past decade to ∀ reasoning:
  - **Z3, CVC4, VeriT, Alt-Ergo**
    - Some superposition-based **[deMoura et al 09]**
    - Mostly **instantiation-based** **[Detlefs et al 05, deMoura et al 07, Ge et al 07, …]**

⟹ Focus of the remainder of this talk

Quantifiers

- Universal quantification:

$$\underbrace{\forall x{:}\mathtt{Int}.P(x)}$$

P is true for all integers x

- Existential quantification:

$$\underbrace{\exists x{:}\mathtt{Int}.\neg Q(x)}$$

Q is false for some integer x

Quantifiers

- Universal quantification:

$$\underbrace{\forall x{:}\mathtt{Int}.P(x)}$$

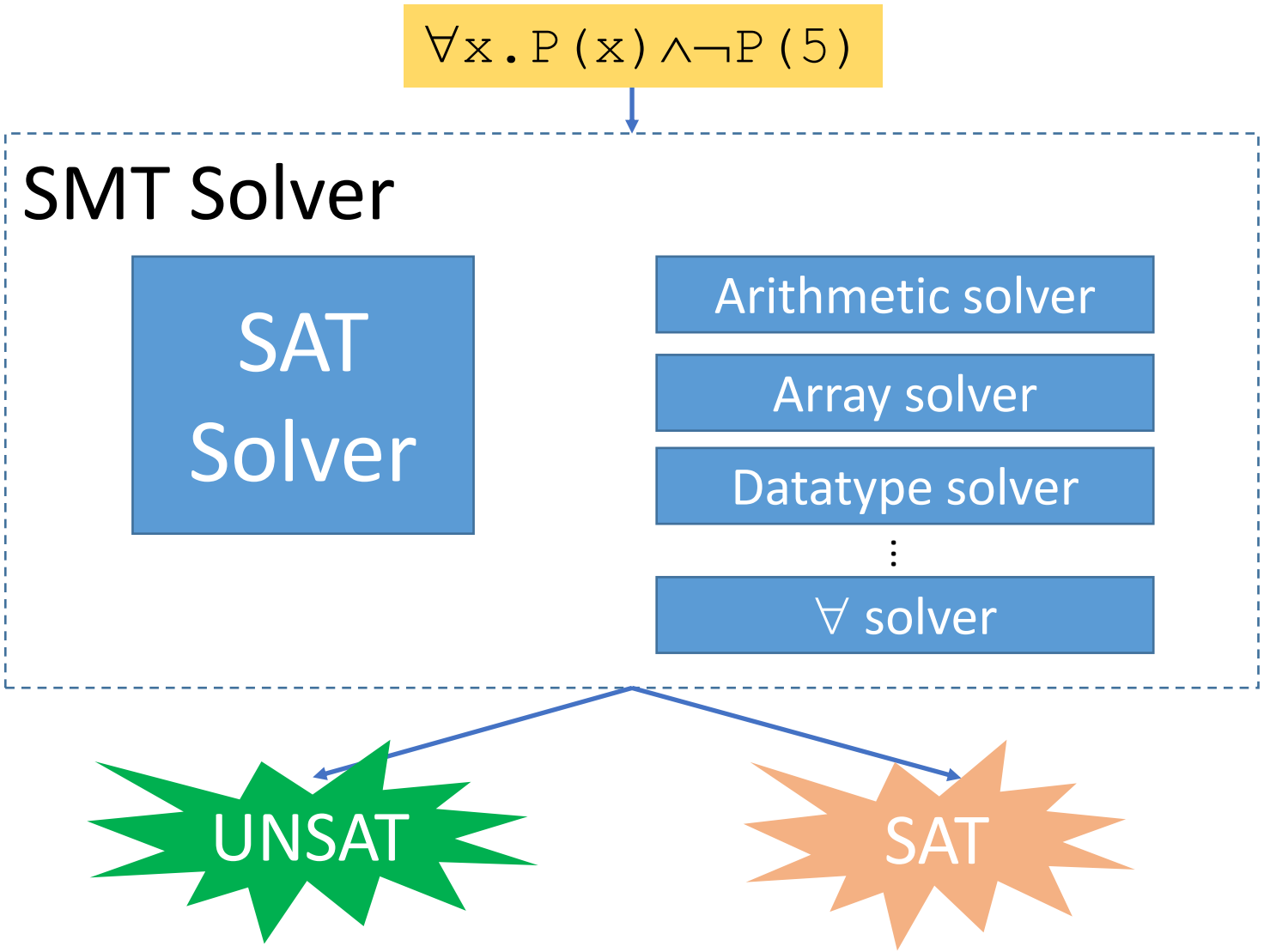
P is true for all integers x

- Existential quantification:

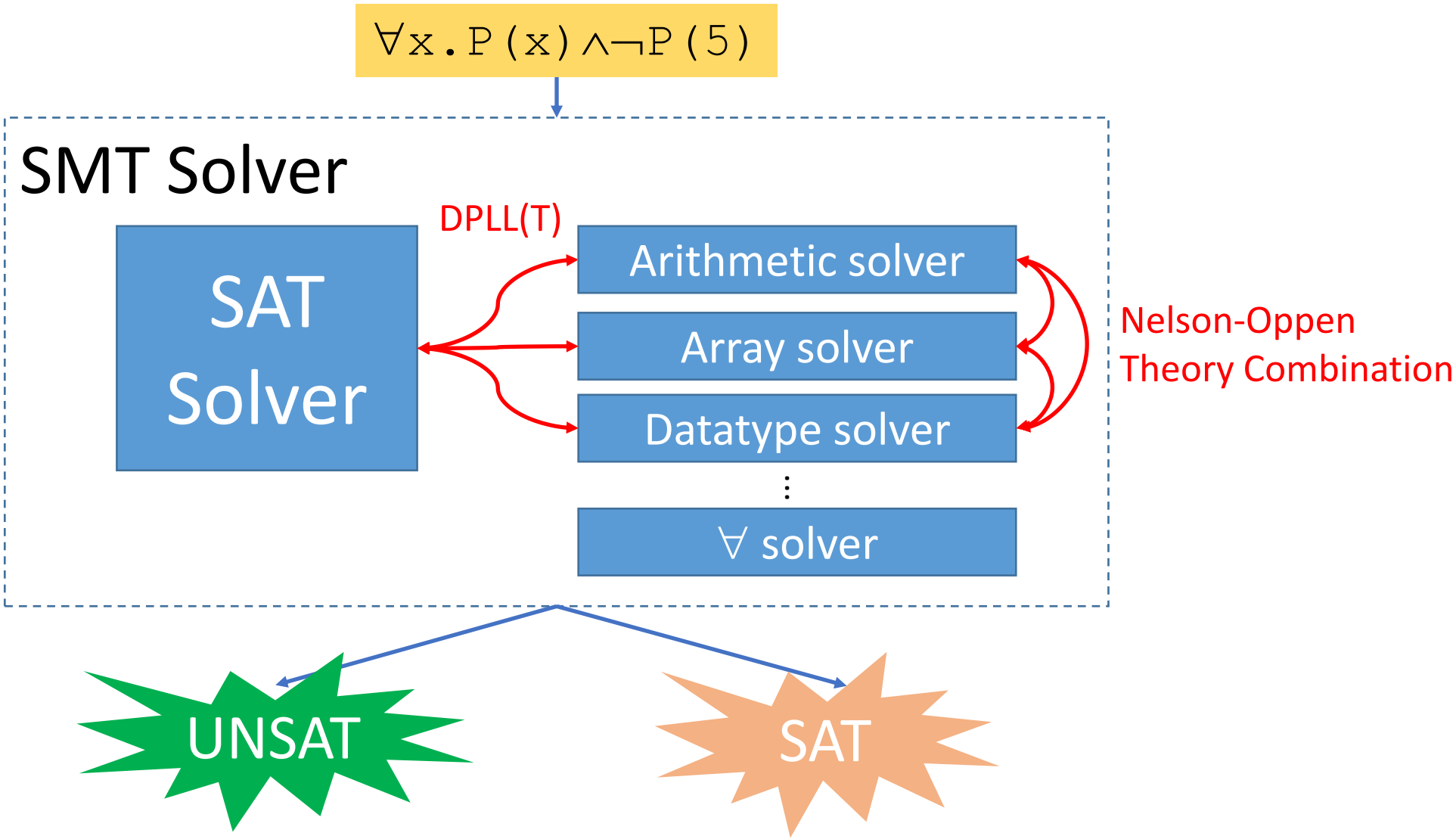
$$\exists x{:}\mathtt{Int}.\neg Q(x) \quad \rightarrow \quad \mathbf{\neg \forall x{:}Int.Q(x)}$$

$\Rightarrow$ For consistency, assume existential quantification is rewritten as universal quantification

