Satisfiability Modulo Theories : Beyond Decision Procedures

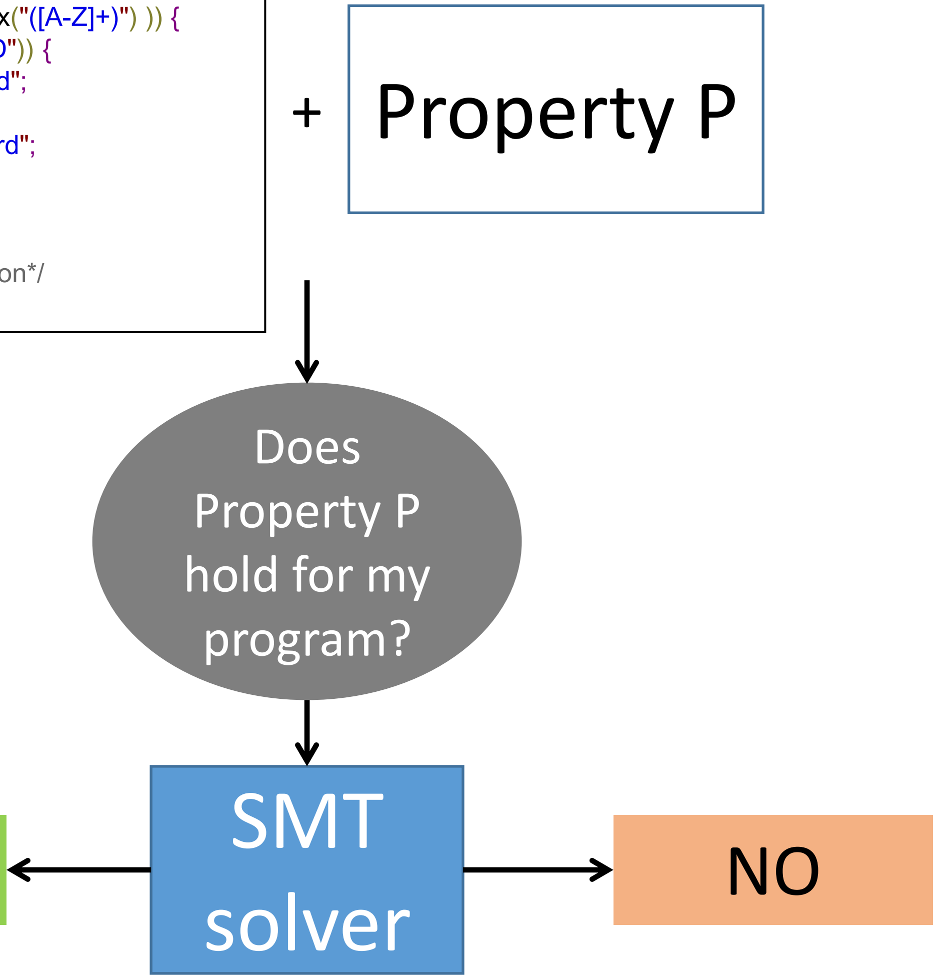
Andrew Reynolds

May 20, 2015

SMT Solvers for Software Verification/Security

Program

```
char buff[15], 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 the root permission*/
}
```



SMT Solvers for Software Verification/Security

Program

```
char buff[15], 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 the root permission*/
}
```

+ Property P

Does Property P hold for my program?

SMT solver

YES

NO

What kind of properties can we ask about?

How do we solve them?

Overview

- Satisfiability Modulo Theories (SMT) Solvers
  - Propositional reasoning, via off-the-shelf SAT solver
  - Decision Procedures for *theories*:
    - UF, Arithmetic, BitVectors, Arrays, …
    - (Co)inductive Datatypes
  - …also support Undecidable Theories:
    - Unbounded Strings + Length Constraints
  - …and even arbitrary Quantified Formulas:
    - Finite Model Finding

Overview

- Satisfiability Modulo Theories (SMT) Solvers
  - Propositional reasoning, via off-the-shelf SAT solver
  - Decision Procedures for *theories*:
    - UF, Arithmetic, BitVectors, Arrays, …
    - (Co)inductive Datatypes
  - …also support Undecidable Theories:
    - Unbounded Strings + Length Constraints
  - …and even arbitrary Quantified Formulas:
    - Finite Model Finding

Focus of this talk,
my work in **CVC4**

What is a Theory?

- A *theory* T is a pair
  - A signature Σ_T containing sorts and function symbols
  - A class of models I_T describing the intended interpretations of symbols in Σ_T
- For example, linear integer arithmetic (LIA):
  - Σ_{LIA} contains functions { +, -, <, ≤, >, ≥, 0, 1, 2, 3, … }
  - Each I $\in I_{LIA}$ interpret functions in Σ_{LIA} in standard way:
    - 1+1 = 2, 1+2 = 3, …, 1 > 0 = true, 0 > 1 = false, …
- Number of widely-supported theories in SMT:
  - Bitvectors : `bvsgt(a,#bin0001)`
  - Arrays : `select(store(a,5,b),c)=5`
  - Datatypes : `tail(cons(a,b))=b`
  - …

What is a Decision Procedure for T?

- Input: a set of T-constraints $\mathbb{M}$, under some syntactic restriction
- A decision procedure is a method that terminates with output:
  - "$\mathbb{M}$ is T-satisfiable", i.e. there is a solution
  - "$\mathbb{M}$ is T-unsatisfiable"

What is a Decision Procedure for T?

- Input: a set of T-constraints $\mathbb{M}$, under some syntactic restriction
- A decision procedure is a method that terminates with output:
  - “$\mathbb{M}$ is T-satisfiable”, i.e. there is a solution
  - “$\mathbb{M}$ is T-unsatisfiable”

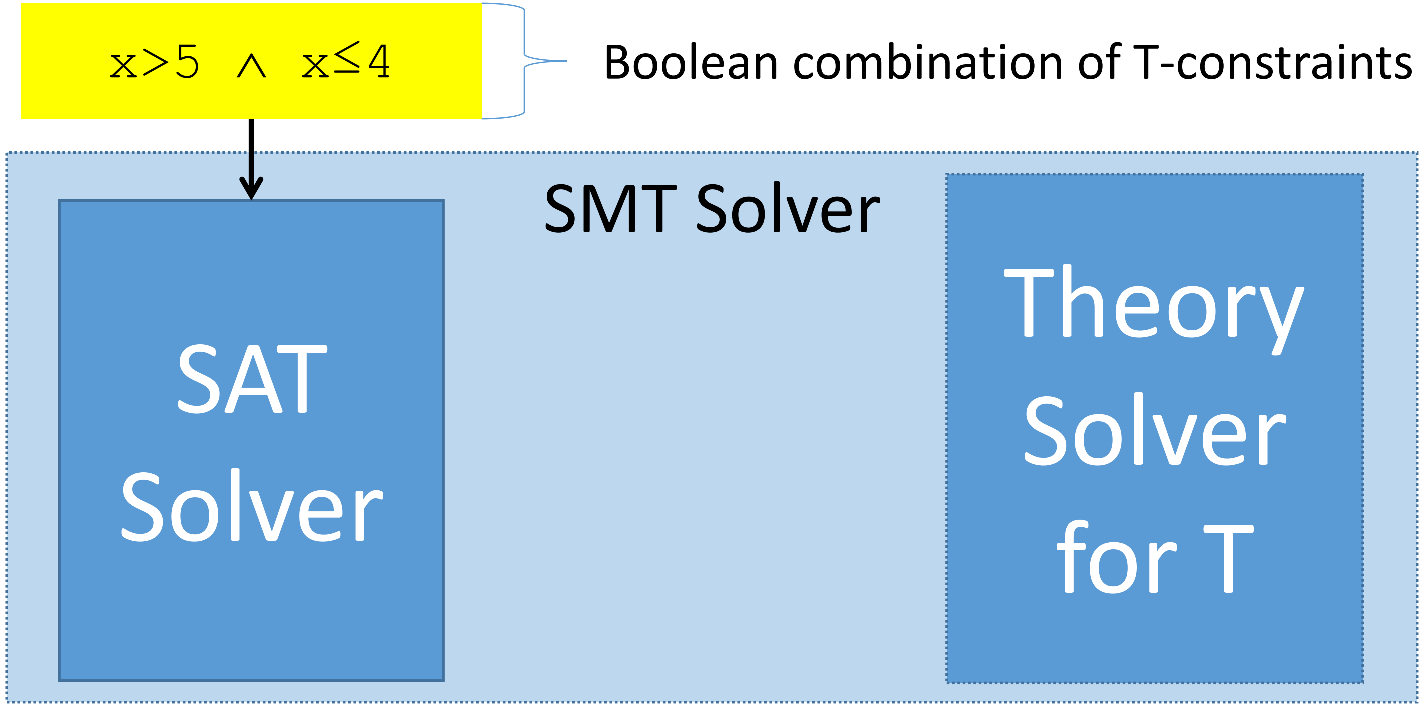
What is a Decision Procedure for T?

- Input: a set of T-constraints $\mathbb{M}$, under some syntactic restriction
- A decision procedure is a method that terminates with output:
  - "$\mathbb{M}$ is T-satisfiable", i.e. there is a solution
    - Must be solution-sound, returns "$\mathbb{M}$ is T-satisfiable" only when $\mathbb{M}$ is T-satisfiable
  - "$\mathbb{M}$ is T-unsatisfiable"

What is a Decision Procedure for T?

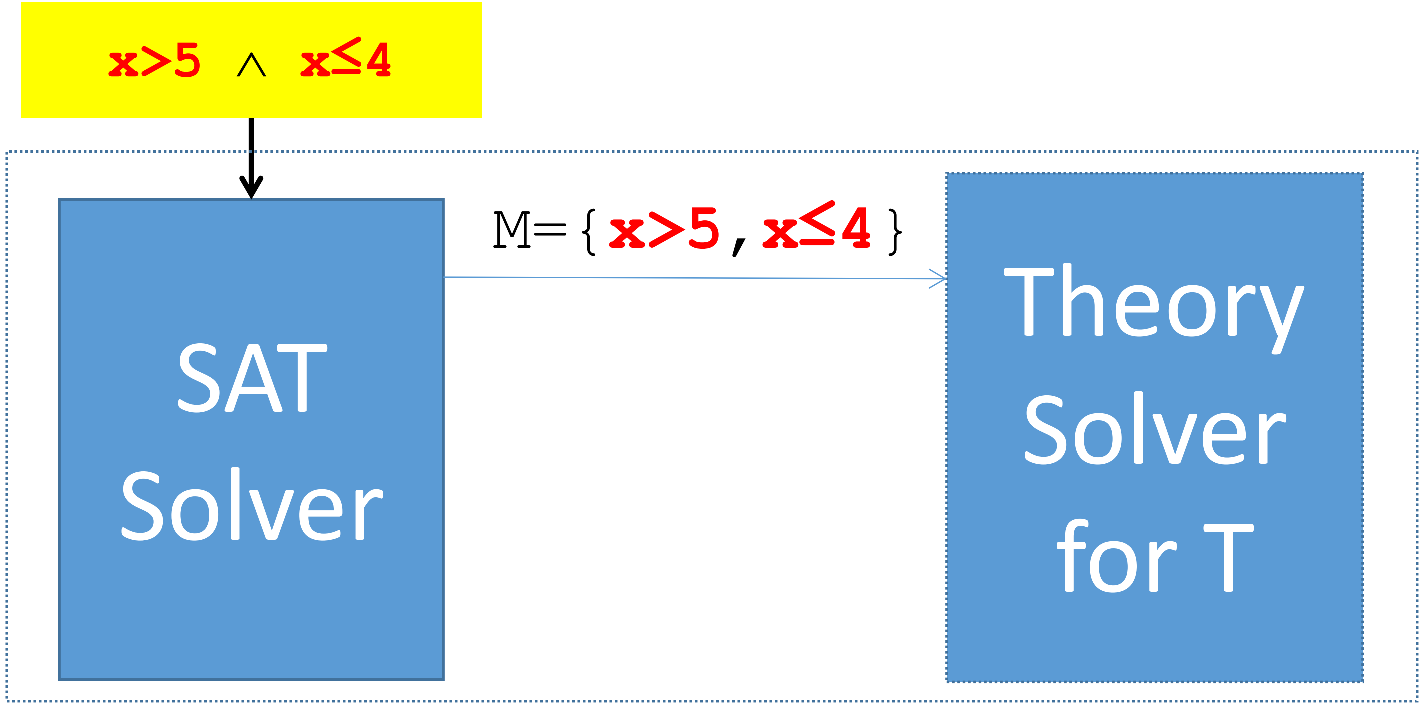
- Input: a set of T-constraints $\mathbb{M}$, under some syntactic restriction
- A decision procedure is a method that terminates with output:
  - "$\mathbb{M}$ is T-satisfiable", i.e. there is a solution
    - Must be solution-sound, returns "$\mathbb{M}$ is T-satisfiable" only when $\mathbb{M}$ is T-satisfiable
  - "$\mathbb{M}$ is T-unsatisfiable"
    - Must be refutation-sound, returns "$\mathbb{M}$ is T-unsatisfiable" only when $\mathbb{M}$ is T-unsatisfiable

How are Decision Procedures Implemented in SMT?



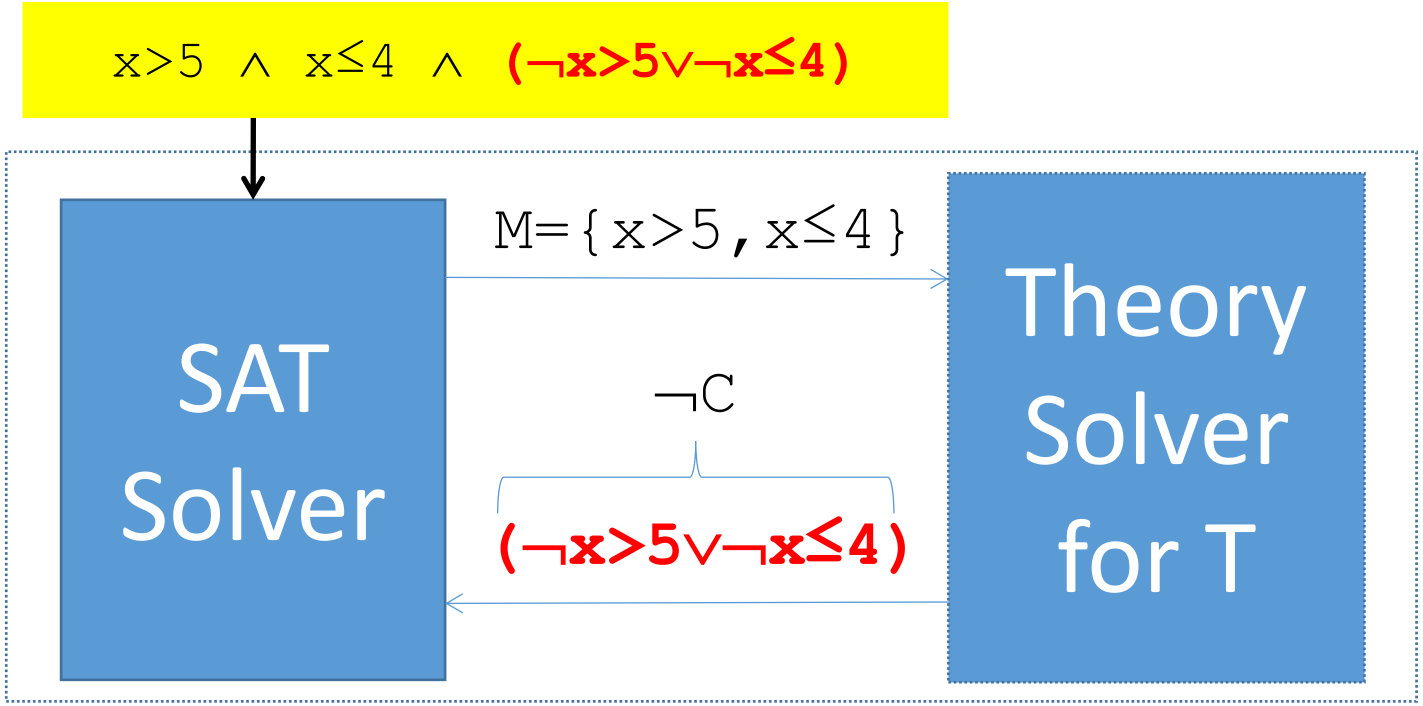
- Decision Procedures are implemented as *theory solvers*

How are Decision Procedures Implemented in SMT?



