Synthesis and Satisfiability Modulo Theory Solvers

Viktor Kuncak
EPFL
Laboratory for Automated Reasoning and Analysis

http://lara.epfl.ch  http://leon.epfl.ch
http://cvc4.cs.nyu.edu/web/
How to automatically verify that an implementation meets a spec?
How to automatically transform a spec into an implementation?
Given a list of numbers, make this list sorted.

```python
def sort_spec(input : List, output : List) : Boolean = 
    content(output)==content(input)   &&
    isSorted(output)
```

**Specification** (for us) is a *program* that checks, for a given *input*, whether the given *output* is acceptable.
Specification vs Implementation

**def** `C(i : List, o : List) : Boolean` =

```python
content(o) == content(i) && isSorted(o)
```

**Implementation**

**true / false**

**specification**

```
8900  
6000  
24140 
2900  
```

**U**

```
2900  
6000  
8900  
24140 
```

**More behaviors**

**Constraint on the output**

```
8900  
6000  
24140 
2900  
```

```
2900  
6000  
8900  
24140 
```

**Fewer behaviors**

**def** `p(i : List) : List` =

```python
sort i using a sorting algorithm
```

**Function that computes the output**

**Equality constraint on output**

```
p \subseteq C
```
- hundreds of thousands of Scala programmers, used in:
  Twitter, Foursquare, Coursera, The Guardian, New York Times, Huffington Post, UBS, LinkedIn, Meetup, Verizon, Intel, ...

**Typesafe** Inc. supports Scala commercially

**EPFL**: industrial advisory board, courses, open source development

**Chisel**: “…an open-source hardware construction language developed at UC Berkeley that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages.” – other Scala DSLs: e.g. OptiML

**Apache Spark**: “an open-source cluster computing framework with in-memory processing to speed analytic applications up to 100 times faster compared to technologies on the market today. Developed in the AMPLab at UC Berkeley, Apache Spark can help reduce data interaction complexity, increase processing speed and enhance mission-critical applications with deep intelligence.”

“...IBM is making a major commitment to the future of **Apache Spark**, with a series of initiatives announced today. IBM will offer Apache Spark as a service on **Bluemix**; commit 3,500 researchers to work on Spark-related projects; donate IBM SystemML to the Spark ecosystem; and offer courses to train 1 million data scientists and engineers to use Spark.”
a) Check assertion while program \( p \) runs: \( C(i, p(i)) \)

b) Verify whether program always meets the spec:
\[
\forall i. \ C(i, p(i))
\]

c) Constraint programming: once \( i \) is known, find \( o \) to satisfy a given constraint: find \( o \) such that \( C(i, o) \)

d) Synthesis: solve \( C \) symbolically to obtain program \( p \) that is correct by construction, for all inputs: find \( p \) such that
\[
\forall i. C(i, p(i)) \quad \text{i.e.} \quad p \subseteq C
\]

http://leon.epfl.ch
Approaches and Their Guarantees

both specification $C$ and program $p$ are given:

a) **Check assertion** while program $p$ runs: $C(i, p(i))$

b) **Verify** that program always meets spec: $\forall i. C(i, p(i))$

only specification $C$ is given:

c) **Constraint programming:** once $i$ is known, find $o$ to satisfy a given constraint: find $o$ such that $C(i, o)$

run-time

d) **Synthesis:** solve $C$ symbolically to obtain program $p$ that is correct by construction, for all inputs: find $p$ such that $\forall i. C(i, p(i))$ i.e. $p \subseteq C$

compile-time
Synthesizing Sort in Leon System

```python
def insertSorted(lst : List, v : Int): List = {
    require(isSorted(lst))
    choose { (r: List) ⇒
        isSorted(r) && content(r) == content(lst) ++ Set(v) } }

def sort(lst : List): List = choose { (r: List) ⇒
    isSorted(r) && content(r) == content(lst) }
```

http://leon.epfl.ch

OOPSLA 2013:
Synthesis Modulo Recursive Functions

Etienne Kneuss  Ivan Kuraj  Philippe Suter
Recursion Schemas + STE in Action

```scala
def delete(in1: List, v: Int) = choose {
  (out: List) => content(out) == content(in1) -- Set(v)
}

def delete(in1: List, v: Int) = {
  def rec(in: List, v: Int): List = in match {
    case Cons(h,t) =>
      val r = rec(t,v)
      if (h == v) {
        CEGIS r
      } else {
        CEGIS Cons(h, r)
      }
    case Nil =>
      CEGIS Nil
  }
  rec(in1, v)
}
```