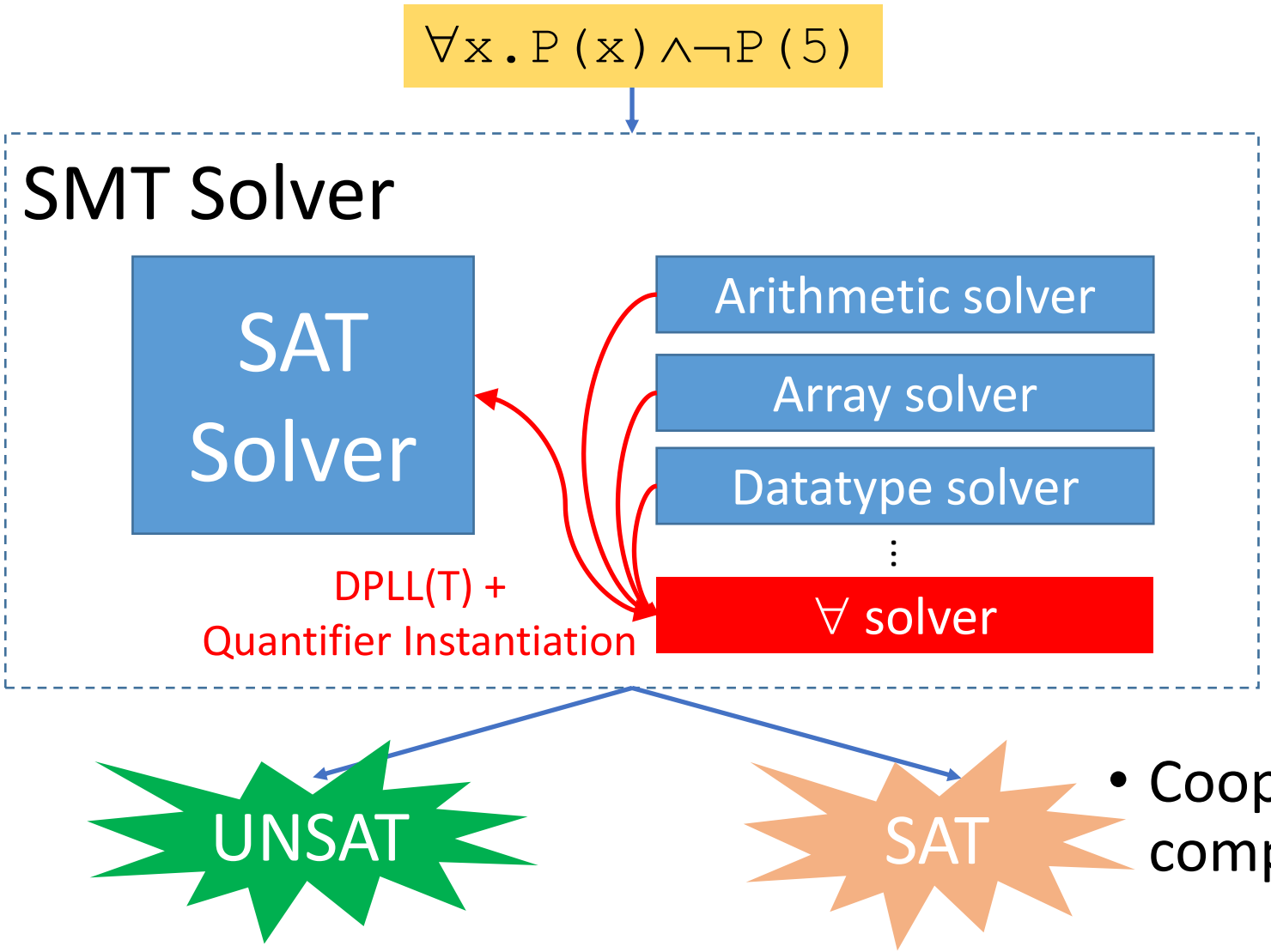
Satisfiability Modulo Theories (SMT) Solvers



Satisfiability Modulo Theories (SMT) Solvers

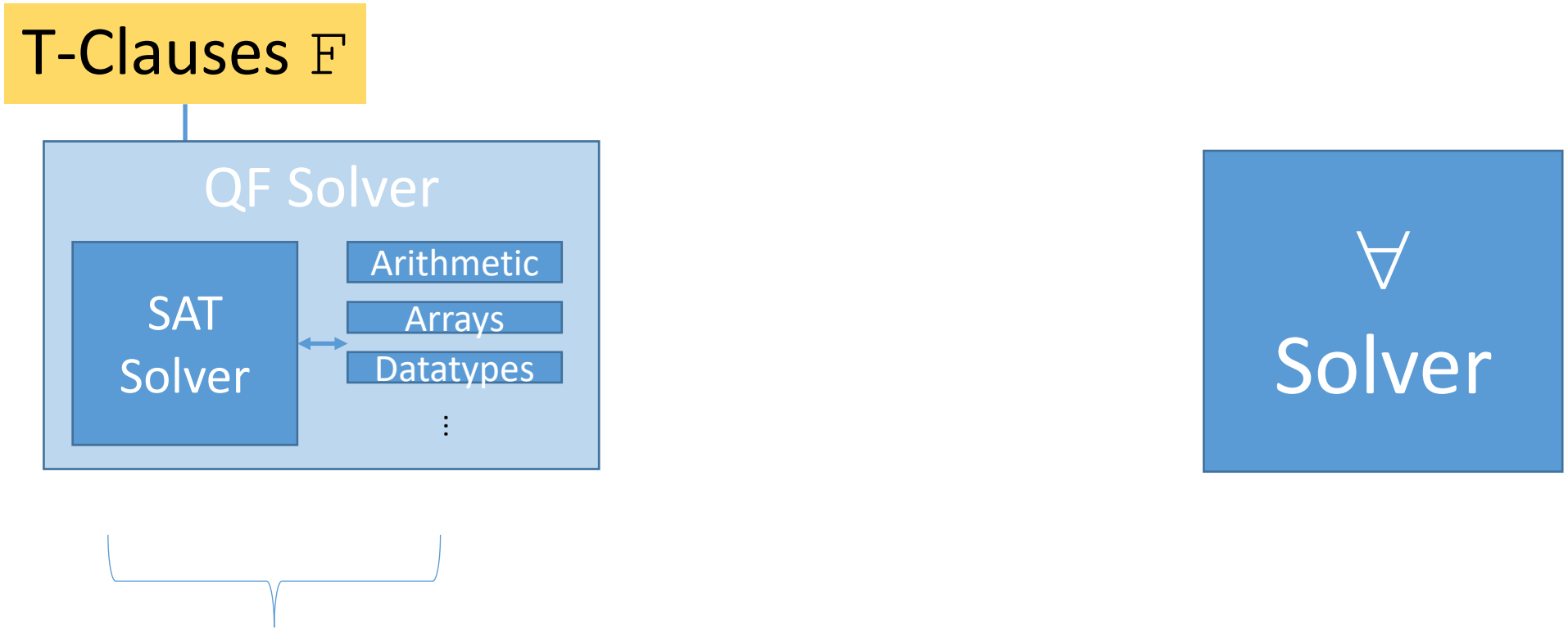


Satisfiability Modulo Theories (SMT) Solvers



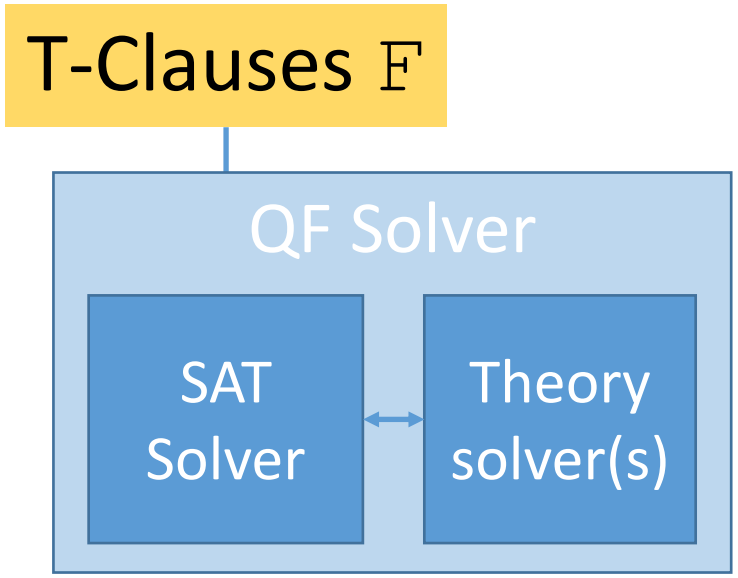
- Cooperative interaction between components