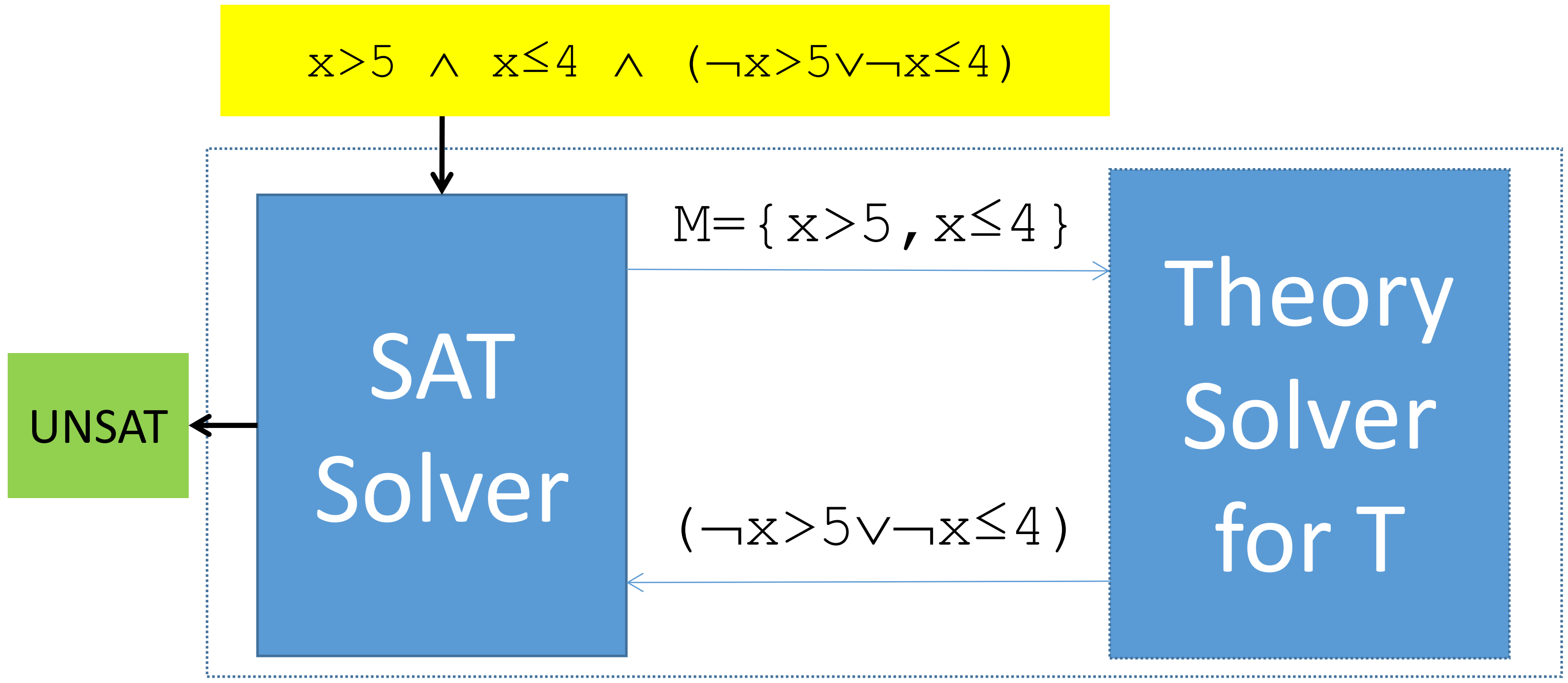
- Decision Procedures are implemented as *theory solvers*

How are Decision Procedures Implemented in SMT?



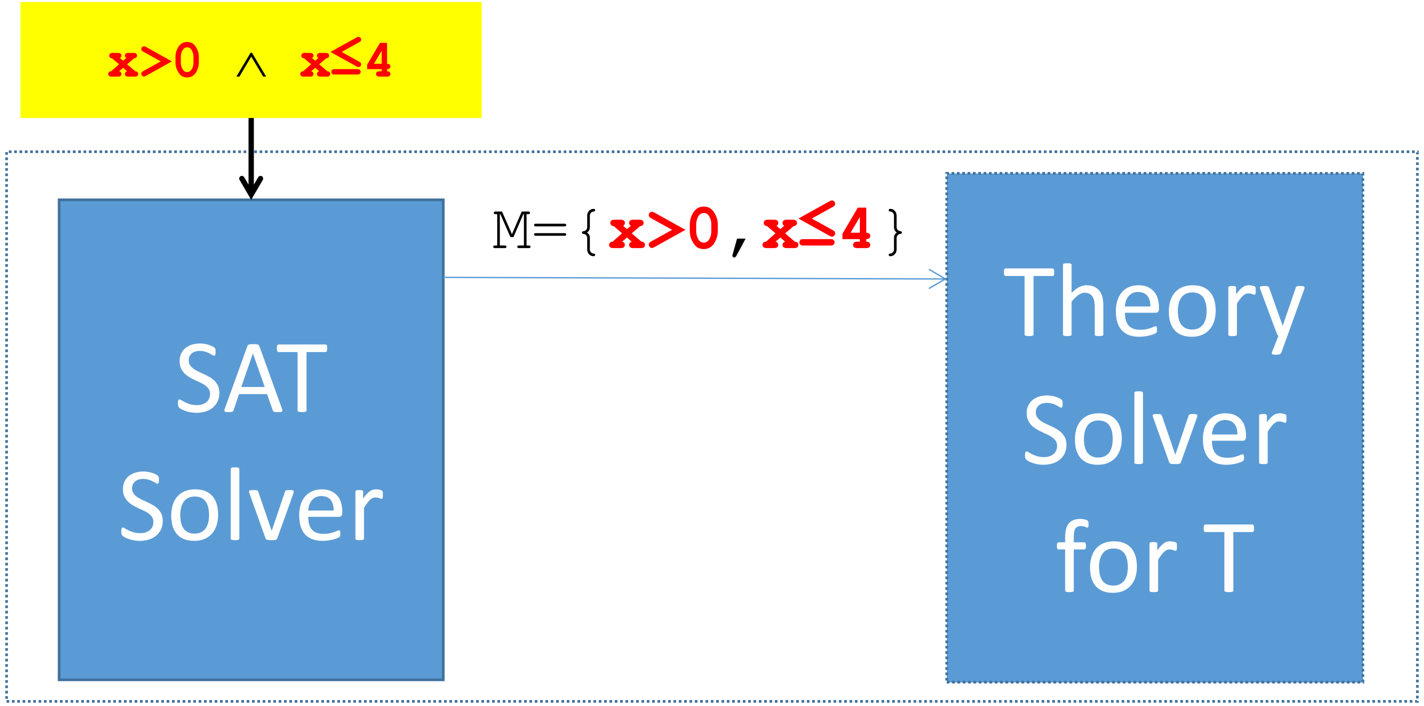
- Decision Procedures are implemented as *theory solvers*
- If M is T-unsat, find an inconsistent subset $C \subseteq M$, add conflict clause $\neg C$

How are Decision Procedures Implemented in SMT?



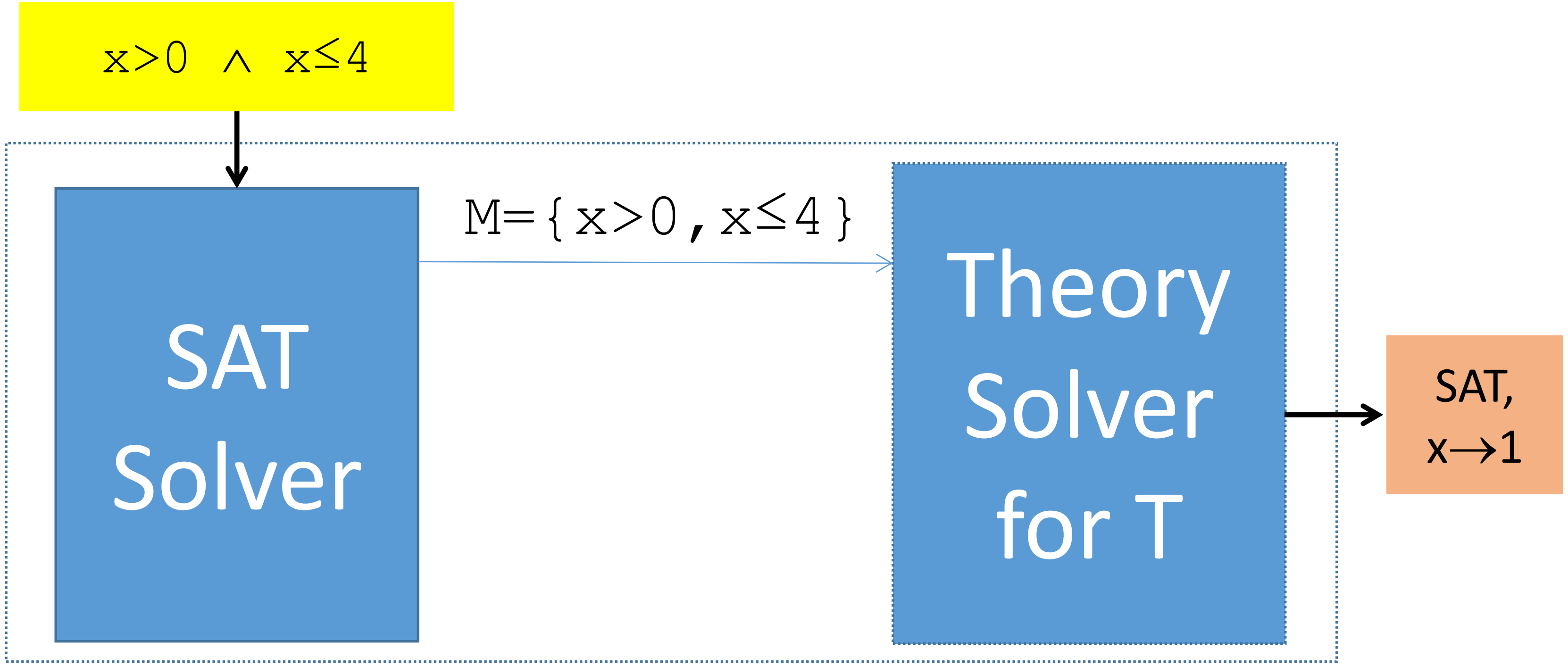
- Decision Procedures are implemented as *theory solvers*
- If M is T-unsat, find an inconsistent subset $C \subseteq M$, add conflict clause $\neg C$

How are Decision Procedures Implemented in SMT?



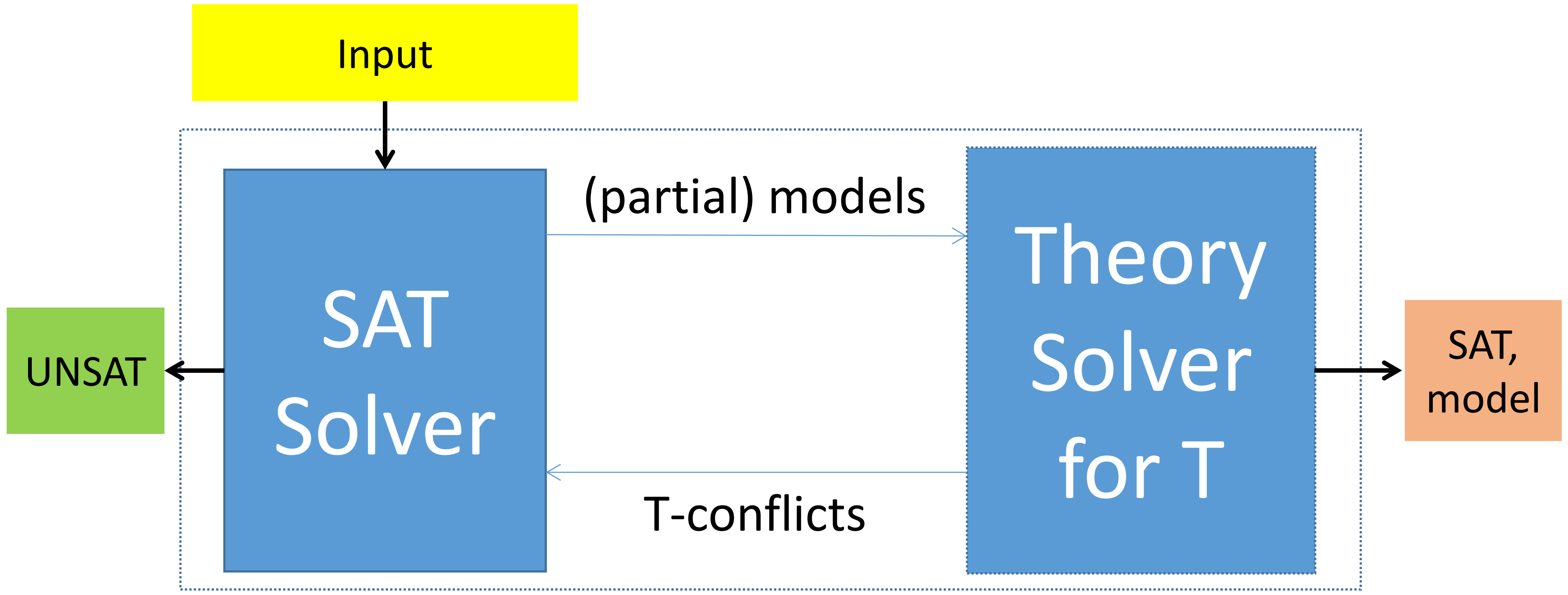
- Decision Procedures are implemented as *theory solvers*
- If M is T-unsat, find an inconsistent subset $C \subseteq M$, add conflict clause $\neg C$

How are Decision Procedures Implemented in SMT?



- Decision Procedures are implemented as *theory solvers*
- If M is T-unsat, find an inconsistent subset C ⊆ M, add conflict clause ¬C
- If M is T-sat, return an interpretation for variables in model of M

How are Decision Procedures Implemented in SMT?



$\Rightarrow$ DPLL(T) procedure [Nieuwenhuis/Oliveras/Tinelli 2007]

Design of Theory Solvers in SMT

- A DPLL(T) theory solver:
  - Should be solution-sound, refutation-sound, terminating for input M
  - Should produce models and T-conflicts
  - Should be designed to work *incrementally*
    - M is constantly being appended to/backtracked upon
  - Should compute useful T-propagations
  - Should cooperate with other theory solvers for combined theories
    - [Nelson/Oppen 1979]

Examples of Decision Procedures in SMT

- Efficient theory solvers have been developed for:
  - Theory of Equality and Uninterpreted Functions (EUF)
    - Congruence closure algorithm [Nieuwenhius/Oliveras 2007]
  - Theory of Linear Integer/Real Arithmetic
    - Simplex algorithm [Detertre/deMoura 2006]
  - Theory of Arrays [deMoura/Bjorner 2009]
  - Theory of Bit Vectors [Brummayer/Biere 2009]
  - Theory of Inductive Datatypes [Barrett et al 2007]

    $\Rightarrow$ *Theory of (Co)Inductive Datatypes* **[Reynolds/Blanchette 2015]**

Theory of (Co)Inductive Datatypes

Theory of Inductive Datatypes : Applications

- Leon verification tool developed at EPFL
  - Reasons about the correctness of simple functional programs written in Scala

- Makes heavy use of SMT solver backend with support for inductive datatypes

Theory of Inductive Datatypes

- Family of theories specified by a set of *types* with *constructors*, e.g:

$$\boxed{\text{List} := \text{cons}(\ \text{head} : \text{Int},\ \text{tail} : \text{List}\ )\ |\ \text{nil}}$$

- Theory of Inductive Datatypes (DT) for Lists of Int
  - Σ_{DT} : { cons, head, tail, nil }
  - Interpretations I_{DT} are such that:
    - Constructors are distinct… cons(x,y) $\neq$ nil
    - Constructors are injective… if cons(x_1, y_1) = cons(x_2, y_2), then $x_1 = x_2$, $y_1 = y_2$
    - Constructors are exhaustive… top symbol of all lists is either cons or nil
    - Selectors access subfields… head(cons(x, y)) = x
    - Terms do not contain themselves as subterms… y $\neq$ cons(x, y)

- My work: decision procedure for DT in CVC4, based on [Barrett et al 2007]

  $\Rightarrow$ *Used as a backend to Leon verification system*

What about infinite data structures?

