Satisfiability Modulo Theories: Beyond Decision Procedures

Andrew Reynolds

May 20, 2015
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';
    }
} else if(pass == 'Y')
    /* Grant the root permission*/

SMT solver
Does Property P hold for my program?

YES
NO
SMT Solvers for Software Verification/Security

```cpp
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*/
```

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 $\Sigma_T$ containing sorts and function symbols
  - A class of models $I_T$ describing the intended interpretations of symbols in $\Sigma_T$

- For example, linear integer arithmetic (LIA):
  - $\Sigma_{\text{LIA}}$ contains functions \{ $+$, $-$, $<$, $\leq$, $>$, $\geq$, 0, 1, 2, 3, ... \}
  - Each $I \in I_{\text{LIA}}$ interpret functions in $\Sigma_{\text{LIA}}$ in standard way:
    - $1+1 = 2$, $1+2 = 3$, ..., $0 > 1 = \text{false}$, ...

- Number of widely-supported theories in SMT:
  - **Bitvectors**: $\text{bvsgt}(a, \#\text{bin0001})$
  - **Arrays**: $\text{select}(\text{store}(a, 5, b), c) = 5$
  - **Datatypes**: $\text{tail}(\text{cons}(a, b)) = b$
  - ...
What is a Decision Procedure for T?

• Input: a set of T-constraints $M$, under some syntactic restriction

• A decision procedure is a method that terminates with output:
  • “$M$ is T-satisfiable”, i.e. there is a solution
  • “$M$ is T-unsatisfiable”
What is a Decision Procedure for T?

• Input: a set of T-constraints $M$, under some syntactic restriction

• A decision procedure is a method that terminates with output:
  • “$M$ is T-satisfiable”, i.e. there is a solution
  • “$M$ is T-unsatisfiable”
What is a Decision Procedure for T?

• Input: a set of T-constraints $M$, under some syntactic restriction
• A decision procedure is a method that **terminates** with output:
  • “$M$ is T-satisfiable”, i.e. there is a solution
    • Must be **solution-sound**, returns “$M$ is T-satisfiable” only when $M$ is T-satisfiable
  • “$M$ is T-unsatisfiable”
What is a Decision Procedure for T?

• Input: a set of T-constraints $M$, under some syntactic restriction
• A decision procedure is a method that terminates with output:
  • “$M$ is T-satisfiable”, i.e. there is a solution
    • Must be solution-sound, returns “$M$ is T-satisfiable” only when $M$ is T-satisfiable
  • “$M$ is T-unsatisfiable”
    • Must be refutation-sound, returns “$M$ is T-unsatisfiable” only when $M$ is T-unsatisfiable
How are Decision Procedures Implemented in SMT?

- Decision Procedures are implemented as theory solvers.

\[ x > 5 \land x \leq 4 \]

Boolean combination of T-constraints

• Decision Procedures are implemented as theory solvers.
How are Decision Procedures Implemented in SMT?

- Decision Procedures are implemented as *theory solvers*.

Given: \( x > 5 \land x \leq 4 \)

M = \{ x > 5, x \leq 4 \}

• 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 \( \neg C \)

\[
x > 5 \land x \leq 4 \land (\neg x > 5 \lor \neg x \leq 4)
\]

\[M = \{x > 5, x \leq 4\}\]
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 $\neg C$
- If $M$ is T-sat, return an interpretation for variables in model of $M$
How are Decision Procedures Implemented in SMT?

\[ \text{Input} \rightarrow \text{SAT Solver} \rightarrow \text{(partial) models} \rightarrow \text{Theory Solver for } T \rightarrow \text{SAT, model} \rightarrow \text{UNSAT} \rightarrow \Rightarrow \text{DPLL}(T) \text{ procedure [Nieuwenhuis/Oliveras/Tinelli 2007]} \]
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 [Nieuwenhuis/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]
    ⇒ 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:

\[
\text{List} := \text{cons}( \text{head} : \text{Int}, \text{tail} : \text{List} ) \mid \text{nil}
\]