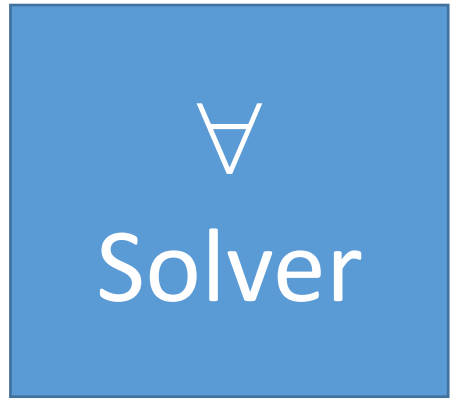
DPLL(T)-Based SMT Solvers + ∀ Instantiation



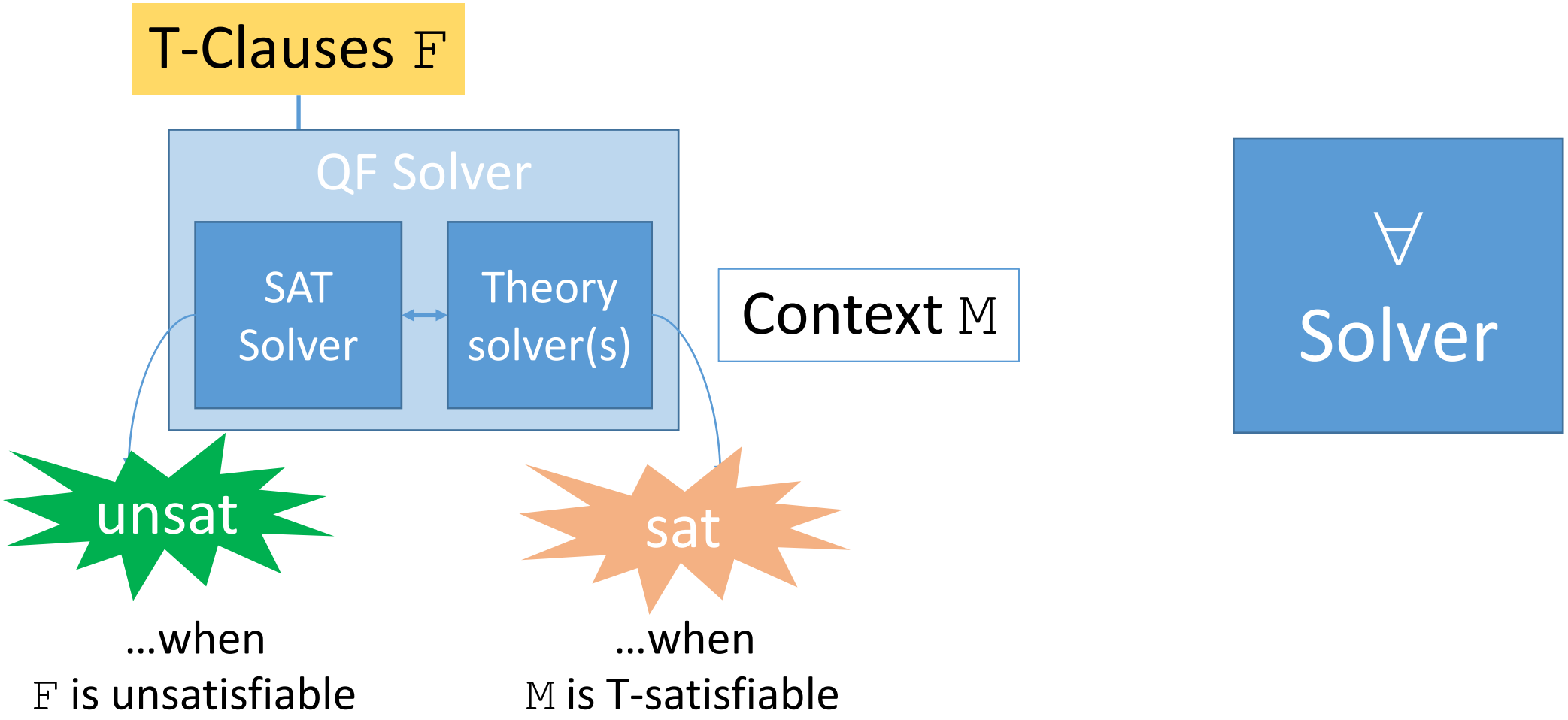
- Portion of SMT solver that focuses on quantifier-free reasoning (treats quantified formulas as propositional variables)