- Consider the definition:

$$\boxed{\text{Stream} := \text{cons}(\ \text{head} : \text{Int},\ \text{tail} : \text{Stream}\ )}$$

- Stream is not well-founded

  $\Rightarrow$ *Decision procedure for inductive datatypes does not apply*

- Instead, need decision procedure for coinductive datatypes
- Applications :
  - Modeling infinite processes
  - Programming languages: CoCaml [Jeannin et al 2013], Dafny [Leino 2014]
  - Proof assistants : Agda, Coq, Isabelle, …

  $\Rightarrow$ These applications can benefit from native support for them in SMT solvers

Theory of (Co)Inductive Datatypes

- Devised a unified decision procedure for inductive/coinductive datatypes
  - Implemented in CVC4

$$\frac{t \in \mathcal{T}(E)}{E := E,\, t \approx t}\ \textsf{Refl} \qquad \frac{t \approx u \in E}{E := E,\, u \approx t}\ \textsf{Sym} \qquad \frac{s \approx t,\ t \approx u \in E}{E := E,\, s \approx u}\ \textsf{Trans}$$

$$\frac{\bar{t} \approx \bar{u} \in E \quad \mathsf{f}(\bar{t}), \mathsf{f}(\bar{u}) \in \mathcal{T}(E)}{E := E,\, \mathsf{f}(\bar{t}) \approx \mathsf{f}(\bar{u})}\ \textsf{Cong} \qquad \frac{t \approx u,\ t \not\approx u \in E}{\bot}\ \textsf{Conflict}$$

$$\frac{\mathsf{C}(\bar{t}) \approx \mathsf{C}(\bar{u}) \in E}{E := E,\, \bar{t} \approx \bar{u}}\ \textsf{Inject} \qquad \frac{\mathsf{C}(\bar{t}) \approx \mathsf{D}(\bar{u}) \in E \quad \mathsf{C} \neq \mathsf{D}}{\bot}\ \textsf{Clash}$$

$$\frac{\delta \in \mathcal{Y}_{\mathrm{dt}} \quad \mathcal{A}[t^{\delta}] = \mu x.\, u \quad x \in \mathrm{FV}(u)}{\bot}\ \textsf{Acyclic} \qquad \frac{\delta \in \mathcal{Y}_{\mathrm{codt}} \quad \mathcal{A}[t^{\delta}] =_{\alpha} \mathcal{A}[u^{\delta}]}{E := E,\, t \approx u}\ \textsf{Unique}$$

- For codatatypes:
  - Terms *can* contain themselves as subterms : x=cons(z,x) is satisfiable
  - Terms are unique up to α-equivalence:
    - If x=cons(z,x) and y=cons(z,y), then x=y

[Reynolds/Blanchette CADE15]

Theory of (Co)Inductive Datatypes

	Distro		AFP		G&L		Overall	
	CVC4	Z3	CVC4	Z3	CVC4	Z3	CVC4	Z3
No (co)datatypes	221	209	775	777	52	51	1048	1037
Datatypes without Acyclic	227	–	780	–	52	–	1059	–
Full datatypes	227	213	786	791	52	51	1065	1055
Codatatypes without Unique	222	–	804	–	56	–	1082	–
Full codatatypes	223	–	804	–	**59**	–	1086	–
Full (co)datatypes	**229**	–	**815**	–	**59**	–	**1103**	–

- Experimental results: Implementation in CVC4 improves state of the art
  - Evaluated on proof obligations from Isabelle theorem prover

[Reynolds/Blanchette CADE15]

Theory Solvers for Harder Theories?

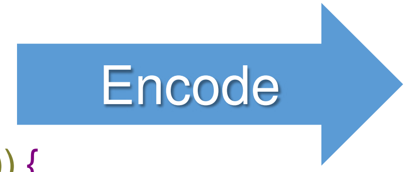
- So far: theory solvers for decision procedures
- However, in practice a theory solver need not be complete
  - E.g. what if background theory is undecidable?
- Examples of problems that use incomplete theory solvers:
  - Theory of Non-Linear (Integer) Arithmetic

  $\Rightarrow$ *Theory of Strings + Length constraints* **[Liang/Reynolds/Tinelli/Barrett/Deters CAV14]**

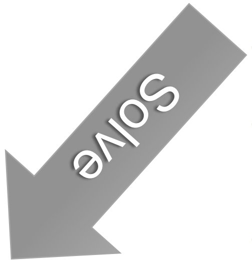
Theory of Strings + Length

Theory of Strings : Applications

```
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 the root permission*/
   }
}
```




```
(set-logic QF_S)
(declare-const input String)
(declare-const buff String)
(declare-const pass0 String)
(declare-const rest String)
(declare-const pass1 String)
(assert (= (str.len buff) 15))
(assert (= (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 (ite (= buff "PASSWORD")
          (= pass1 "Y")
          (= pass1 pass0)))
(assert (not (= buff "PASSWORD")))
(assert (= pass1 "Y"))
```




```
tiliang@milner:~/workspace/security/benchmarks/homemade$ ~/CVC4/bin/pt-cvc4 propsalex.smt2
sat
(model
(define-fun input () String "AAAAAAAAAAAAAAAY")
(define-fun buff () String "AAAAAAAAAAAAAAA")
(define-fun pass0 () String "Y")
(define-fun rest () String "")
(define-fun pass1 () String "Y")
)
```

- Security applications frequently rely on reasoning about string constraints

Theory of Strings + Length

- Signature Σ_S :
  - Constants from a fixed finite alphabet A^* =(a, ab, cbc...)
  - String concatenation $\_\cdot\_$: String $\times$ String $\rightarrow$ String
  - Length terms `len`(_) : String $\rightarrow$ Int
- Example input:

$$\texttt{len}(x) > \texttt{len}(y) \ \wedge \ x \cdot b = y \cdot ab$$

Theory of Strings + Length

- Theoretical complexity of:
  - Word equation problem is in PSPACE
  - ...with length constraints is OPEN
  - ...with extended functions, e.g. `replace`, is UNDECIDABLE
- Instead, focus on:
  - Solver that is efficient in practice
  - Tightly integrated into SMT solver architecture
    - Conflict-Driven Clause Learning, Propagation, Composable with other theories

Theory of Strings : Rule-Based Procedure

…

$$\text{F-Unify}\ \frac{\mathsf{F}\,s = (w, u, u_1) \quad \mathsf{F}\,t = (w, v, v_1) \quad s \approx t \in \mathcal{C}(\mathsf{S}) \quad \mathsf{S} \models \mathsf{len}\,u \approx \mathsf{len}\,v}{\mathsf{S} := \mathsf{S}, u \approx v}$$

$$\text{F-Split}\ \frac{\begin{array}{c}\mathsf{F}\,s = (w, u, u_1) \quad \mathsf{F}\,t = (w, v, v_1) \quad s \approx t \in \mathcal{C}(\mathsf{S}) \quad \mathsf{S} \models \mathsf{len}\,u \not\approx \mathsf{len}\,v \\ u \notin \mathcal{V}(v_1) \quad v \notin \mathcal{V}(u_1)\end{array}}{\mathsf{S} := \mathsf{S}, u \approx \mathsf{con}(v, z) \quad \| \quad \mathsf{S} := \mathsf{S}, v \approx \mathsf{con}(u, z)}$$

$$\text{F-Loop}\ \frac{\mathsf{F}\,s = (w, x, u_1) \quad \mathsf{F}\,t = (w, v, v_1, x, v_2) \quad s \approx t \in \mathcal{C}(\mathsf{S}) \quad x \notin \mathcal{V}((v, v_1))}{\begin{array}{c}\mathsf{S} := \mathsf{S},\ x \approx \mathsf{con}(z_2, z),\ \mathsf{con}(v, v_1) \approx \mathsf{con}(z_2, z_1),\ \mathsf{con}(u_1) \approx \mathsf{con}(z_1, z_2, v_2) \\ \mathsf{R} := \mathsf{R}, z \ \mathsf{in}\ \mathsf{star}(\mathsf{set}\ \mathsf{con}(z_1, z_2)) \quad \mathsf{C} := \mathsf{C}, t\end{array}}$$

…

- Existing approaches rely on reduction to bitvectors, e.g. HAMPI [Kiezun 2009]
- Instead, we use an algebraic rule-based procedure for strings, which:
  - Infers equalities over strings based on length constraints
  - Models interaction of string + arithmetic solvers
  - Recognizes conflicts due to cardinality of alphabet

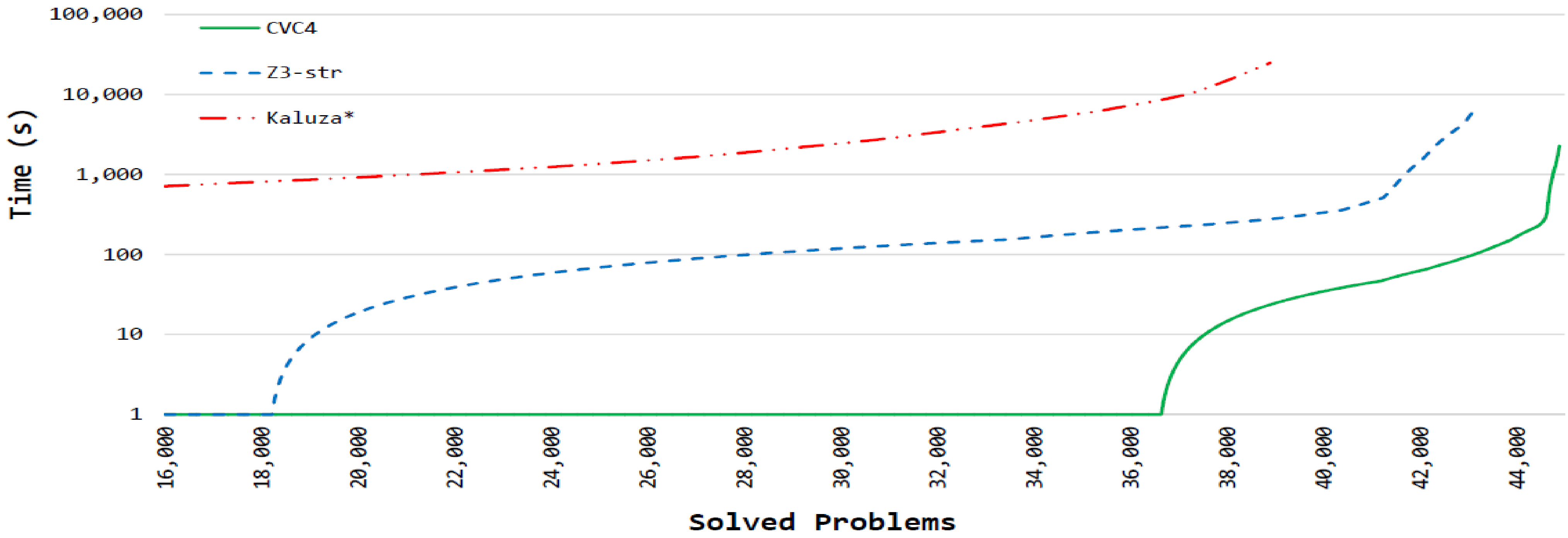
[Liang/Reynolds/Tinelli/Barrett/Deters CAV14]

Theory of Strings : Theoretical Results

- For strings + length:
  - Procedure is:
    - Refutation sound, even for strings of unbounded length
    - Solution sound
  - (A version of) procedure is:
    - Solution complete
      - When problem is “SAT”, it will eventually find a model (finite model finding)
  - When input is in acyclic form (variables only on 1 side of equalities),
    - Refutation complete
      - When problem is “UNSAT”, it will derive a refutation

[Liang/Reynolds/Tinelli/Barrett/Deters CAV14]

Theory of Strings : Experimental Results



- Tested 50,000 VCs in web security applications (Kudzu)
- Implementation in CVC4 significantly improved state-of-the-art
  - In terms of precision, performance, and accuracy

[Liang/Reynolds/Tinelli/Barrett/Deters CAV14]

Extending the Theory of Strings

- Theory of strings can be extended with support for:
  - Regular expressions
    - E.g. $x \in (a \cup (bb)^*)^*$
    - Decision procedure for regular memberships + length **[submitted, FroCos15]**
  - Regular languages
    - E.g. $x \in (y \cdot b)^*$
  - Extended functions
    - E.g. `substr`, `contains`, `replace`, `prefixOf`, `suffixOf`, `str.indexOf`, `str.to.int`, `int.to.str`, `strcmp`
    - Occur frequently in practice
    - When signature includes these, problem is generally undecidable

What about arbitrary quantified formulas?

- What if constraints do not fit an existing theory/decision procedure?
  - Frame axioms in software verification
  - Universal safety properties
  - Axiomatization of unsupported theories
  - …
- Want SMT solver to handle arbitrary first-order quantified formulas
  - E.g. `∀x.f(x)>0`, `∀x.select(A,x)=2*x`

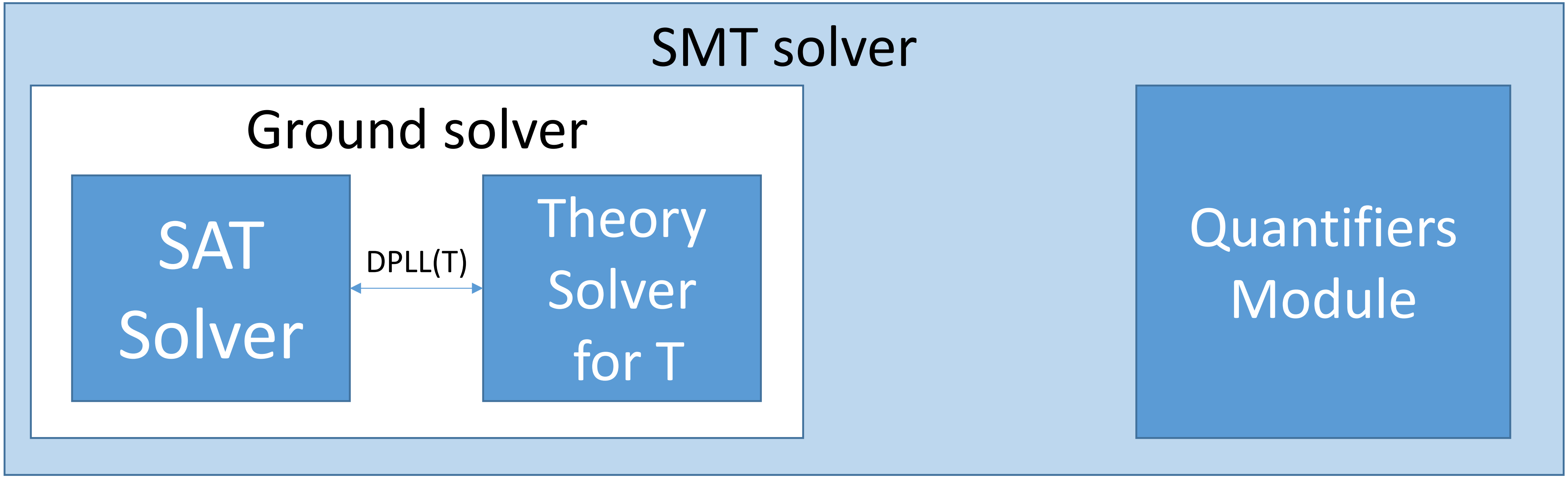
Approaches for Quantified Formulas in SMT

- Heuristic approaches
  - Incomplete, focus on finding unsatisfiable
  - Example:
    - E-matching [Detlefs et al 2003, Ge et al 2007, de Moura/Bjorner 2007]
- Complete approaches
  - Target particular fragments of FOL
  - Examples:
    - Local theory extensions [Sofronie-Stokkermans 2005]
    - Array fragments [Bradley et al 2006, Alberti et al 2014]
    - Complete instantiation [Ge/de Moura 2009]
    - Finite model finding **[Reynolds et al 2013]**