• Theory of Inductive Datatypes (DT) for Lists of Int
  • \( \Sigma_{\text{DT}} : \{ \text{cons, head, tail, nil} \} \)
  • Interpretations \( I_{\text{DT}} \) are such that:
    • Constructors are distinct... \( \text{cons}(x,y) \neq \text{nil} \)
    • Constructors are injective... if \( \text{cons}(x_1, y_1) = \text{cons}(x_2, y_2) \), 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( \text{cons}(x, y) ) = x
    • Terms do not contain themselves as subterms... \( y \neq \text{cons}(x, y) \)

• **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:

\[
\text{Stream} := \text{cons}(\text{head} : \text{Int}, \text{tail} : \text{Stream})
\]

• Stream is not well-founded
  \(\Rightarrow\) \textit{Decision procedure for inductive datatypes does not apply}

• Instead, need decision procedure for \textit{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 \textit{in SMT solvers}
Theory of (Co)Inductive Datatypes

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

\[
\begin{align*}
\frac{t \in T(E)}{E := E, t \approx t} & \quad \text{Refl} \\
\frac{t \approx u \in E}{E := E, u \approx t} & \quad \text{Sym} \\
\frac{s \approx t, t \approx u \in E}{E := E, s \approx u} & \quad \text{Trans}
\end{align*}
\]

\[
\begin{align*}
\frac{\bar{t} \approx \bar{u} \in E \quad t(\bar{t}), t(\bar{u}) \in T(E)}{E := E, t(\bar{t}) \approx t(\bar{u})} & \quad \text{Cong} \\
\frac{C(\bar{t}) \approx C(\bar{u}) \in E}{E := E, \bar{t} \approx \bar{u}} & \quad \text{Inject} \\
\frac{C(\bar{t}) \approx D(\bar{u}) \in E \quad C \neq D}{E := E, \bar{t} \approx \bar{u}} & \quad \text{Clash}
\end{align*}
\]

\[
\begin{align*}
\frac{\delta \in \mathcal{Y}_{\text{dt}} \quad \mathcal{A}[t^\delta] = \mu x. u \quad x \in \text{FV}(u)}{E := E, t \approx u} & \quad \text{Acyclic}
\end{align*}
\]

• For codatatypes:
  • Terms can contain themselves as subterms: \(x = \text{cons}(z, x)\) is satisfiable
  • Terms are unique up to \(\alpha\)-equivalence:
    • If \(x = \text{cons}(z, x)\) and \(y = \text{cons}(z, y)\), then \(x = y\)

[Reynolds/Blanchette CADE15]
Theory of (Co)Inductive Datatypes

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

<table>
<thead>
<tr>
<th></th>
<th>Distro</th>
<th></th>
<th></th>
<th>G&amp;L</th>
<th></th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CVC4</td>
<td>Z3</td>
<td>CVC4</td>
<td>Z3</td>
<td>CVC4</td>
<td>Z3</td>
</tr>
<tr>
<td>No (co)datatypes</td>
<td>221</td>
<td>209</td>
<td>775</td>
<td>777</td>
<td>52</td>
<td>51</td>
</tr>
<tr>
<td>Datatypes without Acyclic</td>
<td>227</td>
<td>–</td>
<td>780</td>
<td>–</td>
<td>52</td>
<td>–</td>
</tr>
<tr>
<td>Full datatypes</td>
<td>227</td>
<td>213</td>
<td>786</td>
<td>791</td>
<td>52</td>
<td>51</td>
</tr>
<tr>
<td>Codatatypes without Unique</td>
<td>222</td>
<td>–</td>
<td>804</td>
<td>–</td>
<td>56</td>
<td>–</td>
</tr>
<tr>
<td>Full codatatypes</td>
<td>223</td>
<td>–</td>
<td>804</td>
<td>–</td>
<td><strong>59</strong></td>
<td>–</td>
</tr>
<tr>
<td>Full (co)datatypes</td>
<td><strong>229</strong></td>
<td>–</td>
<td><strong>815</strong></td>
<td>–</td>
<td><strong>59</strong></td>
<td>–</td>
</tr>
</tbody>
</table>

[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
    ⇒ Theory of Strings + Length constraints [Liang/Reynolds/Tinelli/Barrett/Deters CAV14]
Theory of Strings + Length
Theory of Strings : Applications