DPLL(T)-Based SMT Solvers + ∀ Instantiation



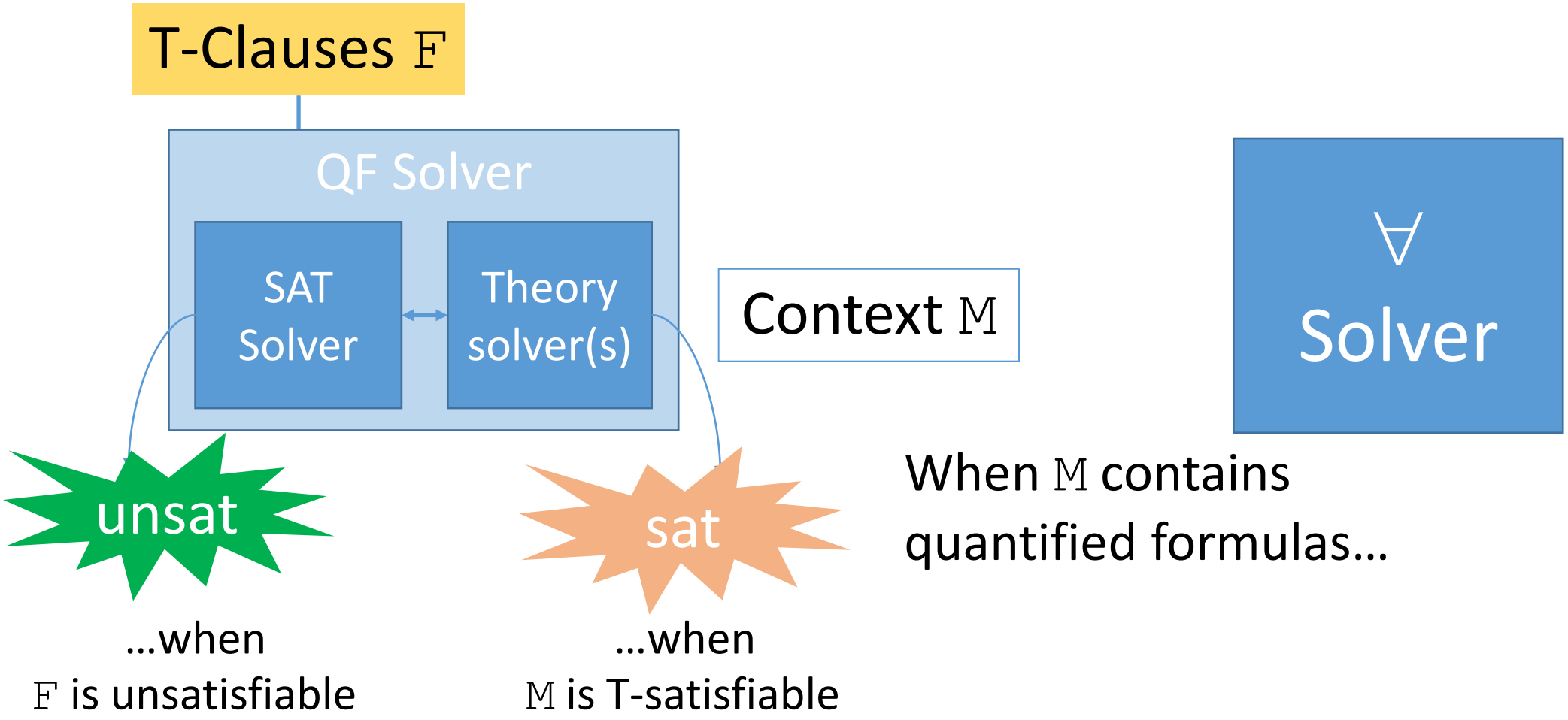


DPLL(T)-Based SMT Solvers + ∀ Instantiation

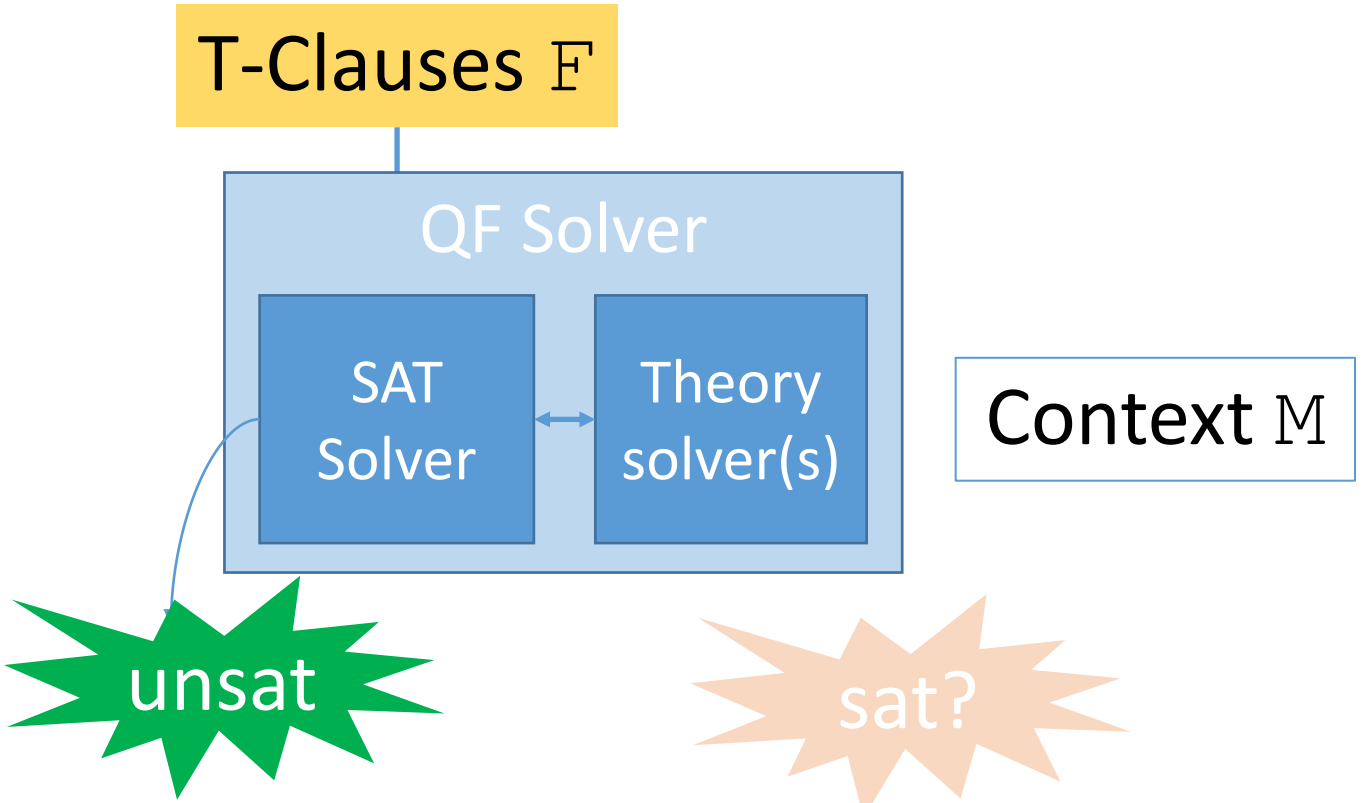


∀ Solver

DPLL(T)-Based SMT Solvers + ∀ Instantiation



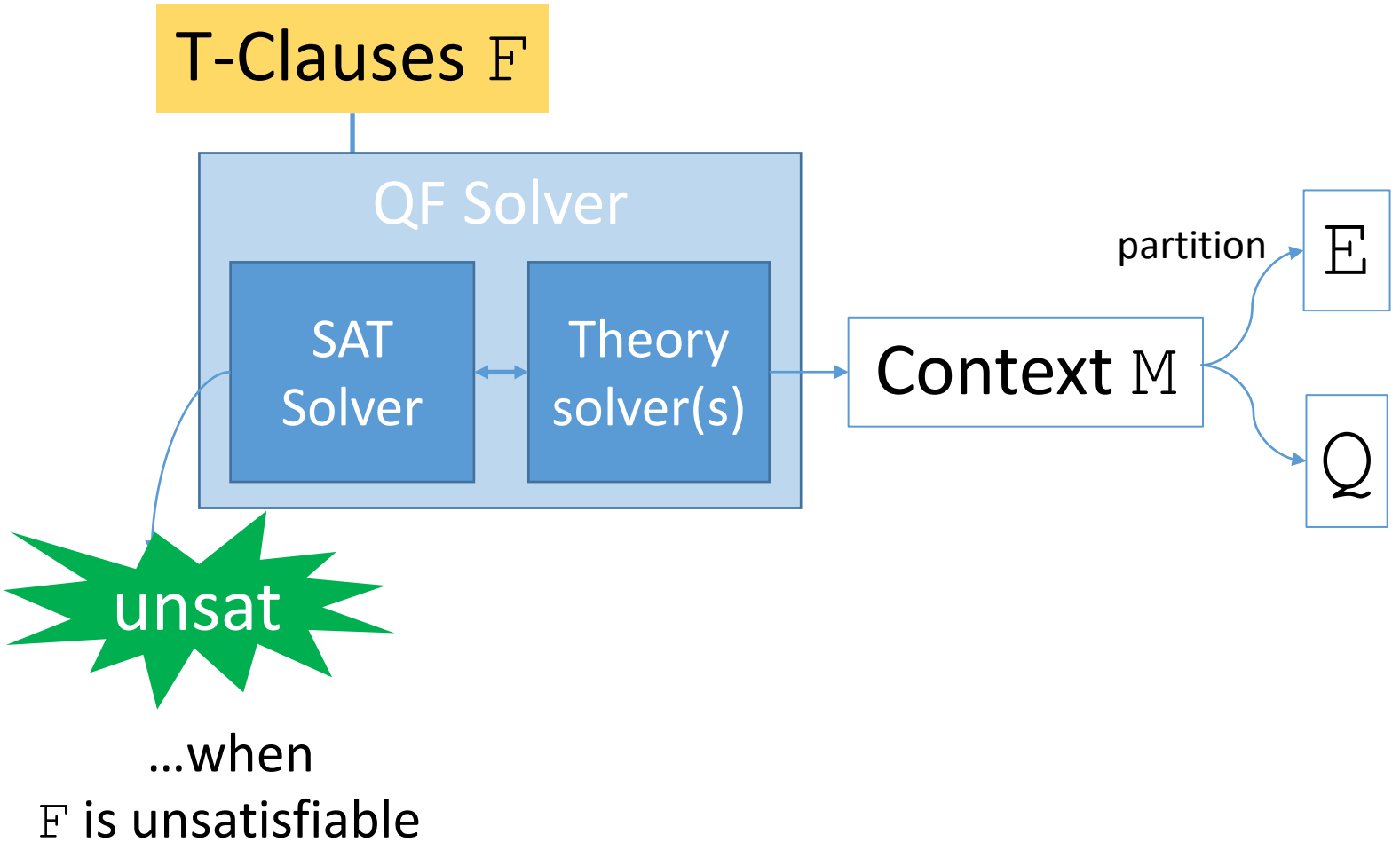
DPLL(T)-Based SMT Solvers + ∀ Instantiation



∀ Solver

…must consider quantified formulas in $\mathbb{M}$

DPLL(T)-Based SMT Solvers + ∀ Instantiation



…when
F is unsatisfiable

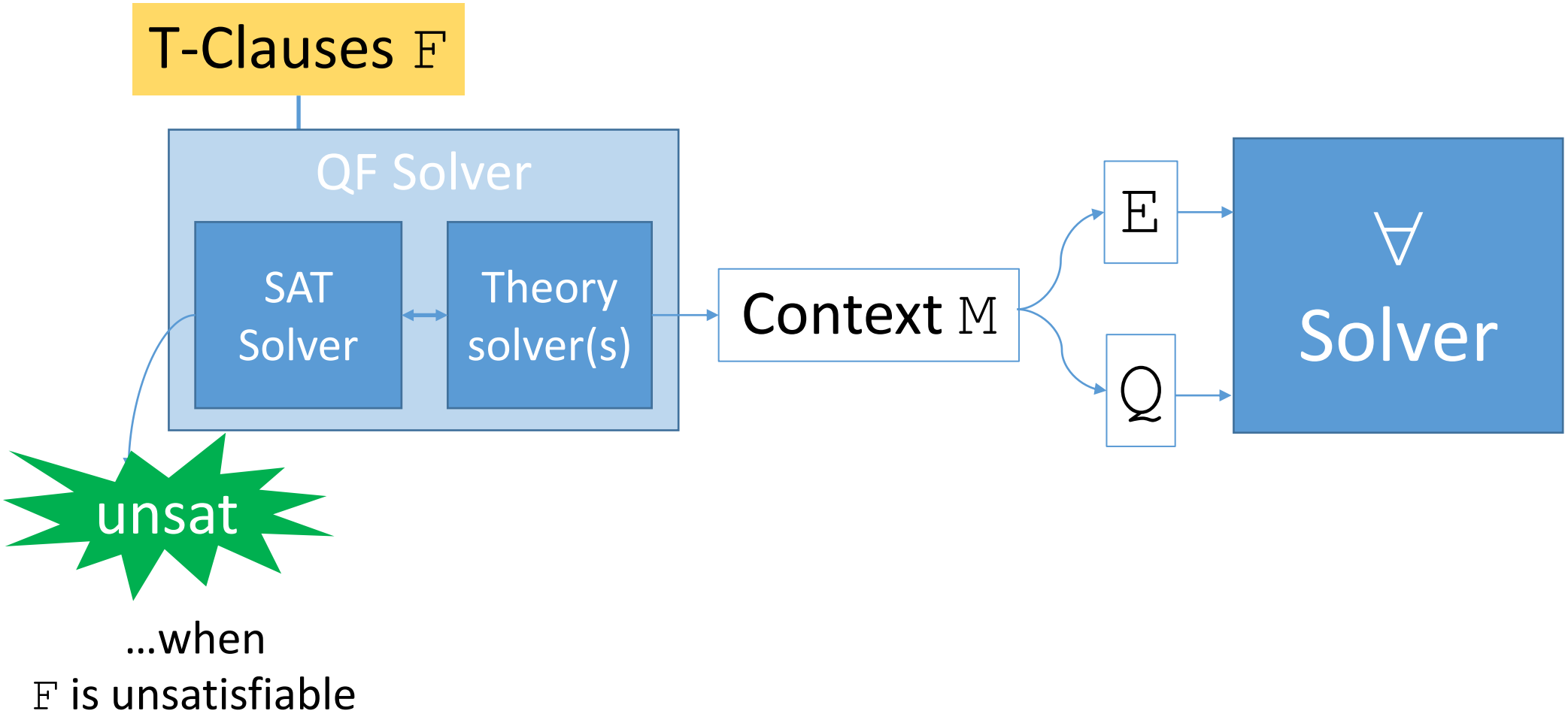
Set of ground equalities and disequalities