Focus of next part of the talk

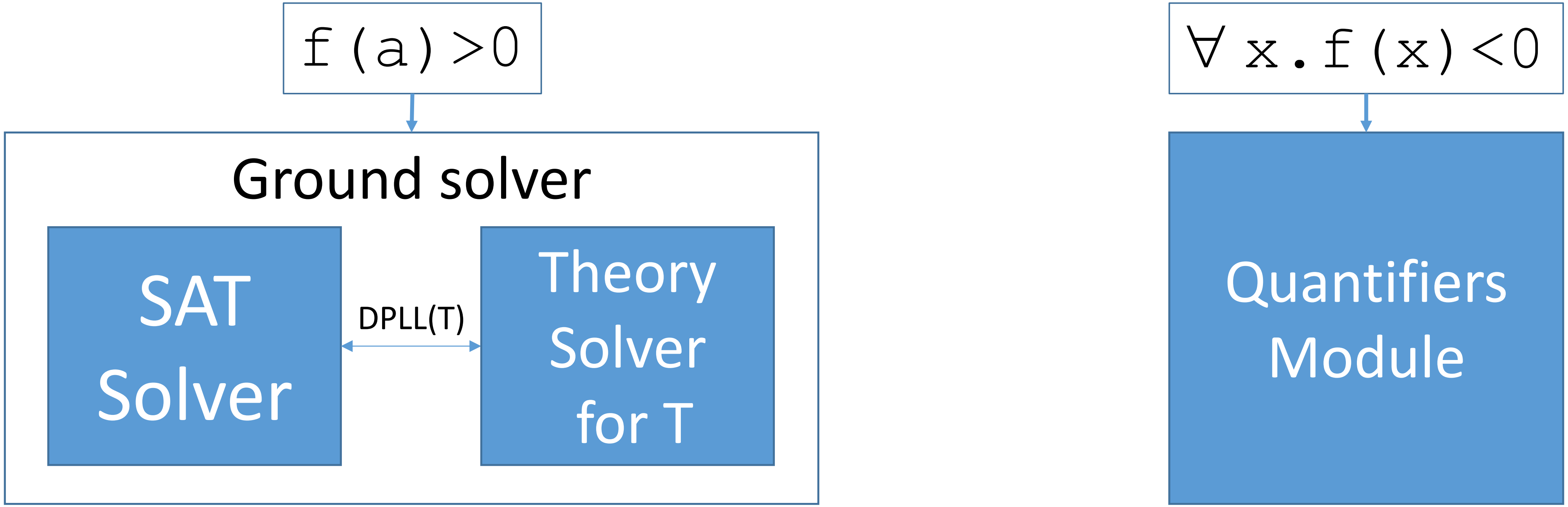
Finite Model Finding for Quantified Formulas in SMT

SMT Solver + Quantified Formulas



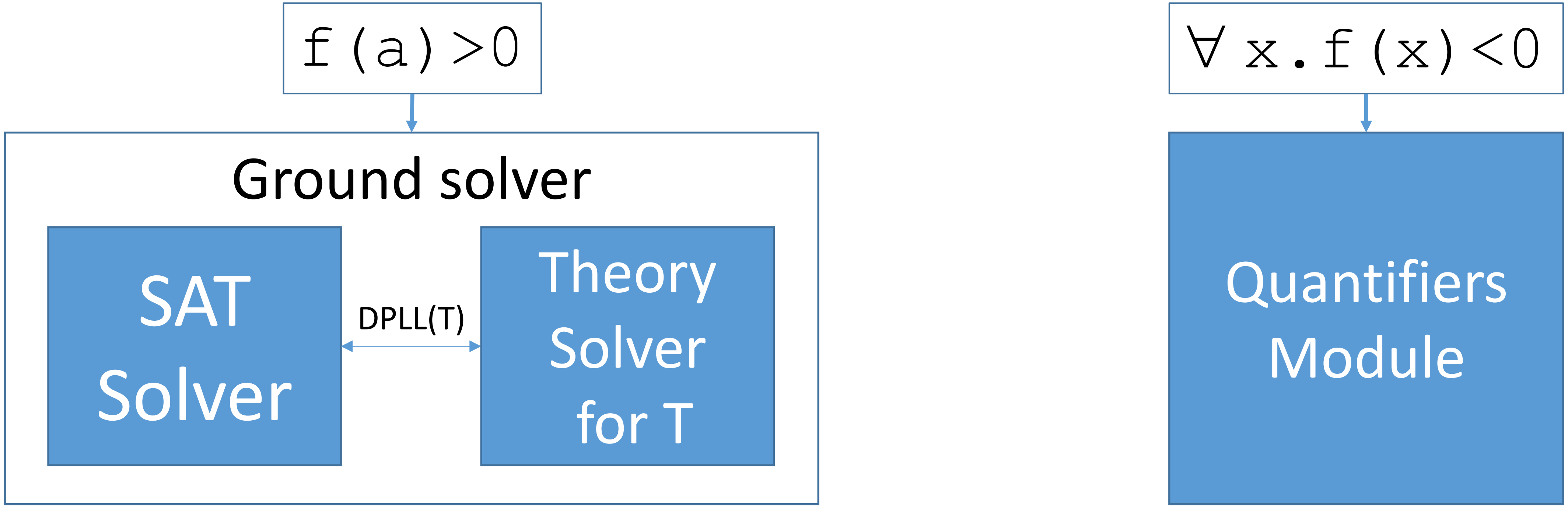
- SMT solvers support for (first-order) quantified formulas $\forall$

SMT Solver + Quantified Formulas

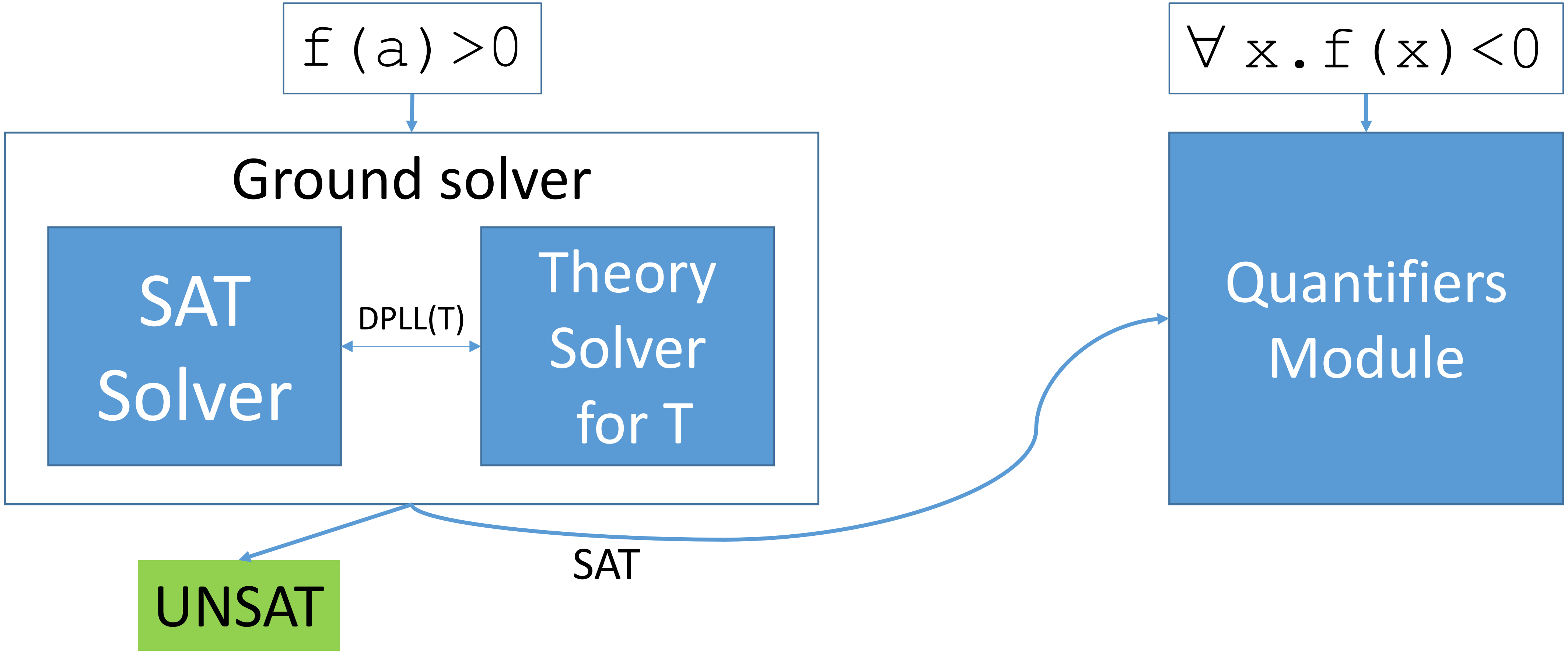


- For input $f(a)>0 \wedge \forall x. f(x)<0$
  - Ground solver maintains a set of ground (variable-free) constraints : $f(a)>0$
  - Quantifiers Module maintains a set of axioms : $\forall x. f(x)<0$

SMT Solver + Quantified Formulas

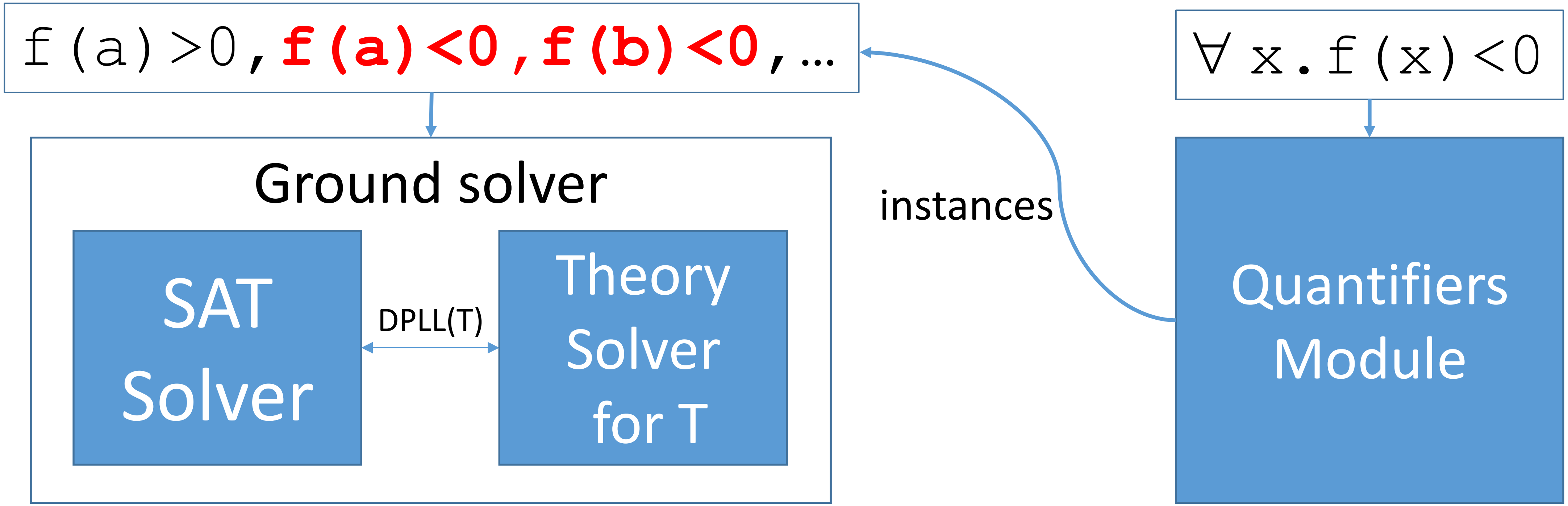


SMT Solver + Quantified Formulas



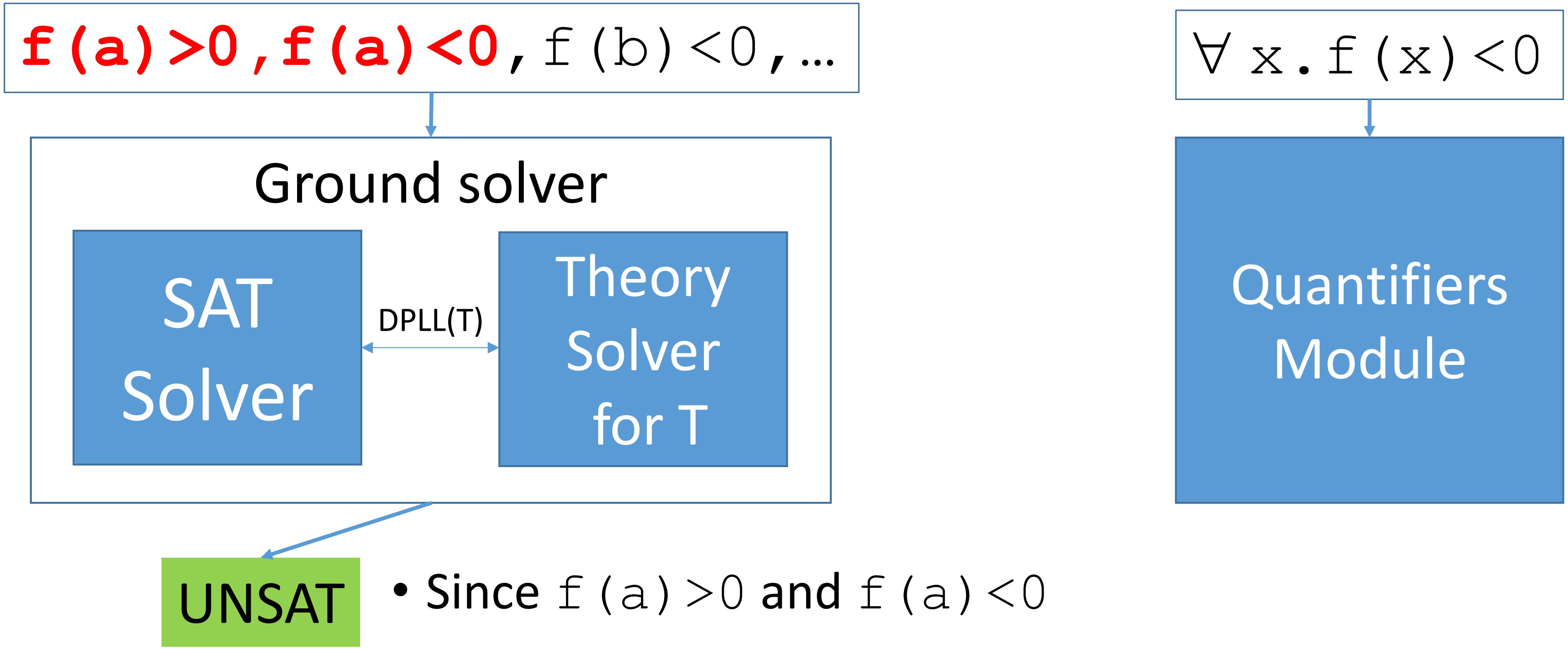
- Ground solver checks T-satisfiability of current set of constraints