- Security applications frequently rely on reasoning about string constraints
Theory of Strings + Length

• Signature $\Sigma_s$:
  • Constants from a fixed finite alphabet $A^* = (a, ab, cbc...)$
  • String concatenation $\_ \cdot \_ : \text{String} \times \text{String} \rightarrow \text{String}$
  • Length terms $\text{len}(\_) : \text{String} \rightarrow \text{Int}$

• Example input:

$$\text{len}(x) > \text{len}(y) \land x \cdot b = y \cdot ab$$
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
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

[Theory of Strings : Rule-Based Procedure

F-Unify \[ F_s = (w, u, u_1) \quad F_t = (w, v, v_1) \quad s \approx t \in C(S) \quad S \models \text{len} u \approx \text{len} v \]
\[ S := S, u \approx v \]

F-Split \[ F_s = (w, u, u_1) \quad F_t = (w, v, v_1) \quad s \approx t \in C(S) \quad S \models \text{len} u \not\approx \text{len} v \]
\[ u \notin \mathcal{V}(v_1) \quad v \notin \mathcal{V}(u_1) \]
\[ S := S, u \approx \text{con}(v, z) \quad || \quad S := S, v \approx \text{con}(u, z) \]

F-Loop \[ F_s = (w, x, u_1) \quad F_t = (w, v, v_1, x, v_2) \quad s \approx t \in C(S) \quad x \notin \mathcal{V}((v, v_1)) \]
\[ S := S, x \approx \text{con}(z_2, z) \quad \text{con}(v, v_1) \approx \text{con}(z_2, z_1) \quad \text{con}(u_1) \approx \text{con}(z_1, z_2, v_2) \]
\[ R := R, z \quad \text{in star(set con}(z_1, z_2)) \quad C := C, t \]
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]
• 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

Theory of Strings : Experimental Results

[Liag/Reynolds/Tinelli/Barrett/Deters CAV14]
Extending the Theory of Strings

• Theory of strings can be extended with support for:
  • Regular expressions
    • E.g. \( x \in (a \cup (bb) \ast) \ast \)
    • Decision procedure for regular memberships + length [submitted, FroCos15]
  • Regular languages
    • E.g. \( x \in (y \cdot b) \ast \)
  • 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. $\forall x. f(x) > 0, \forall x. select(A,x) = 2*x$
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]
    - 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 $\forall$

Diagram:

- SMT solver
- Ground solver
- SAT Solver
- Theory Solver for T
- Quantifiers Module
- DPLL(T)
SMT Solver + Quantified Formulas

• For input $f(a) > 0 \land \forall x. f(x) < 0$
  
  • **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

\[ f(a) > 0 \]

\[ \forall x. f(x) < 0 \]

Ground solver

SAT Solver

Theory Solver for T

Quantifiers Module

DPLL(T)
SMT Solver + Quantified Formulas

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

Sat

Theory Solver for T

DPLL(T)

Quantifiers Module

∀x . f(x) < 0

Δ (a) > 0

• 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

\[ f(a) > 0, f(a) < 0, f(b) < 0, \ldots \]

Ground solver

SAT Solver

Theory Solver for T

DPLL(T)

Quantifiers Module

\[ \forall x. f(x) < 0 \]

UNSAT

- 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 $\forall x.Q[x]$?)

\[ G, Q[t_1], Q[t_2], ... \]

Ground solver

- SAT Solver
- Theory Solver for T

Quantifiers Module

- `\forall x. Q[x]`

Instances of $Q$
How SMT Solvers Handle Quantified Formulas

- SAT Solver
- Ground solver
- DPLL(T)
- $G, Q[t_1], Q[t_2], …$
- Quantifiers Module
- $\forall x. Q[x]$
- Instances of $Q$
- Theory Solver for $T$
- $\text{sat}$
- $\Rightarrow$ 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 null 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])( 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)
Finite Model Finding: Application

Verification conditions translated into (multiple) SMT queries, requiring:

- Theories (arithmetic, bit vectors, datatypes, ...)
- Quantified formulas for stating universal properties over:
  - Memory addresses, resources, processes, ...