- {f(a)=b, P(a)=⊥, …}

Set of quantified formulas

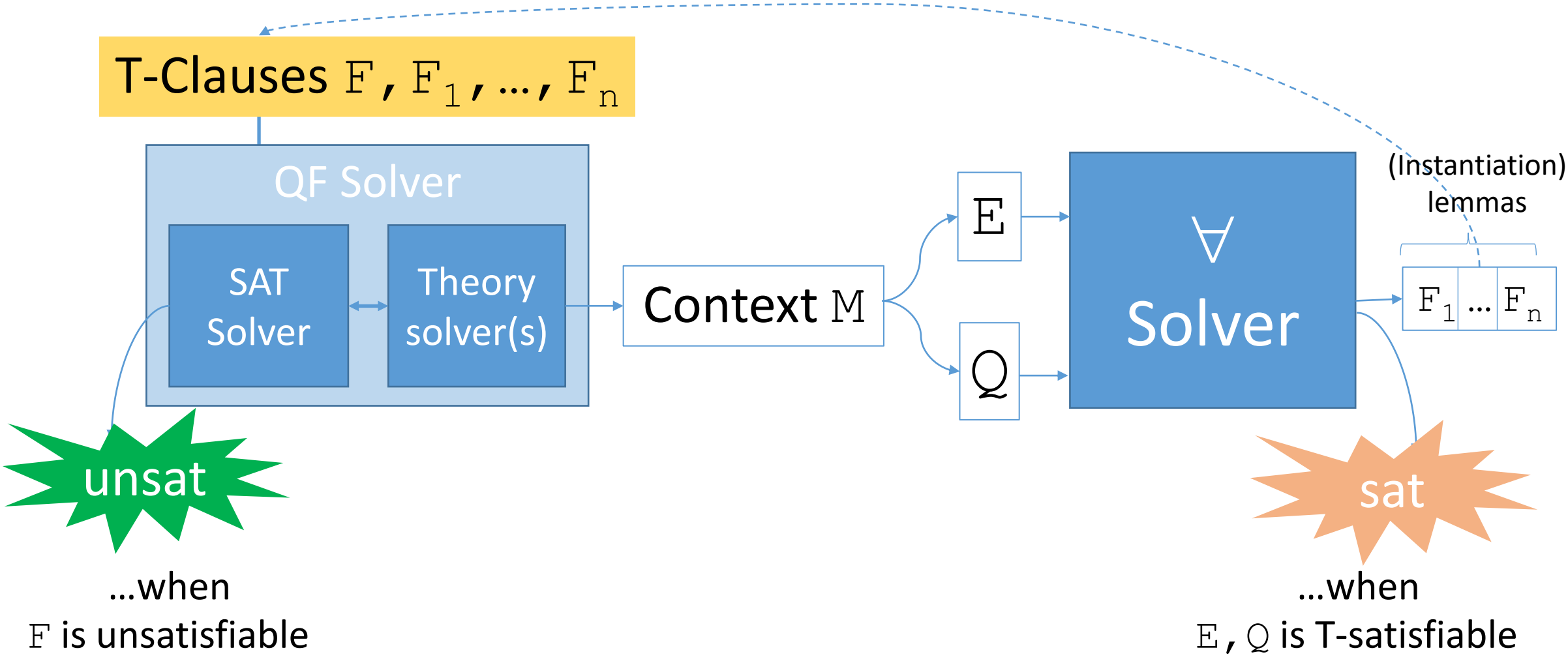
- {∀x.P(x), …}

DPLL(T)-Based SMT Solvers + ∀ Instantiation



…when
$\mathbb{F}$ is unsatisfiable

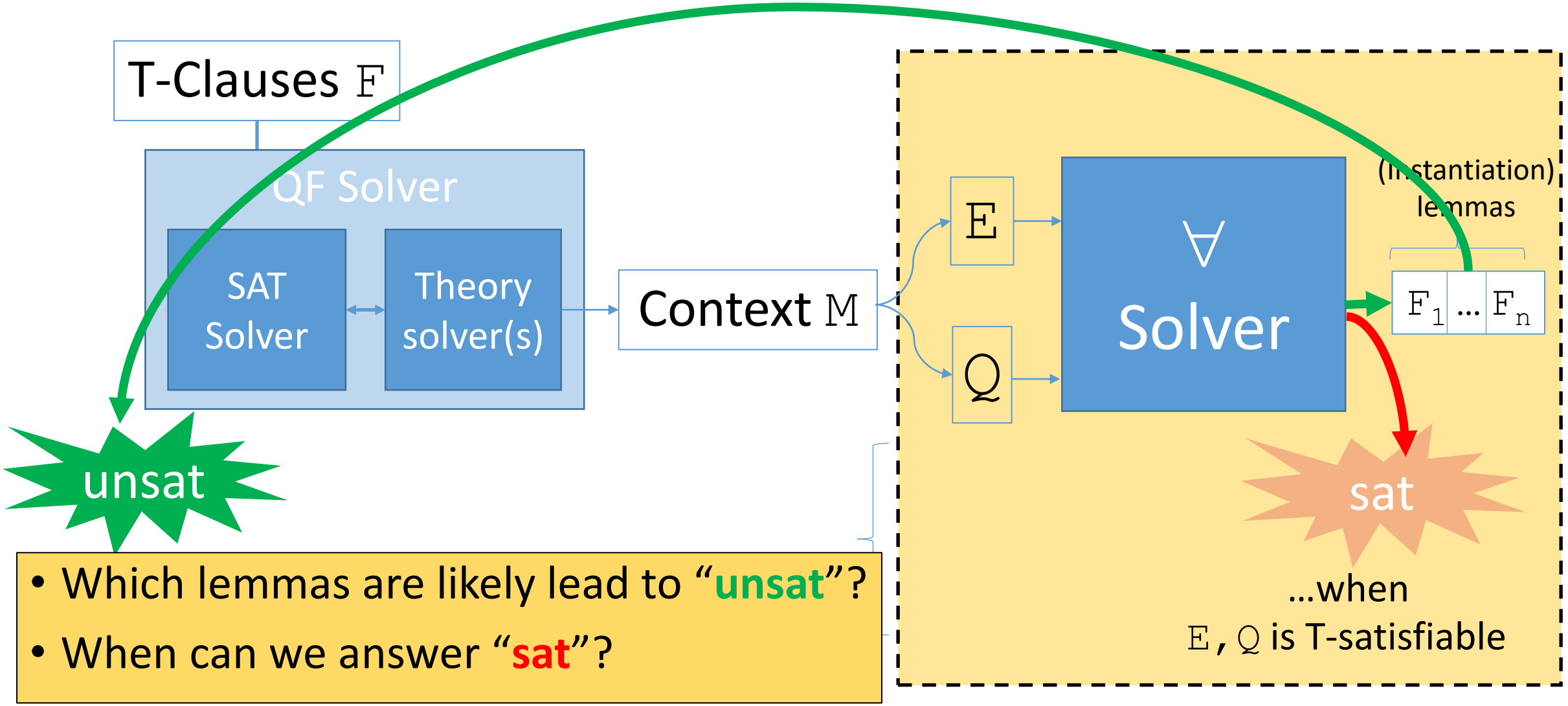
DPLL(T)-Based SMT Solvers + ∀ Instantiation