SMT Solver + Quantified Formulas



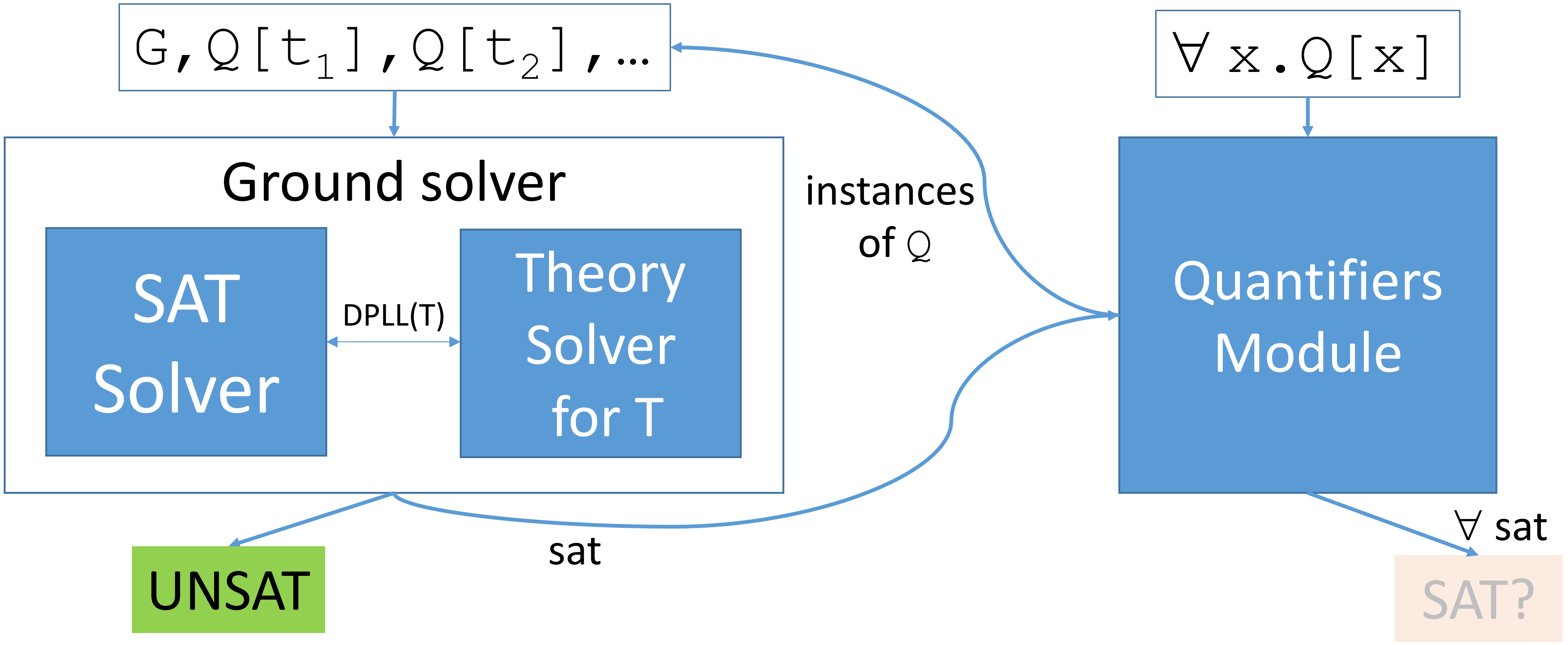
- Quantifiers Module adds instances of axioms
  - Goal : add instances until ground solver can answer “unsat”

SMT Solver + Quantified Formulas



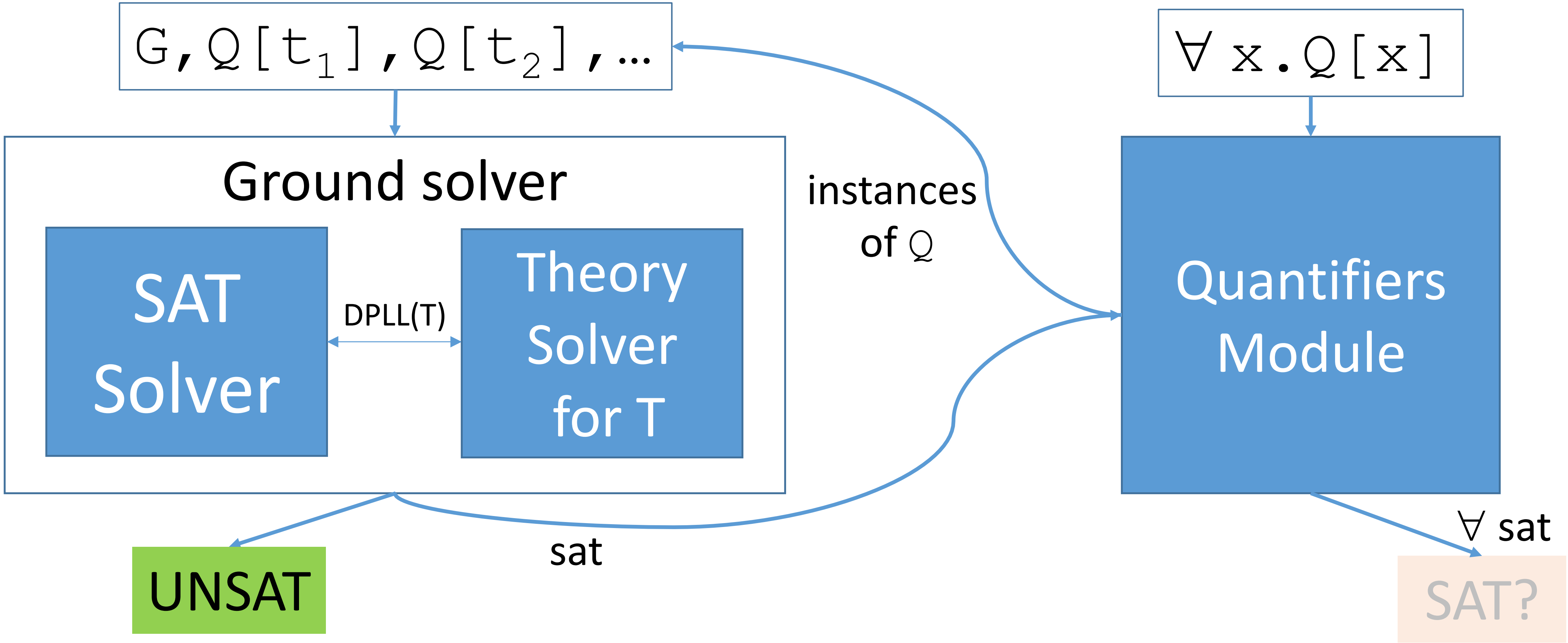
- Since `f(a)>0` and `f(a)<0`

How SMT Solvers Handle Quantified Formulas



- Generally, a sound but incomplete procedure
  - Difficult to answer SAT (when have we added enough instances of `Q[x]`?)

How SMT Solvers Handle Quantified Formulas



$\Rightarrow$ *Lack of ability to answer SAT is major weakness*

Finite Model Finding : Application

- Deductive Verification Framework [Goel et al 2012] used at Intel Corporation for:
  - Architecture/Security Validation for Hardware Systems

Definitions

```
type resource
const resource nullvar array(resource, bool) valid = mk_array[resource](false)
var array(resource, int) count
var array(process, resource) ref = mk_array[process](null)
…
module S = Set<type process>
```

Transition System

```
transition create (resource r)
require (r != null, !valid[r]){ count[r] := 0; }
…
```

Properties

```
def bool prop = forall (process p) (ref[p] != null => valid[ref[p]])
def bool refs_non_zero = forall (process p) (ref[p] != null => count[ref[p]] > 0)
…
```

Verification Conditions

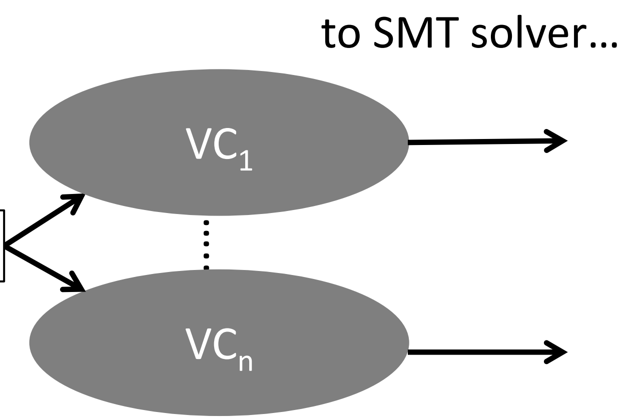
```
goal main = invariant prop assuming refs_non_zero
goal rnz  = formula (… && prop && … => refs_non_zero)
```

Finite Model Finding : Application

```
type resource
const resource null
type process
var array(resource, bool) valid = mk_array[resource](false)
var array(resource, int) count
var array(process, resource) ref = mk_array[process](null)
…
module S = Set<type process>

transition create (resource r)
require (r != null, !valid[r]){
  valid[r] := true;
  count[r] := 0;
}
…

def bool prop = forall (process p) (ref[p] != null => valid[ref[p]])
def bool refs_non_zero = forall (process p) (ref[p] != null =>
count[ref[p]] > 0)
…

goal main = invariant prop assuming refs_non_zero
…
goal rnz  = formula (… && prop && … => refs_non_zero)
```

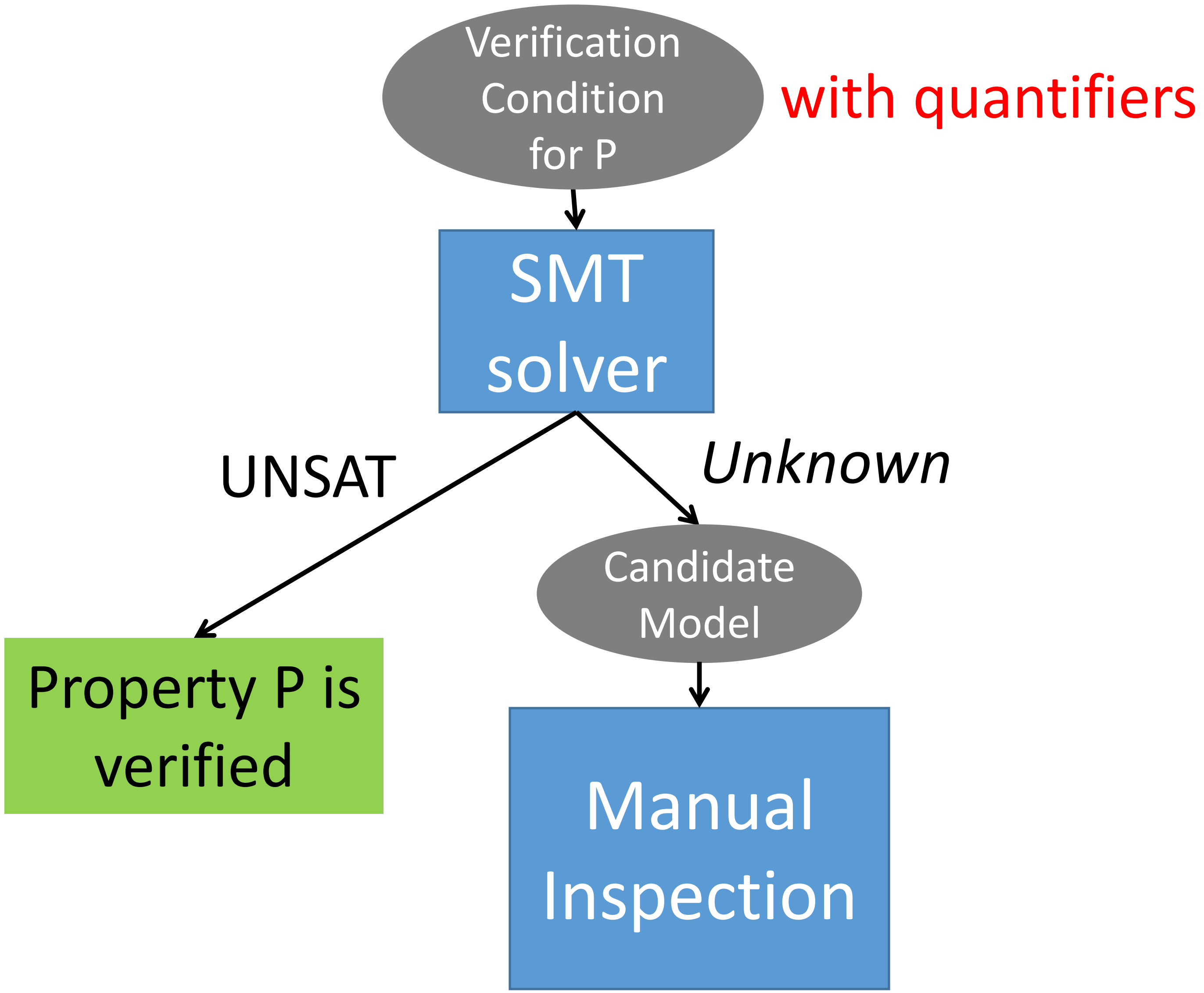
Verification Conditions

goal main = invariant prop assuming refs_non_zero

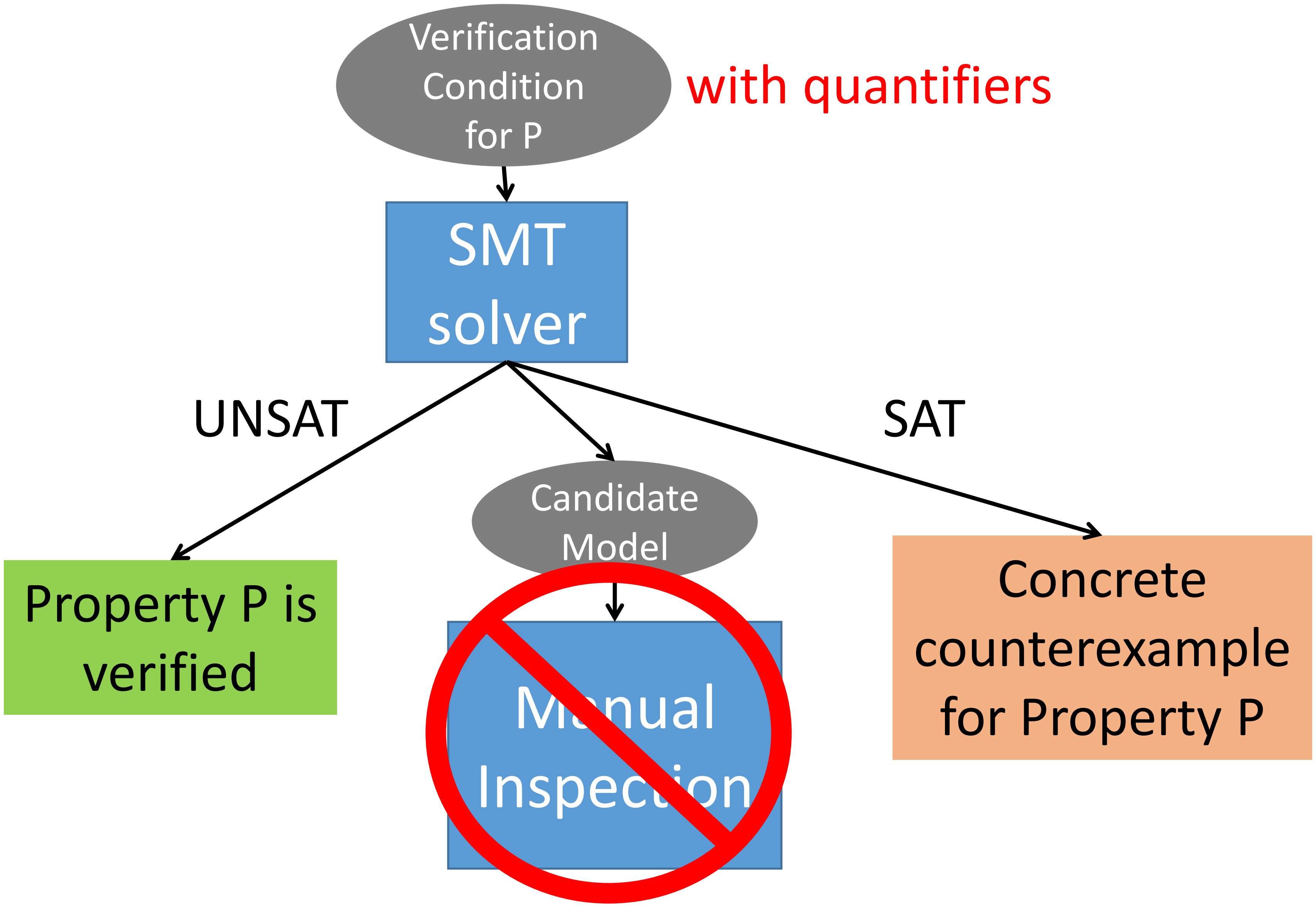


- Verification conditions translated into (multiple) SMT queries, requiring:
  - Theories (arithmetic, bit vectors, datatypes, …)
  - Quantified formulas for stating universal properties over:
    - Memory addresses, resources, processes, …