Verification Conditions

\[
\text{goal main = invariant prop assuming refs\_non\_zero}
\]

\[
\text{goal mz = formula [... \& prop \& ... \Rightarrow refs\_non\_zero]}
\]

\[
\text{def bool prop = forall (process p) (ref[p] \neq null \Rightarrow valid[ref[p]])}
\]

\[
\text{def bool refs\_non\_zero = forall (process p) (ref[p] \neq null \Rightarrow count[ref[p]] > 0)}
\]

\[
\text{transition create (resource r)}
\]

\[
\text{require r \neq null, valid[r]}
\]

\[
\text{valid[r] := true;}
\]

\[
\text{count[r] := 0;}
\]

\[
\text{module S = Set<type process>}
\]

to SMT solver...
Why are Models Important?

Verification Condition for P

with quantifiers

SMT solver

UNSAT

Unknown

Candidate Model

Property P is verified

Manual Inspection
Why are Models Important?

Verification Condition for P

SMT solver

with quantifiers

UNSAT

Candidate Model

Concrete counterexample for Property P

SAT

Manual Inspection

Property P is verified
Finite Model Finding in SMT

\[ \forall x y : S . Q(x, y) \]

Diagram:
- Ground Solver
- Quantifiers Module

\( G \)
Finite Model Finding in SMT

\[ \forall xy : S \cdot Q(x, y) \]

⇒ If \( S \) has finite interpretation, 
• use finite model finding
Finite Model Finding in SMT

\( \forall x y : S. Q(x, y) \)

\( S = \{a, b, c, d, e\} \)
Finite Model Finding in SMT

\[ G \equiv \forall xy : S . Q(x, y) \]

\[ S = \{a, b, c, d, e\} \]

- Reduction of quantified formulas to ground formulas
Finite Model Finding in SMT

\[ \forall xy: S . \neg Q(x, y) \]

\[ G \wedge Q(a, a) \wedge \ldots \wedge Q(e, a) \wedge \]
\[ Q(a, b) \wedge \ldots \]
\[ Q(a, c) \wedge \ldots \]
\[ Q(a, d) \wedge \ldots \]
\[ Q(a, e) \wedge \ldots \wedge Q(e, e) \]

\[ S = \{ a, b, c, d, e \} \]

\[ \Rightarrow \text{Ability to answer SAT, assuming decision procedure for } G \wedge Q(a, a) \wedge \ldots \]
Finite Model Finding in SMT

\[ \forall x y : S. Q(x, y) \]

\[ S = \{a, b, c, d, e\} \]

- Can be very large

\[ \land Q(a, a) \land \ldots \land Q(e, a) \land Q(a, b) \land \ldots \land Q(a, c) \land \ldots \land Q(a, d) \land \ldots \land Q(a, e) \land \ldots \land Q(e, e) \]
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**

Search for models where $|S| = 1$

If none exist, search for models where $|S| = 2$

etc.

[Reynolds/Tinelli/Goel/Krstic CAV13]
Finite Model Finding : Minimizing Model Sizes

• Minimize model sizes using a theory solver for cardinality constraints

\[ |S| \leq 1, \quad \neg |S| \leq 1 \]

Search for models where \(|S|=1\)

\[ |S| \leq 2, \quad \neg |S| \leq 2 \]

If none exist, search for models where \(|S|=2\)

\[ |S| \leq 3, \quad \neg |S| \leq 3 \]

\(\Rightarrow\) 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 $M$

$G \quad \forall xy: S \cdot Q(x, y)$

- Ground Solver
- Quantifiers Module

$S = \{a, b, c, d, e\}$
$\{Q \rightarrow Q^M, \ldots\}$

[Reynolds/Tinelli/Goel/Krstic/Barrett/Deters CADE13]
Finite Model Finding: Model-Based Instantiation

\[ \forall x y : S . Q(x, y) \]

- Evaluate quantified formulas based on \( Q^M \)

\[ S = \{a, b, c, d, e\} \]
\[ \{Q \rightarrow Q^M, \ldots\} \]