…when
F is unsatisfiable

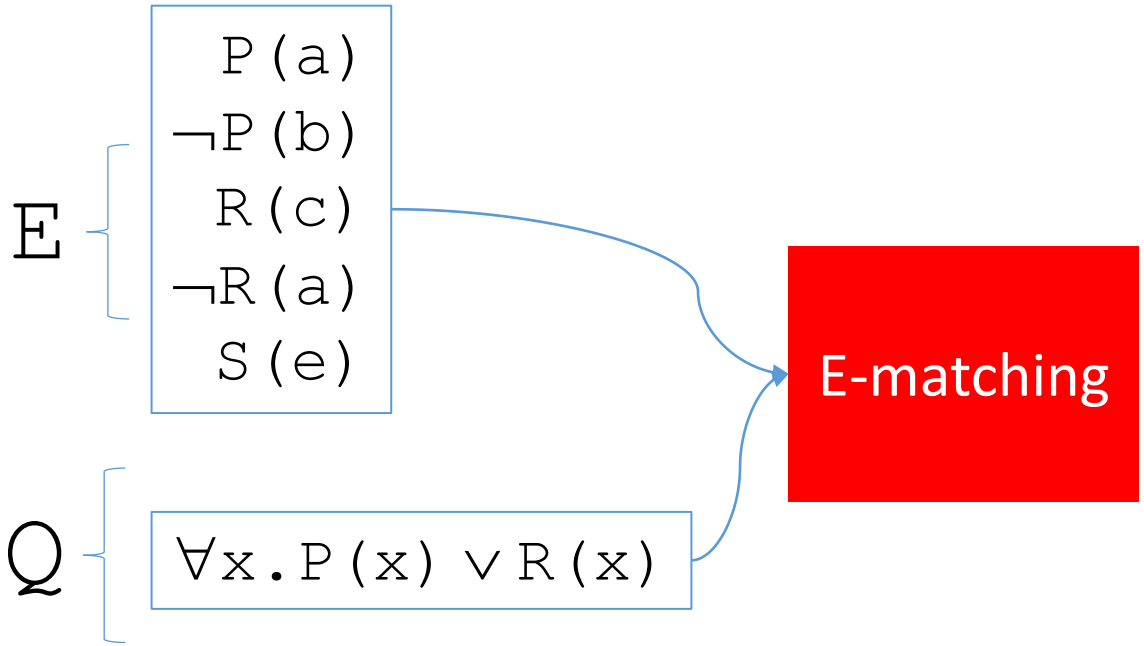
…when
E, Q is T-satisfiable

DPLL(T)-Based SMT Solvers + ∀ Instantiation



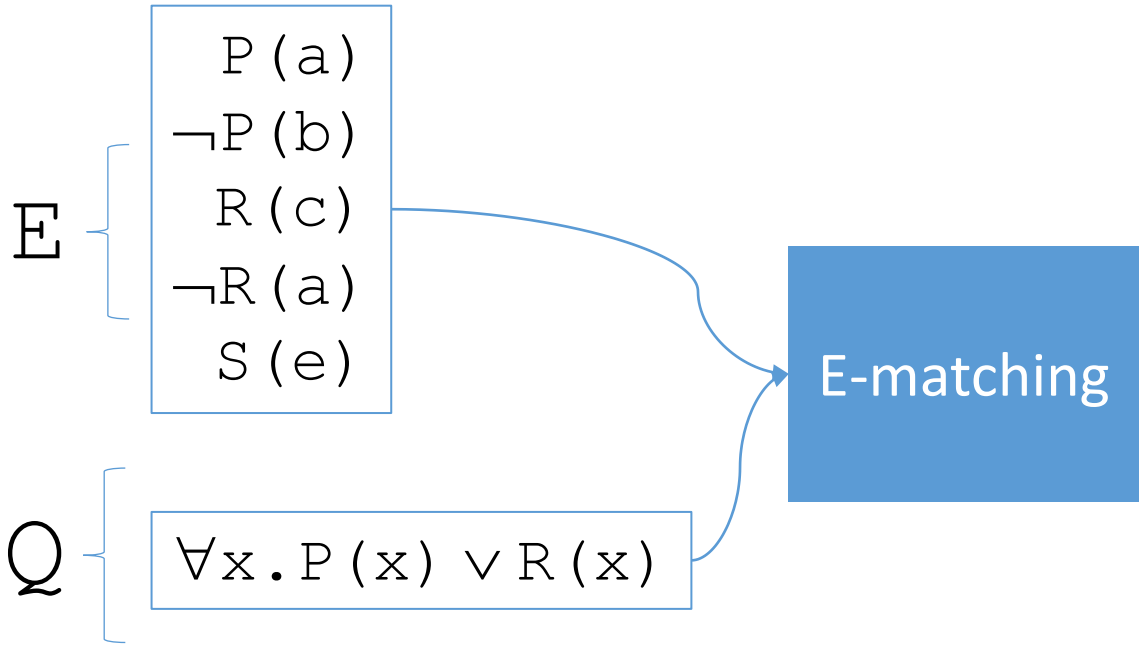
- Which lemmas are likely lead to "**unsat**"?
- When can we answer "**sat**"?

E-matching

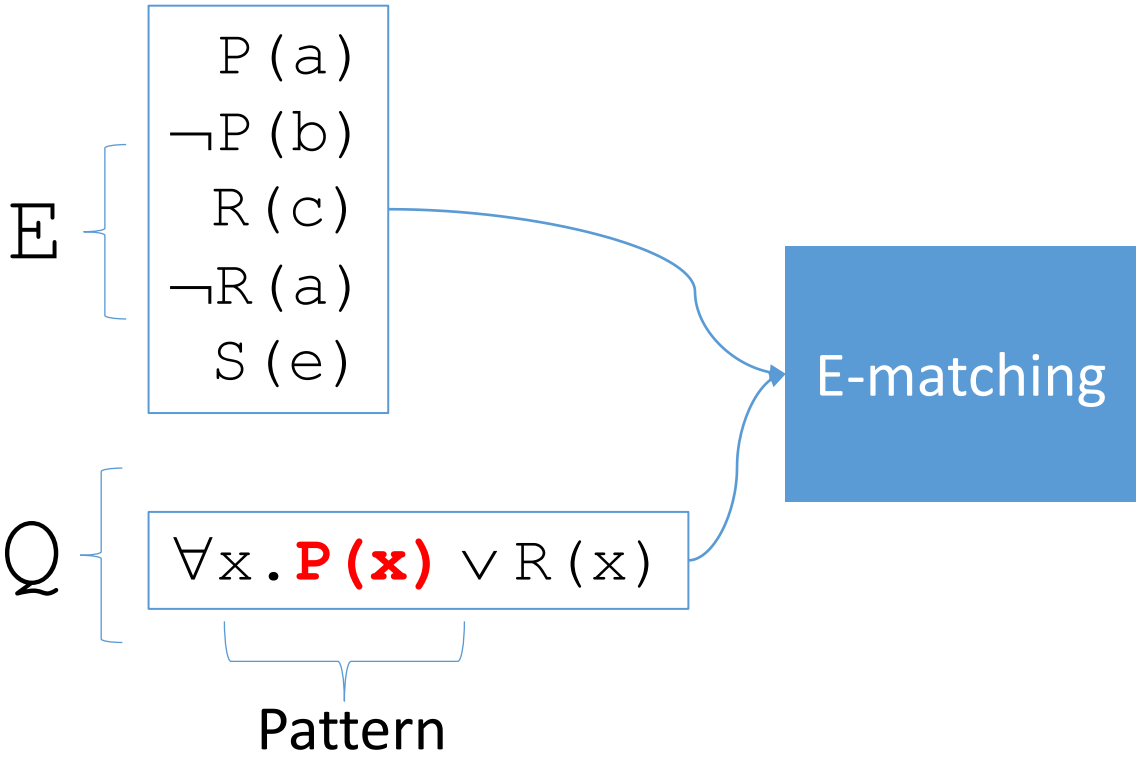


- Introduced in Nelson's Phd Thesis [Nelson 80]