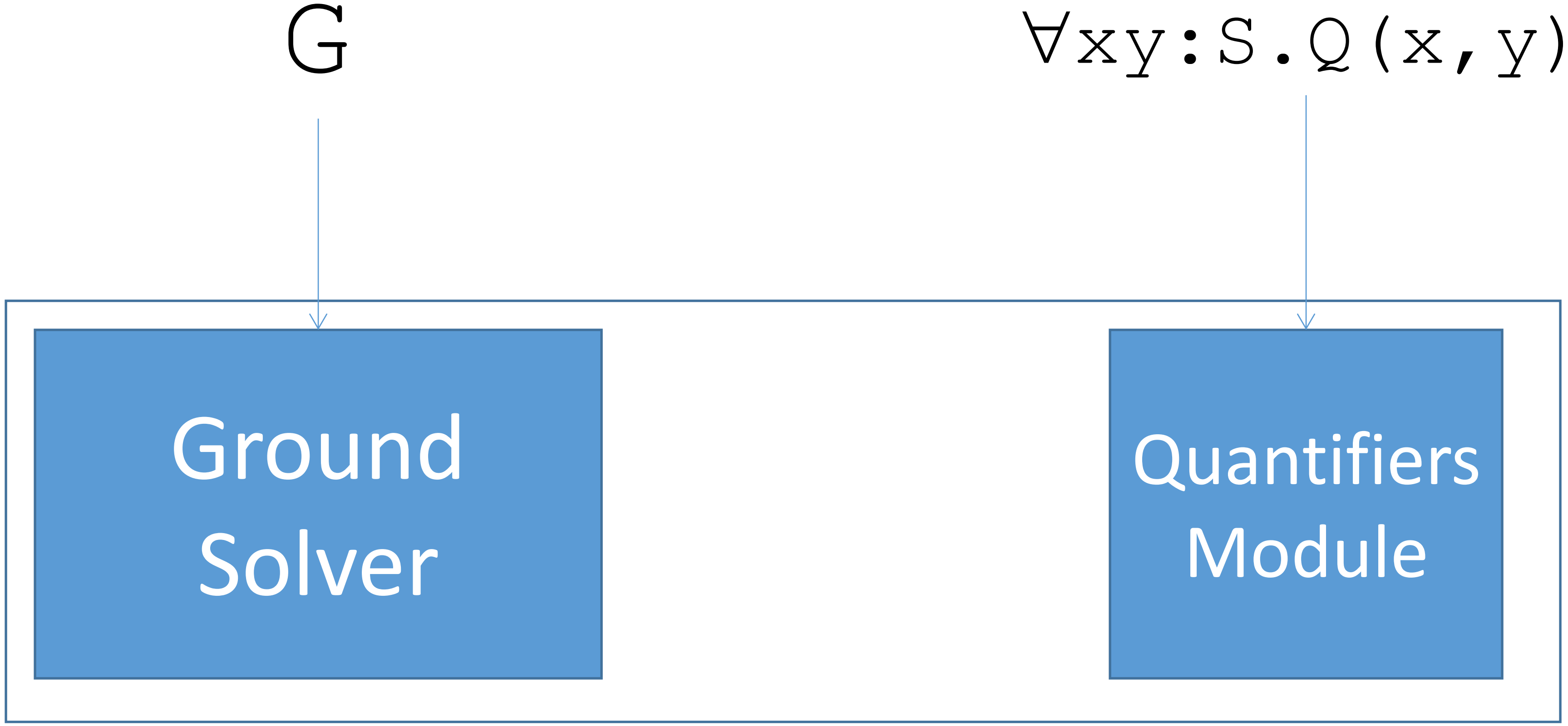
Why are Models Important?



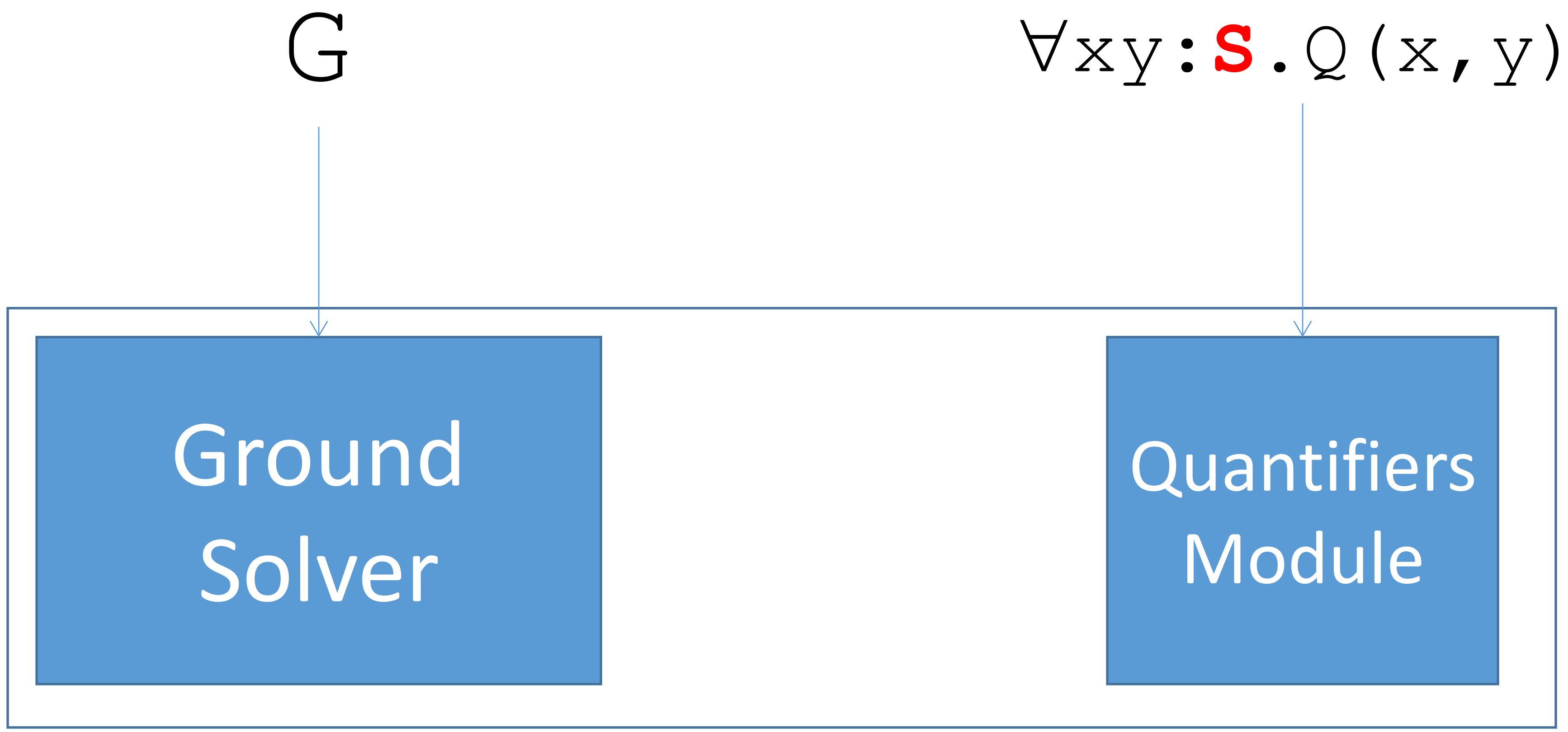
Why are Models Important?



Finite Model Finding in SMT



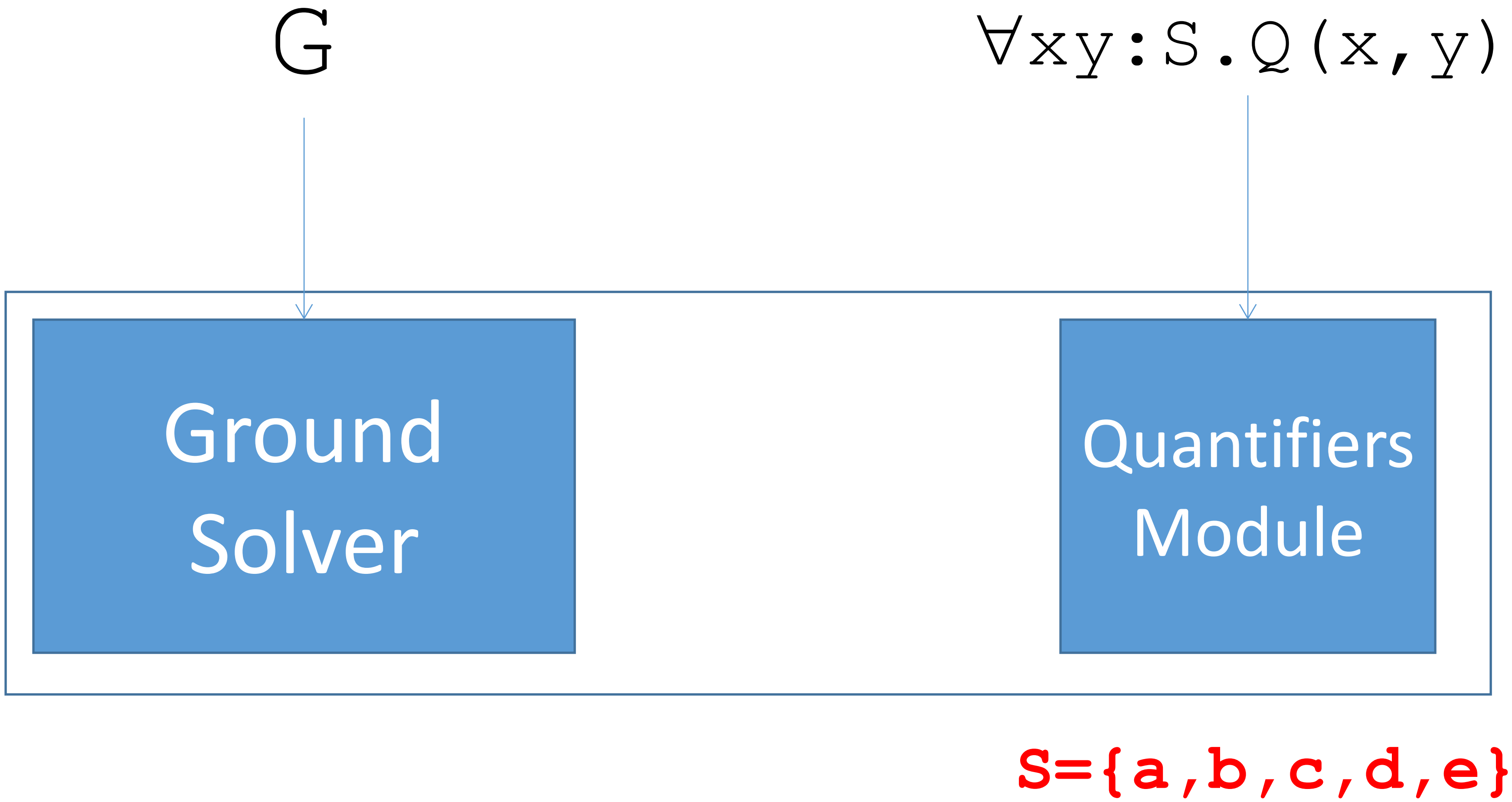
Finite Model Finding in SMT



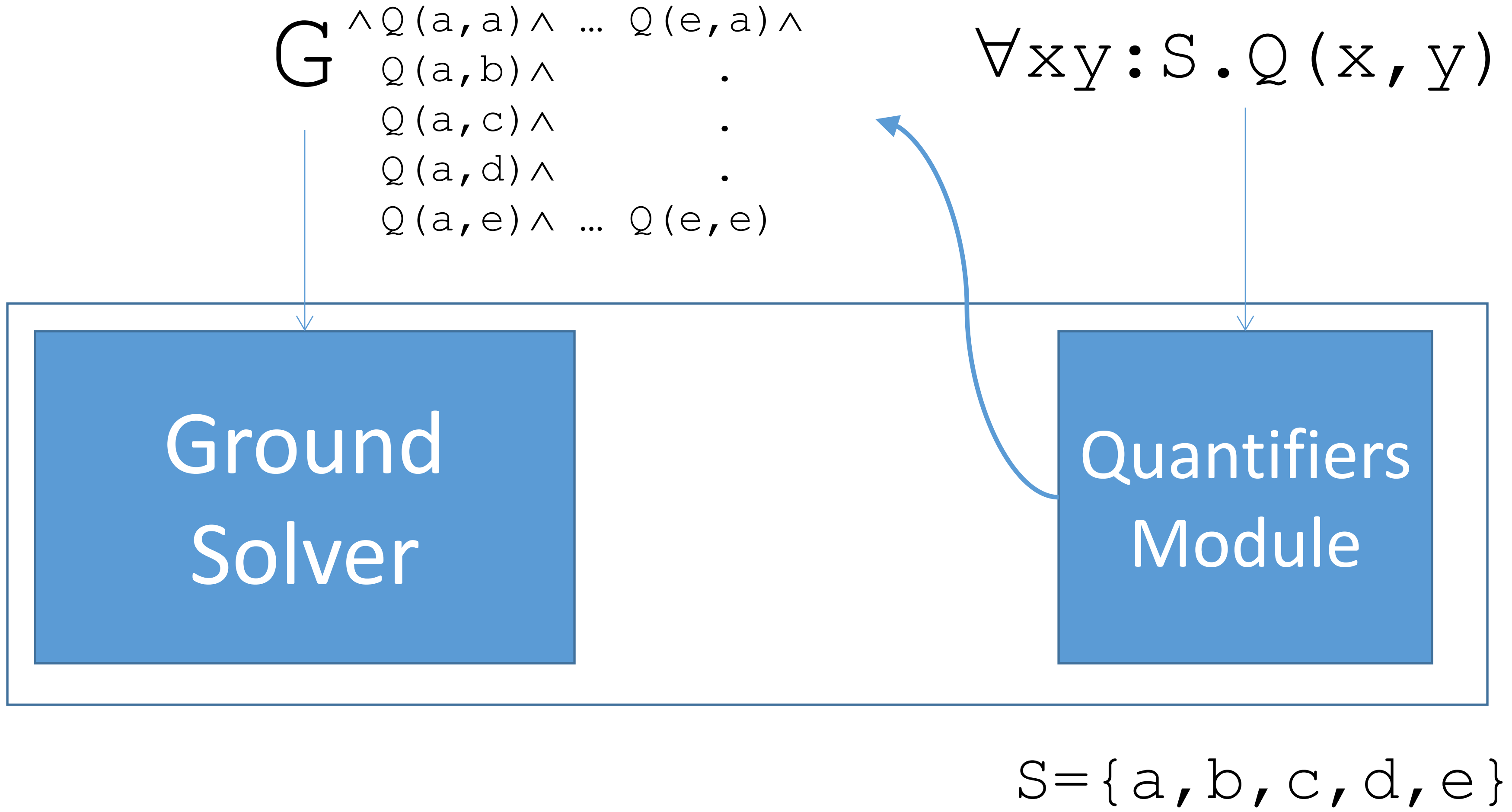
⇒*If* ***S*** *has finite interpretation,*