[Reynolds/Tinelli/Goel/Krštic/Barrett/Deters CADE13]
Finite Model Finding: Model-Based Instantiation

- Only add instances that evaluate to F in $Q^M$
  - Significantly increased scalability

$G$^\land $Q(e,e)$
$Q(b,a)$
$orall xy : S . Q(x,y)$

$S = \{a, b, c, d, e\}$
$Q \rightarrow Q^M, ...$

[Reynolds/Tinelli/Goel/Krstic/Barrett/Deters CADE13]
### Results: Hardware Verification at Intel

#### 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**

<table>
<thead>
<tr>
<th>SAT</th>
<th>german</th>
<th>refcount</th>
<th>agree</th>
<th>apg</th>
<th>bmk</th>
<th>Total</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>45</td>
<td>6</td>
<td>42</td>
<td>19</td>
<td>37</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>z3</td>
<td>45</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>46</td>
<td>8.1</td>
</tr>
<tr>
<td>cvc4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>cvc4+f</td>
<td>45</td>
<td>6</td>
<td>42</td>
<td>19</td>
<td>37</td>
<td>149</td>
<td>409.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNSAT</th>
<th>german</th>
<th>refcount</th>
<th>agree</th>
<th>apg</th>
<th>bmk</th>
<th>Total</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>145</td>
<td>40</td>
<td>488</td>
<td>304</td>
<td>244</td>
<td>1221</td>
<td></td>
</tr>
<tr>
<td>z3</td>
<td>145</td>
<td>40</td>
<td>488</td>
<td>304</td>
<td>244</td>
<td>1221</td>
<td>31.0</td>
</tr>
<tr>
<td>cvc4</td>
<td>145</td>
<td>40</td>
<td>484</td>
<td>304</td>
<td>244</td>
<td>1217</td>
<td>21.3</td>
</tr>
<tr>
<td>cvc4+f</td>
<td>145</td>
<td>40</td>
<td>488</td>
<td>302</td>
<td>244</td>
<td>1219</td>
<td>1185.0</td>
</tr>
</tbody>
</table>

**cvc4:**
- $f$: finite model finding
### Results: CASC Competition

- 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

<table>
<thead>
<tr>
<th>First-order Non-theorems</th>
<th>iProver 1.0-SAT</th>
<th>Paradox 3.0</th>
<th>CVC4 1.2-SAT</th>
<th>E 1.8</th>
<th>Nitrox 2013</th>
<th>Vampire 3.0-SAT</th>
<th>E-KRHyL 1.4</th>
<th>iProver-E 0.85</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solved/150</td>
<td>122/150</td>
<td>99/150</td>
<td>96/150</td>
<td>79/150</td>
<td>79/150</td>
<td>78/150</td>
<td>67/150</td>
<td>37/150</td>
</tr>
<tr>
<td>Av. CPU Time</td>
<td>52.47</td>
<td>2.28</td>
<td>25.94</td>
<td>20.94</td>
<td>29.70</td>
<td>15.89</td>
<td>7.57</td>
<td>30.77</td>
</tr>
<tr>
<td>Solutions</td>
<td>122/150</td>
<td>99/150</td>
<td>96/150</td>
<td>79/150</td>
<td>79/150</td>
<td>78/150</td>
<td>67/150</td>
<td>0/150</td>
</tr>
<tr>
<td>μEfficiency</td>
<td>165</td>
<td>549</td>
<td>204</td>
<td>396</td>
<td>36</td>
<td>395</td>
<td>292</td>
<td>92</td>
</tr>
<tr>
<td>SOTAC</td>
<td>0.28</td>
<td>0.29</td>
<td>0.19</td>
<td>0.22</td>
<td>0.24</td>
<td>0.20</td>
<td>0.19</td>
<td>0.15</td>
</tr>
<tr>
<td>New Solved</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>0/4</td>
<td>0/4</td>
</tr>
</tbody>
</table>
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

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

∀ 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

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';
    }
} else {
    cout << "Wrong Password";
}
if (pass == 'Y') /* Grant the root permission*/

Does Property P hold for my program?

⇒ Increased support for applications

Increased complexity
• Expressive theories
• Quantified Formulas

Increased ability
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?