E-matching

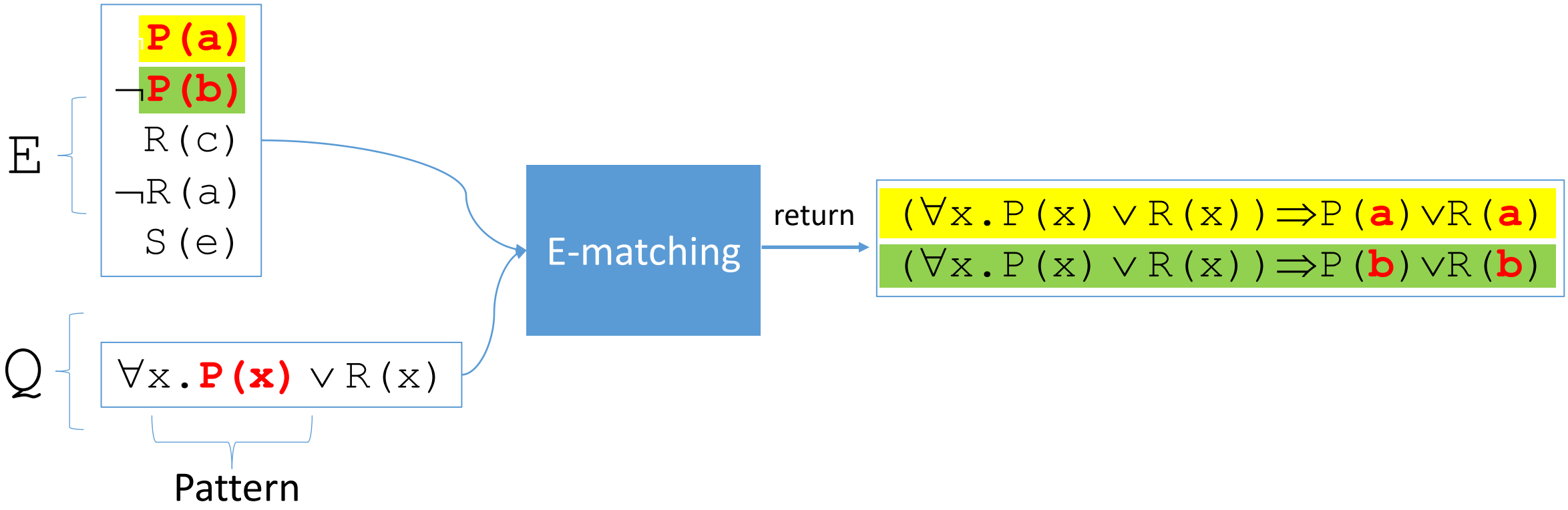


E-matching

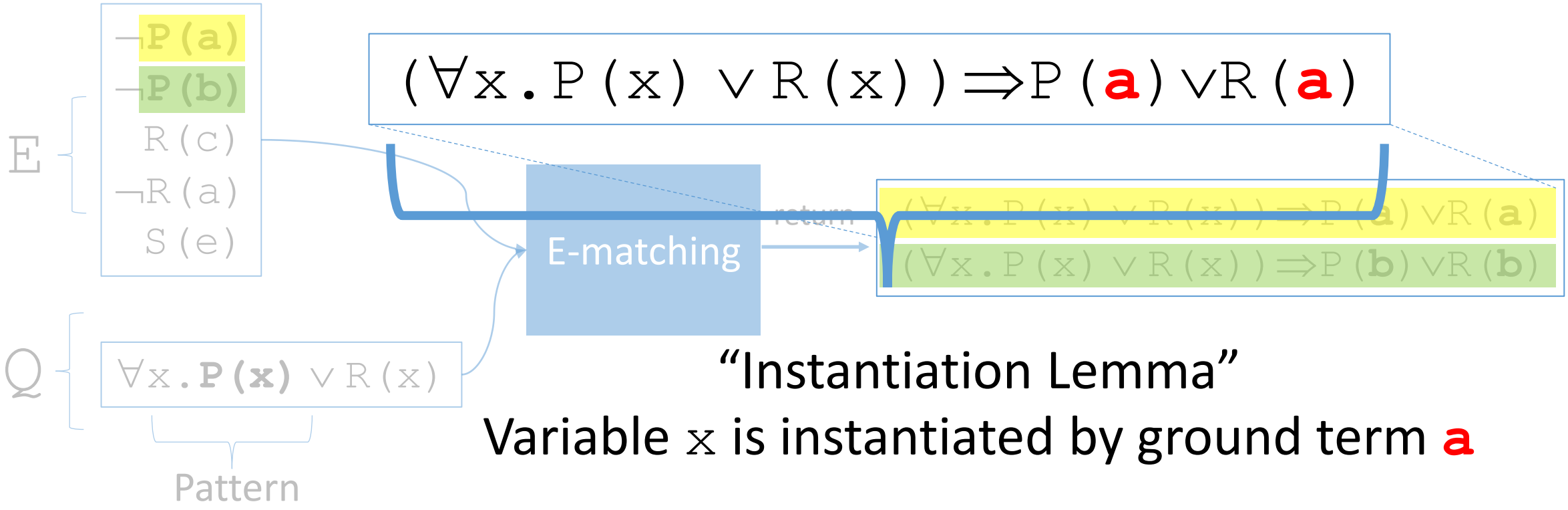


⇒ **Idea**: choose instances based on pattern matching

E-matching


E
P(a)
¬P(b)
R(c)
¬R(a)
S(e)
Q
∀x.P(x) ∨ R(x)
Pattern
E-matching
return
(∀x.P(x) ∨ R(x)) ⇒ P(a) ∨ R(a)
(∀x.P(x) ∨ R(x)) ⇒ P(b) ∨ R(b)

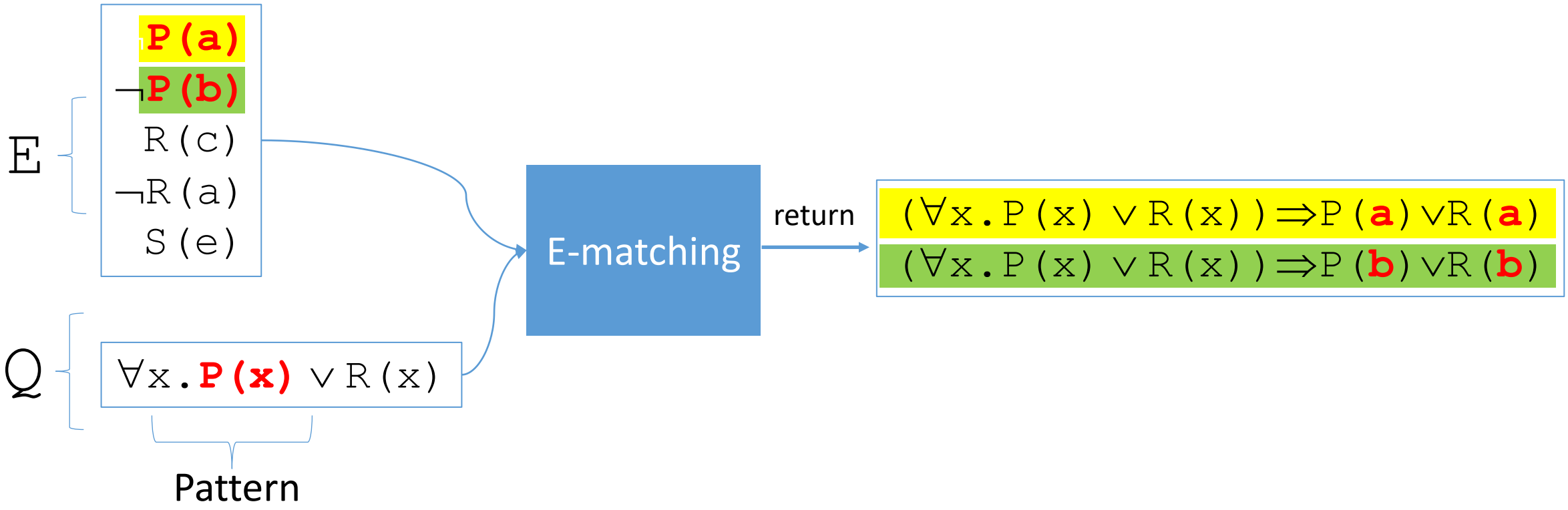
E-matching



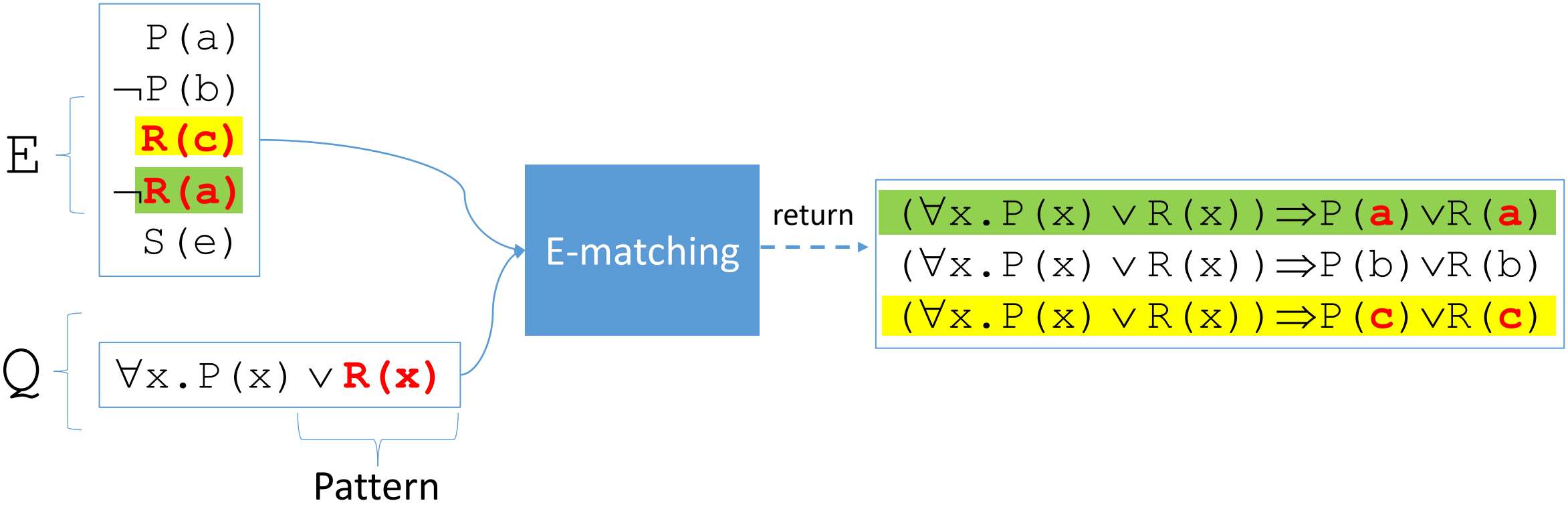
"Instantiation Lemma"