- *use finite model finding*

Finite Model Finding in SMT

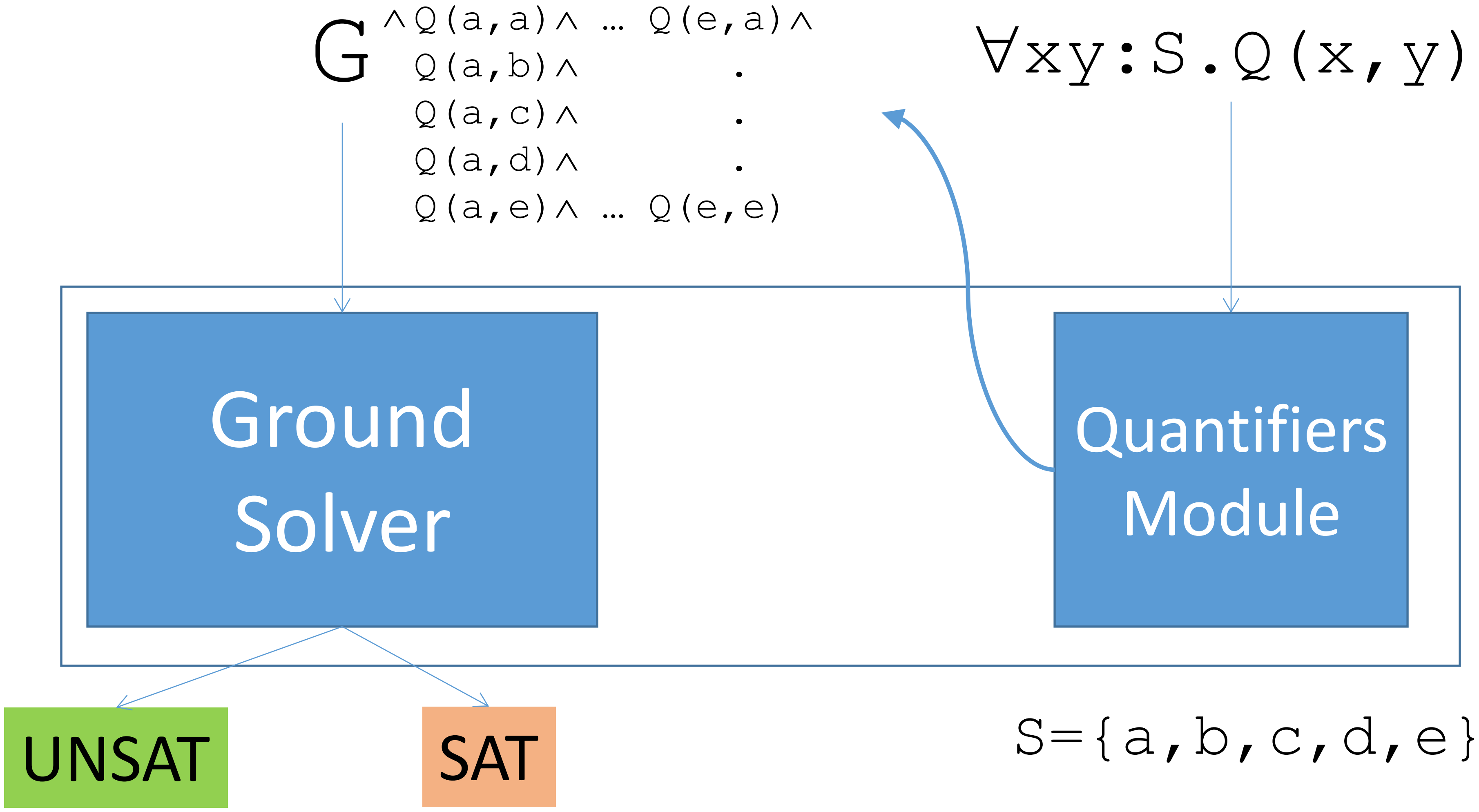


Finite Model Finding in SMT



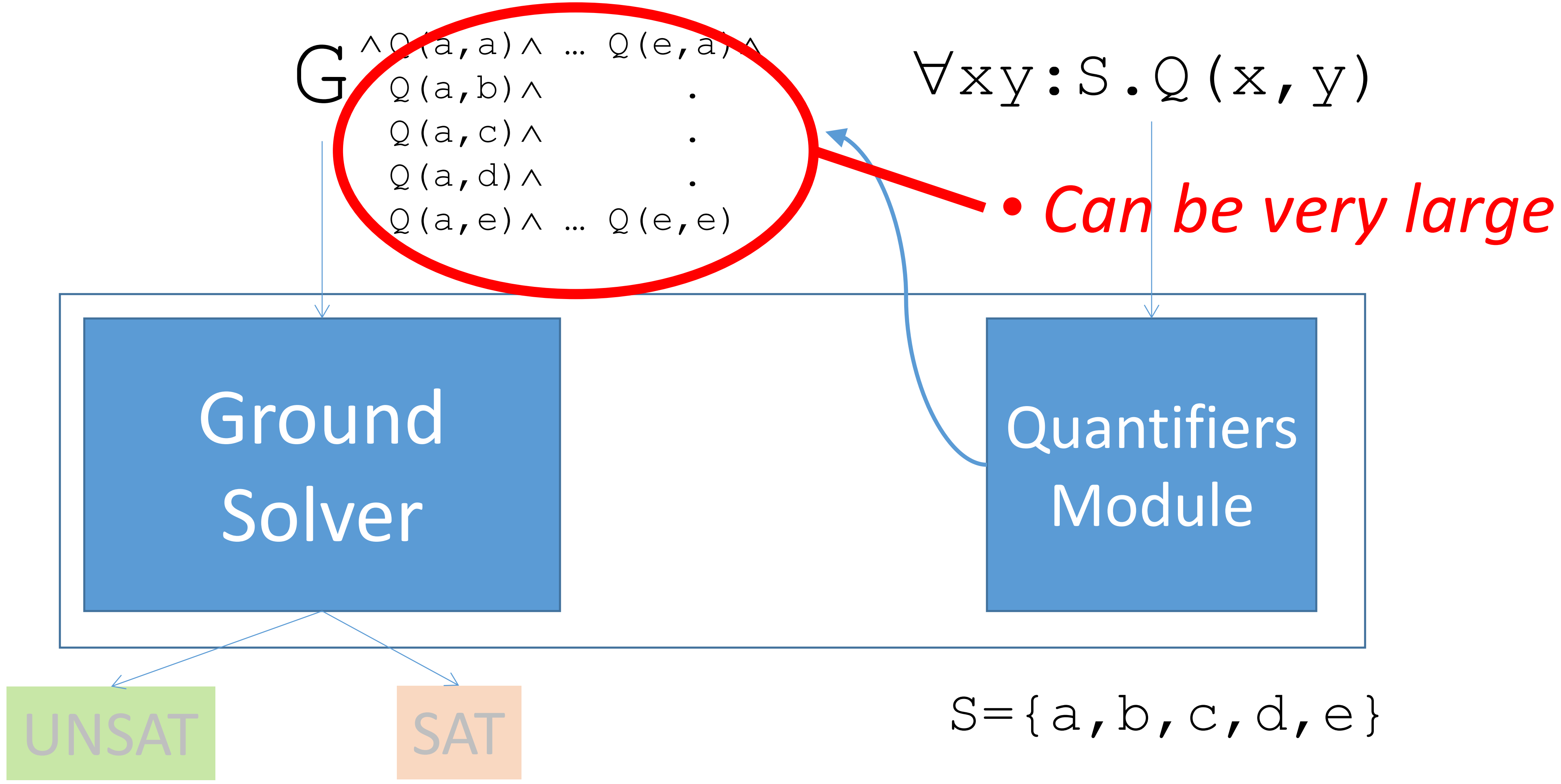
- Reduction of quantified formulas to ground formulas

Finite Model Finding in SMT



⇒ Ability to answer SAT, assuming decision procedure for `G∧Q(a,a)∧…`

Finite Model Finding in SMT

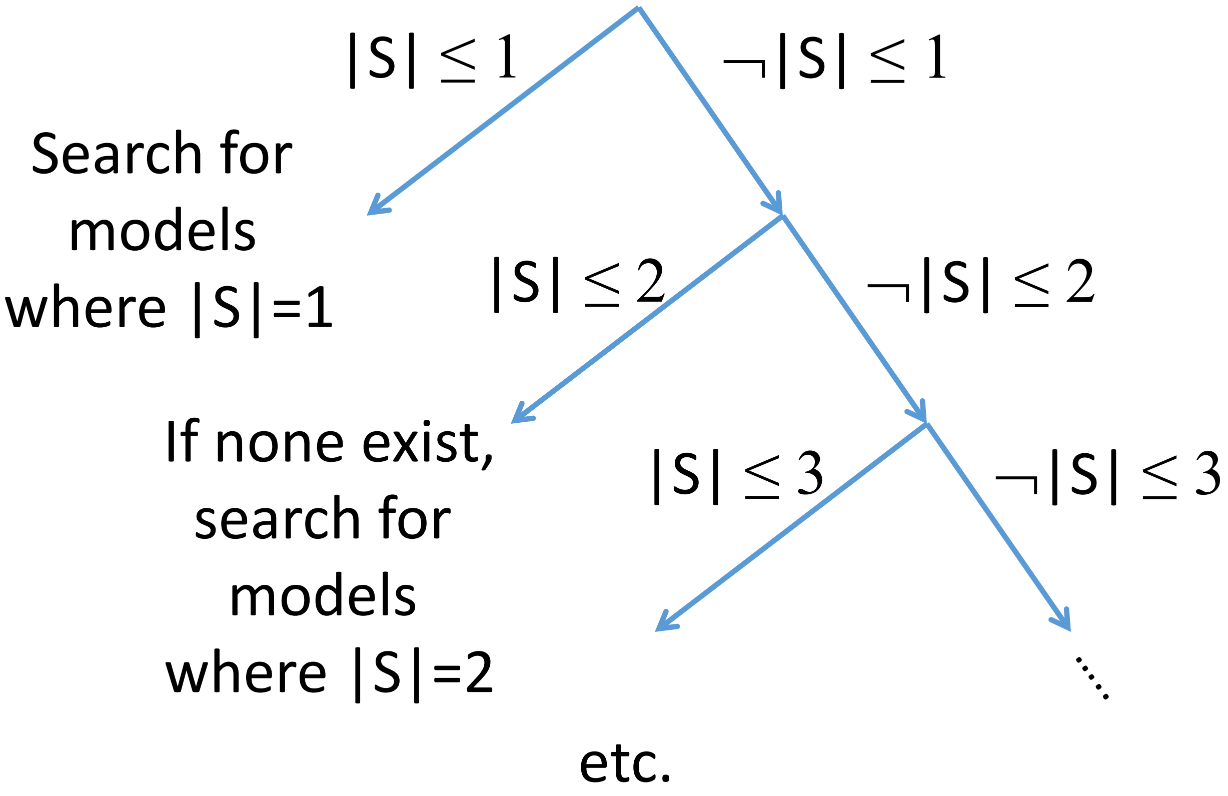


Finite Model Finding in SMT

- Address large # instantiations by:
  1. Minimizing model sizes **[Reynolds et al CAV13]**
     - Find interpretation that minimizes the #elements in S
  2. Only add instantiations that refine model **[Reynolds et al CADE13]**
     - Model-based quantifier instantiation [Ge/deMoura CAV 2009]

Finite Model Finding : Minimizing Model Sizes

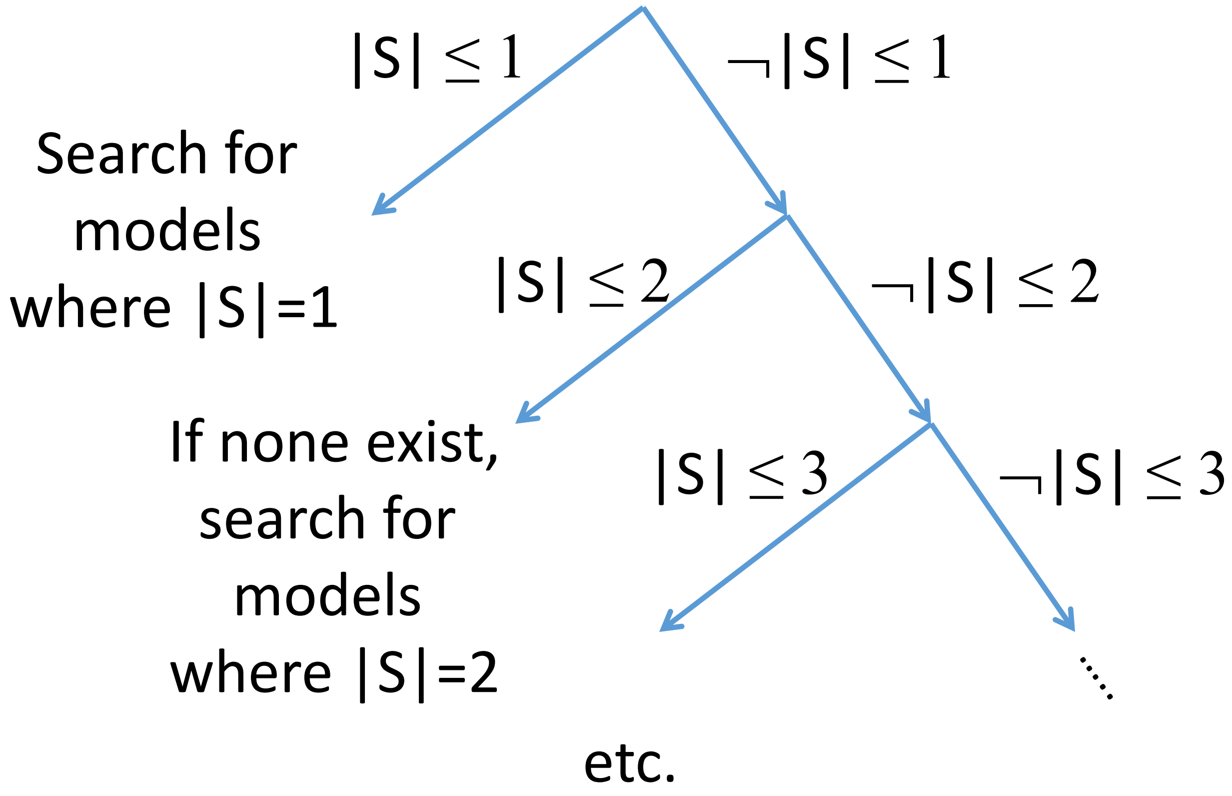
- Minimize model sizes using a theory solver for cardinality constraints



[Reynolds/Tinelli/Goel/Krstic CAV13]

Finite Model Finding : Minimizing Model Sizes

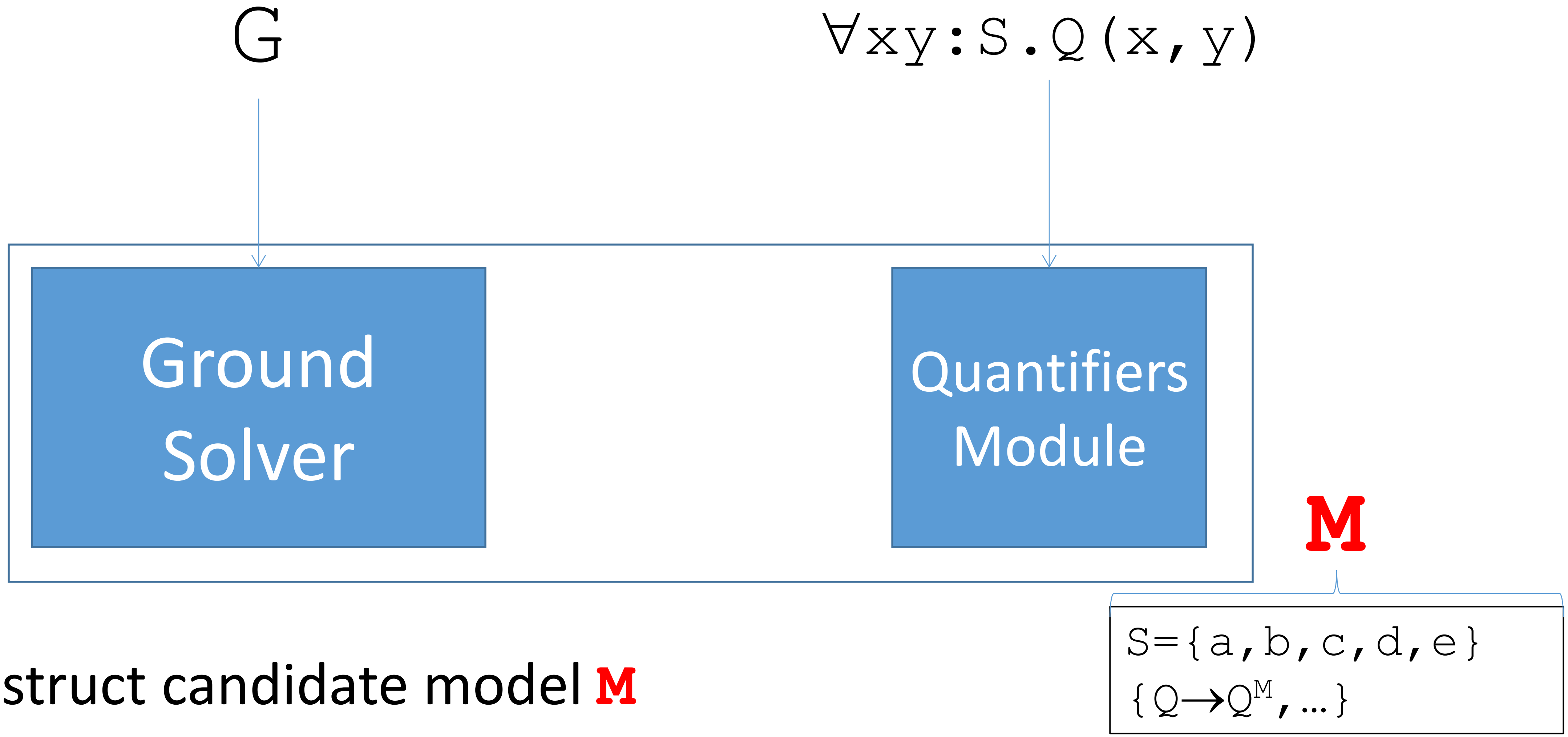
- Minimize model sizes using a theory solver for cardinality constraints



$\Rightarrow$ If model exists where $|S| \leq 3$, only need 3*3=9 instances instead of 5*5=25 instances

[Reynolds/Tinelli/Goel/Krstic CAV13]

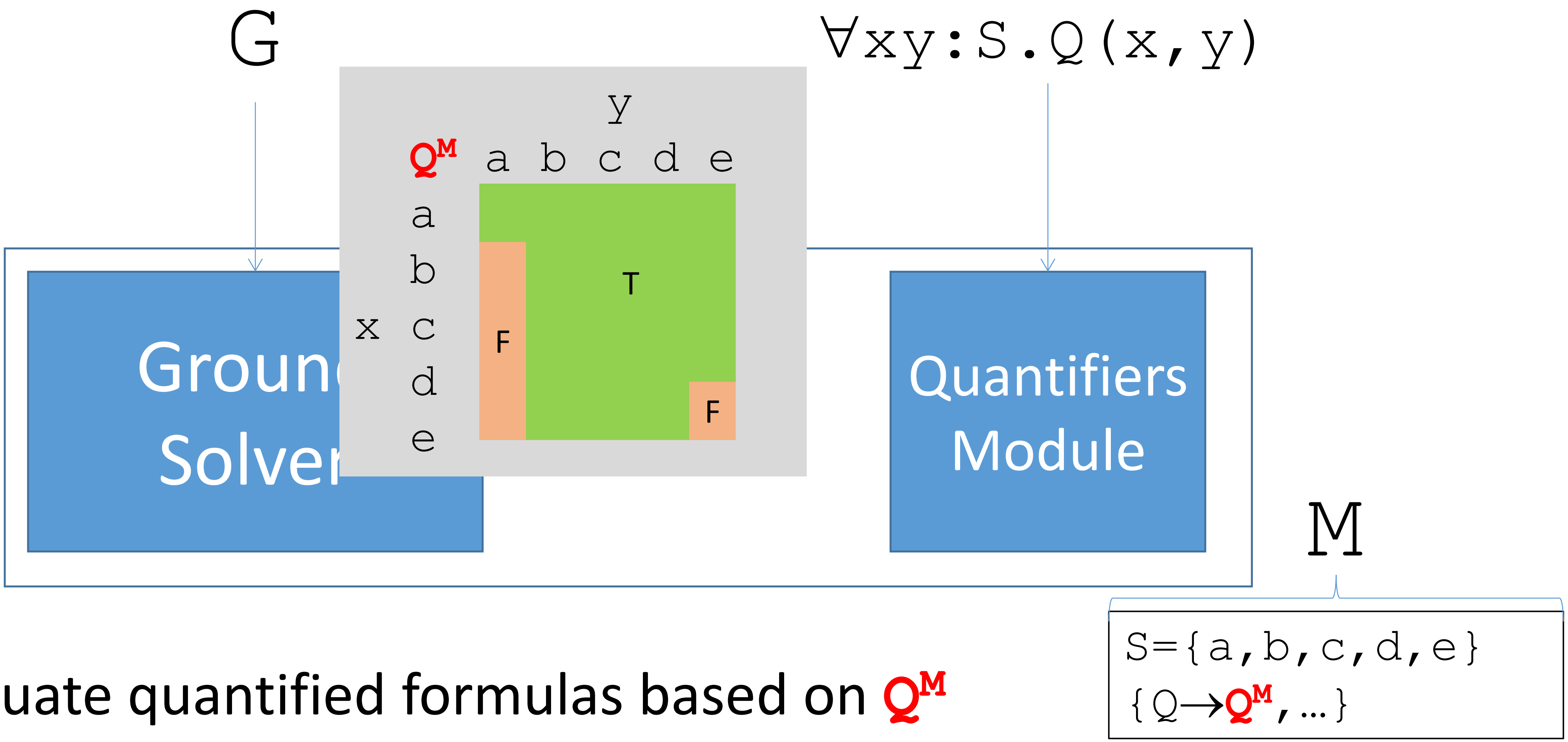
Finite Model Finding : Model-Based Instantiation



- Construct candidate model **M**

[Reynolds/Tinelli/Goel/Krstic/Barrett/Deters CADE13]

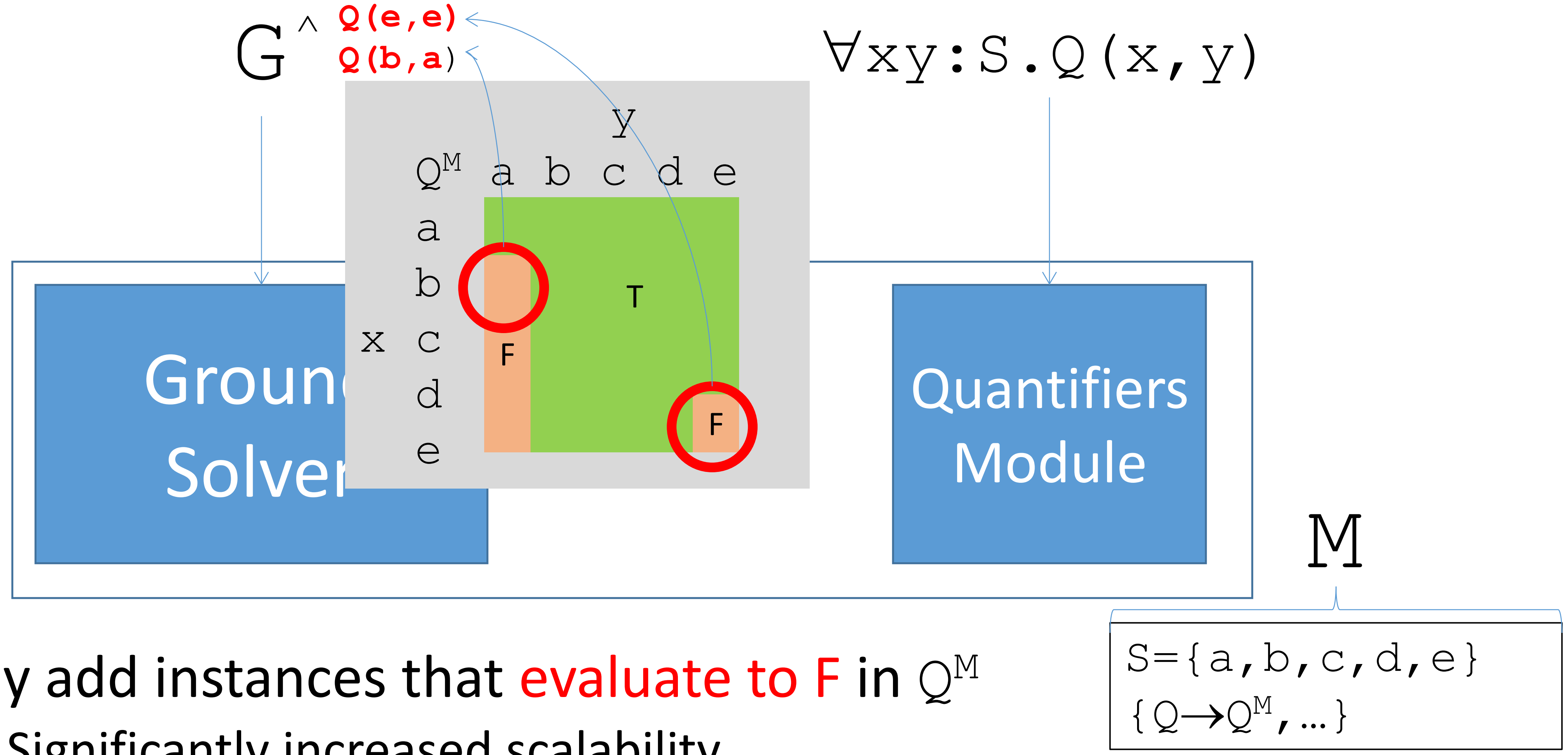
Finite Model Finding : Model-Based Instantiation



- Evaluate quantified formulas based on Q^M

[Reynolds/Tinelli/Goel/Krstic/Barrett/Deters CADE13]

Finite Model Finding : Model-Based Instantiation



- Only add instances that evaluate to F in Q^M

  ⇒ Significantly increased scalability

[Reynolds/Tinelli/Goel/Krstic/Barrett/Deters CADE13]

Results : Hardware Verification at Intel

SAT	german	refcount	agree	apg	bmk	Total	Time
#	45	6	42	19	37	149	
z3	45	1	0	0	0	46	8.1
cvc4	2	0	0	0	0	2	0.0
cvc4+f	**45**	**6**	**42**	**19**	**37**	**149**	409.8

UNSAT	german	refcount	agree	apg	bmk	Total	Time
#	145	40	488	304	244	1221	
z3	145	**40**	**488**	304	244	**1221**	31.0
cvc4	**145**	40	484	**304**	**244**	1217	21.3
cvc4+f	145	40	488	302	244	1219	1185.0

cvc4 :

- f : finite model finding

- Benchmarks taken from DVF tool at Intel
- Improved state of the art for **SAT** for SMT problems with $\forall$
- Can be competitive for **UNSAT** as well

[Reynolds/Tinelli/Goel/Krstic/Barrett/Deters CADE13]

Results : CASC Competition

First-order Non-theorems	iProver 1.0-SAT	Paradox 3.0	CVC4 1.2-SAT	E 1.8	Nitrox 2013	Vampire 3.0-SAT	E-KRHyp 1.4	iProver-E 0.85
Solved/150	122/150	99/150	96/150	79/150	79/150	78/150	67/150	37/150
Av. CPU Time	52.47	2.28	25.94	20.94	29.70	15.89	7.57	30.77
Solutions	122/150	99/150	96/150	79/150	79/150	78/150	67/150	0/150
µEfficiency	165	549	204	396	36	395	292	92
SOTAC	0.28	0.23	0.19	0.22	0.24	0.20	0.19	0.15
New Solved	1/4	1/4	1/4	1/4	1/4	1/4	0/4	0/4

- Competitive with existing approaches for model finding in ATP community
- CVC4 placed 3rd in non-theorems division of CASC 24
  - Is competitive with state-of-the-art ATP systems

Ongoing work/applications

- SMT solvers with support for $\forall$ are doing increasingly complex tasks:
  - As an efficient *first order theorem prover*
    - **[Reynolds/Tinelli/de Moura FMCAD 2014]**
  - As an *inductive reasoner for program verification*
    - **[Reynolds/Kuncak VMCAI 2015]**
  - As a tool for *syntax-guided software synthesis*
    - **[Reynolds/Deters/Kuncak/Tinelli/Barrett CAV 2015]**
  - In development: As a *program analyzer*
    - Idea: built-in support for (recursive) function definitions in SMT

```
(define-fun-rec len (x List) Int (ite (is-cons x) (1 + (len (tail x))) 0))
```

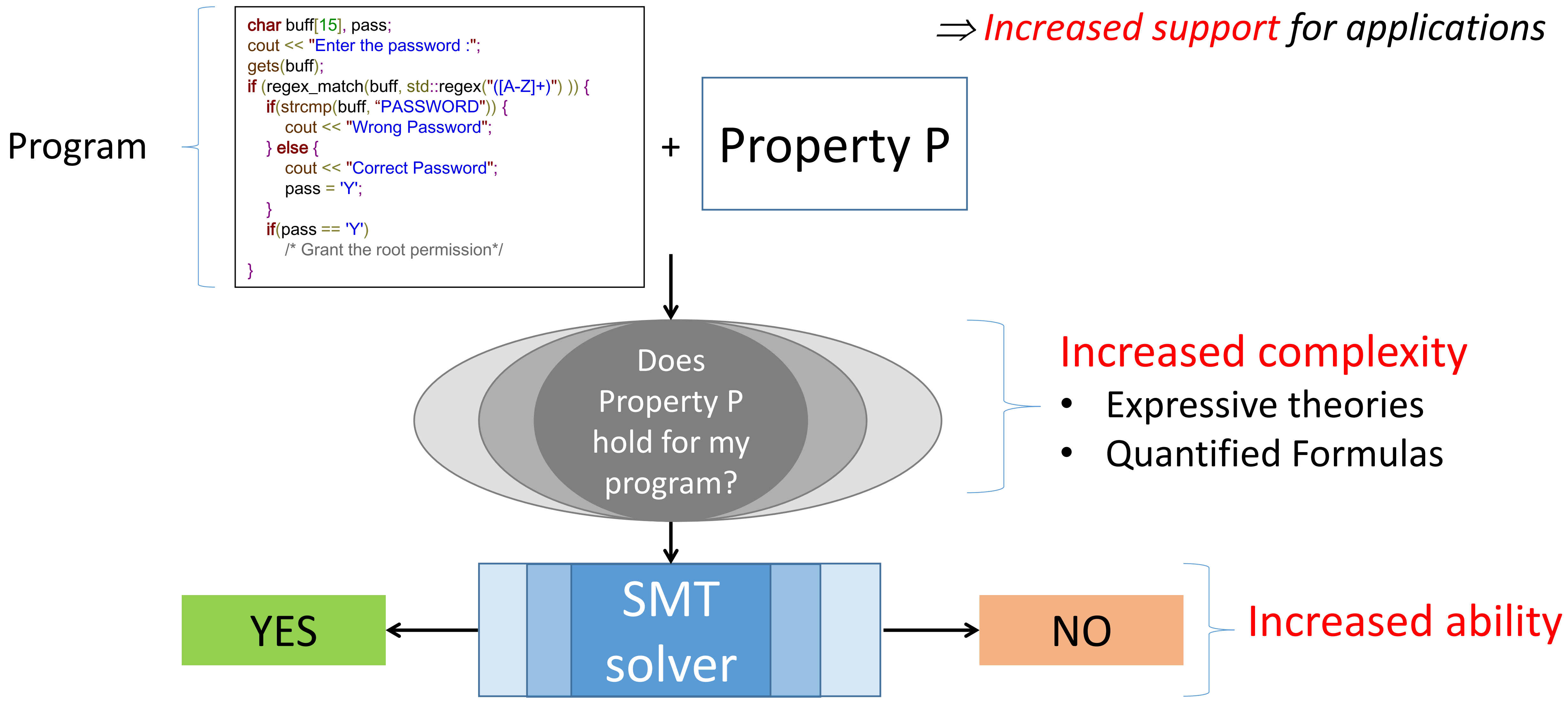
$\downarrow$

$\forall_{\mathtt{len}}$ `x. len(x)= (ite (is-cons x) (1 + (len (tail x))) 0)`

Conclusions

- Satisfiability Modulo Theories (SMT) is
  - Mature technology, both in theory and practice
  - …but is still evolving:
    - Improved approaches for (combinations) of theories
    - Solvers for new theories:
      - Floating Points, Sets, *(Co)datatypes, Extended Strings + Length*, Regular Languages
      - …
    - Specialized approaches for first-order quantified formulas

Conclusions



Thanks for your Attention!

- Collaborators:
  - Cesare Tinelli, Clark Barrett, Morgan Deters, Tim King, Liana Hadarean, Dejan Jovanovic, Kshitij Bansal, Tianyi Liang, Nestan Tsiskaridze, Amit Goel, Sava Krstic, Leonardo de Moura, Viktor Kuncak, Jasmin Blanchette

- Questions?