Variable x is instantiated by ground term **a**

E-matching


E
P(a)
¬P(b)
R(c)
¬R(a)
S(e)
Q
∀x.P(x) ∨ R(x)
Pattern
E-matching
return
(∀x.P(x) ∨ R(x)) ⇒ P(a) ∨ R(a)
(∀x.P(x) ∨ R(x)) ⇒ P(b) ∨ R(b)

E-matching


E
P(a)
¬P(b)
R(c)
¬R(a)
S(e)
Q
∀x.P(x) ∨ R(x)
Pattern
E-matching
return
(∀x.P(x) ∨ R(x)) ⇒ P(a) ∨ R(a)
(∀x.P(x) ∨ R(x)) ⇒ P(b) ∨ R(b)
(∀x.P(x) ∨ R(x)) ⇒ P(c) ∨ R(c)

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7))$

- DPLL(UFLIA) + E-Matching

Context

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true

Context

$\forall x.P(x)$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Decide : $P(2) \rightarrow$ false

Context
$\forall x.P(x)$ $\neg P(2)^{d}$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Decide : $P(2) \rightarrow$ false
  - Invoke UF solver for {$\neg P(2)$ }…UF-satisfiable

Context
$\forall x.P(x)$
$\neg P(2)^{d}$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Decide : $P(2) \rightarrow$ false
  - Invoke E-matching for E = { $\neg P(2)$ }, Q = { $\forall x.P(x)$ }

Context

$\forall x.P(x)$

$\neg P(2)^{d}$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Decide : $P(2) \rightarrow$ false
  - Invoke E-matching for E = { $\neg P(2)$ }, Q = { $\forall x.P(x)$ }

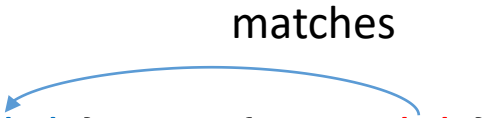
Pattern

Context
$\forall x.P(x)$
$\neg P(2)^d$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Decide : $P(2) \rightarrow$ false
  - Invoke E-matching for E = { $\neg P(2)$ }, Q = { $\forall x.P(x)$ }



Context

$\forall x.P(x)$

$\neg P(2)^d$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Decide : $P(2) \rightarrow$ false
  - Invoke E-matching for E = { $\neg P(2)$ }, Q = { $\forall x.P(x)$ }

    matches

    $\Rightarrow$ Return instantiation lemma ($\forall x.P(x) \Rightarrow P(2)$)

Context
$\forall x.P(x)$
$\neg P(2)^d$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - …Backtrack

Context

$\forall x.P(x)$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Propagate : $P(2) \rightarrow$ true

Context

$\forall x.P(x)$
$P(2)$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Propagate : $P(2) \rightarrow$ true
  - Propagate : $P(7) \rightarrow$ false

Context

$\forall x.P(x)$
$P(2)$
$\neg P(7)$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Propagate : $P(2) \rightarrow$ true
  - Propagate : $P(7) \rightarrow$ false
  - Invoke E-matching for E = { $P(2)$, $\neg P(7)$ }, Q = { $\forall x.P(x)$ }

Context
$\forall x.P(x)$
$P(2)$
$\neg P(7)$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Propagate : $P(2) \rightarrow$ true
  - Propagate : $P(7) \rightarrow$ false
  - Invoke E-matching for E = { $P(2)$, $\neg P(7)$ }, Q = { $\forall x.P(x)$ }

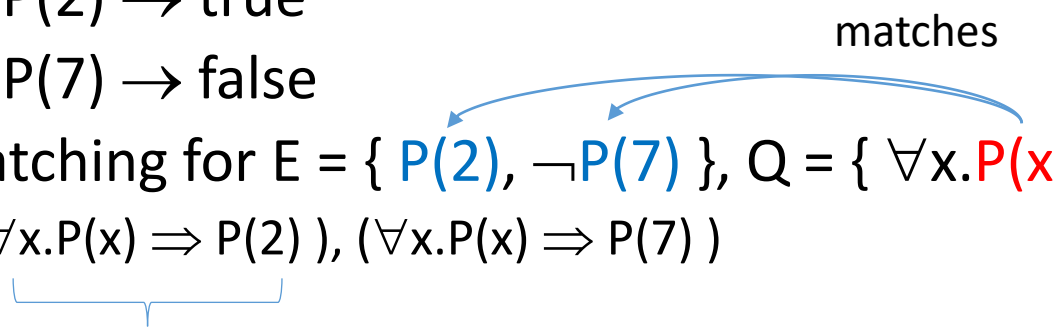
Context
$\forall x.P(x)$
$P(2)$
$\neg P(7)$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2)) \wedge (\neg \forall x.P(x) \vee P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Propagate : $P(2) \rightarrow$ true
  - Propagate : $P(7) \rightarrow$ false
  - Invoke E-matching for E = { $P(2)$, $\neg P(7)$ }, Q = { $\forall x.P(x)$ }

    $\Rightarrow$ Return ($\forall x.P(x) \Rightarrow P(2)$), ($\forall x.P(x) \Rightarrow P(7)$)



Context
$\forall x.P(x)$
$P(2)$
$\neg P(7)$

Example

$\forall x.P(x) \wedge (\neg P(2) \vee \neg P(7)) \wedge$
$(\neg \forall x.P(x) \vee P(2)) \wedge (\neg \forall x.P(x) \vee P(7))$

- DPLL(UFLIA) + E-Matching
  - Propagate : $\forall x.P(x) \rightarrow$ true
  - Propagate : $P(2) \rightarrow$ true
  - Propagate : $P(7) \rightarrow$ false
  - Invoke E-matching for E = { $P(2)$, $\neg P(7)$ }, Q = { $\forall x.P(x)$ }

    $\Rightarrow$ Return ($\forall x.P(x) \Rightarrow P(2)$), ($\forall x.P(x) \Rightarrow P(7)$)

$\Rightarrow$ Conflicting clause!

…no decision to backtrack

$\Rightarrow$ Input is UFLIA-unsat

Context
$\forall x.P(x)$
$P(2)$
$\neg P(7)$

Encoding in *.smt2

```
(set-logic UFLIA)
(declare-fun P (Int) Bool)
(assert (forall ((x Int)) (P x)))
(assert (or (not (P 2)) (not (P 7))))
(check-sat)
```

SMT Solvers for ∀ **using Quantifier Instantiation**

- Traditionally:
  - E-matching **[Detlefs et al 2005,Bjorner et al 2007, Ge et al 2007]**

- More recently:
  - Model-Based Instantiation **[Ge et al 2009, Reynolds et al 2013]**
  - Conflict-Based Instantiation **[Reynolds et al 2014, Barbosa et al 2017]**
  - Theory-specific Approaches
    - Linear arithmetic **[Bjorner 2012, Reynolds et al 2015, Janota et al 2015]**
    - Bit-Vectors **[Wintersteiger et al 2013, Dutertre 2015, Niemetz et al 2018]**

Implemented in
simplify, cvc3, z3, FX7, Alt-Ergo, Princess, cvc4, veriT
z3, cvc4
cvc4, veriT
z3, cvc4, yices, veriT+redlog

Summary

- SMT solvers: efficient tools
  - DPLL, DPLL(T), decision procedures, Nelson-Oppen theory combination
- Many applications
  - Verification, interactive theorem proving, synthesis
- Many ongoing research areas in SMT
  - More features: new theories, optimization, interpolants, abducts, proofs
  - Improvements to existing techniques

- …Thanks for listening!

- Techniques from these lectures available in CVC4:
  - Open source
  - Available at : https://cvc4.github.io/
  - Accepts *.smt2, *.sy formats