ADT Induction

EQ Split
Decomposition Example: Case Split

\[
\langle P_1 \mid T_1 \rangle \quad \text{Case Split} \quad \langle P_2 \mid T_2 \rangle
\]

\[
\llbracket \bar{a} \langle \varphi_1 \lor \varphi_2 \rangle \overline{x} \rrbracket
\]

\[
\llbracket \bar{a} \langle \varphi_1 \rangle \overline{x} \rrbracket \quad \text{Case Split} \quad \llbracket \bar{a} \langle \varphi_2 \rangle \overline{x} \rrbracket
\]

\[
\langle P_1 \lor P_2 \mid \text{if}(P_1) T_1 \text{ else } T_2 \rangle
\]
Symbolic Term Exploration (STE)

Symbolic search over many expressions of bounded size

\[ T(\bar{b}) = \text{if } (b_0) a_0 \]

\[ \quad \text{elseif } (b_1) \text{ Nil} \]

\[ \quad \text{elseif } (b_2) \text{ Cons}(0, \text{Nil}) \]

\[ \quad \text{elseif } (b_3) \ldots \]

SMT solver searches exponentially many expressions given by polynomially many Boolean variables

Concrete execution prunes search space

Leon’s verifier validates candidate expressions

– using again SMT solvers
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
  choose((h: Int, m: Int, s: Int) ⇒ (  
    h * 3600 + m * 60 + s == totalSeconds    
    && h ≥ 0    
    && m ≥ 0 && m < 60    
    && s ≥ 0 && s < 60    
  ))
Implementation

V1) Quantifier elimination procedure for Presburger arith.
   – optimization for some important cases
   – inefficient in general

V2) Automata-based procedure for int. arithmetic with bitwise operations (Hamza, Jobstmann, K., FMCAD’10)
   – handles larger subset
   – bad at e.g. multiplication by large constants - we do not know good general techniques for sequential circuits

V3) Inside an SMT solver: Andrew Reynolds, Morgan Deters, Cesare Tinelli, Clark Barrett, K. – CAV’15
   ➔ focus today
Synthesis Problem for an SMT Solver

• Synthesis Problem: \( \exists f. \forall x. P(f, x) \)

There exists a function \( f \) such that for all \( x \), \( P(f, x) \)

• Most existing approaches for synthesis
  • Rely on specialized solver that makes subcalls to an SMT Solver
• Goal: approach implemented entirely inside SMT solver
SMT Solver + Quantified Formulas

SMT solver consists of:
- **Ground solver** maintains a set of ground (variable-free) constraints
- **Quantifiers Module** maintains a set of quantified formulas: $\forall x. P(x)$

Using SMT solvers: game-changer in automated software verification

Increasingly relevant in industry:
- symbolic execution of systems code, microcode, word-level reasoning
- Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, Jonathan Shalev: **Applying SMT in symbolic execution of microcode.** *FMCAD 2010*: 121-128 – Best Paper Award
SMT Solver + Quantified Formulas

\[ \forall x. P(x) \]

\[ P(a), P(b), P(c), \ldots \]

Ground solver

SAT Solver

Decision Procedure for \( T_i \)

\text{instances}

unsat?

• Goal: add instances of axioms until ground solver can answer “unsat”
Running Example: Max of Two Integers

\[
\exists f. \forall xy. (f(x, y) \geq x \land f(x, y) \geq y \land (f(x, y) = x \lor f(x, y) = y))
\]

- Specifies that \( f \) computes the maximum of integers \( x \) and \( y \)
- A solution:

\[
f := \lambda xy. \text{ite}(x \geq y, x, y)
\]
Approach: Refutation-Based Synthesis

\[ \neg \exists f. \forall x. P(f, x) \]

• What if we negate the synthesis conjecture?
• If we are in a satisfaction-complete theory T (e.g. LIA, BV):
  • \( F \) is T-satisfiable if and only if \( \neg F \) is T-unsatisfiable

⇒ Will suffice for us to show the above formula is \text{unsat}
Challenge: Second-Order Quantification

\[ \neg \exists f. \forall x. P(f, x) \]

\[ \forall f. \exists x. \neg P(f, x) \]

- Challenge: negation introduces universal \( \forall \) over function \( f \)
  - No SMT solvers directly support second-order quantification
Challenge: Second-Order Quantification

\[
\neg \exists f. \forall x. P(f, x)
\]

\[
\forall f. \exists x. \neg P(f, x)
\]

• Challenge: negation introduces universal \( \forall \) over function \( f \)
  
  • No SMT solvers directly support second-order quantification

• However, we can avoid this quantification using two approaches:
  
  1. When property \( P \) is single invocation for \( f \)  ➞ focus now
  
  2. When \( f \) is given syntactic restrictions
Single Invocation Properties

\[ \forall f. \exists x y. (f(x, y) \neq x \land f(x, y) \neq y) \]

\[ (f(x, y) < x \lor f(x, y) < y) \lor \]

\[ (f(x, y) = x) \lor f(x, y) = y) \]
Single Invocation Properties

\[ \forall f. \exists x y. (f(x, y) < x \lor f(x, y) < y \lor (f(x, y) \neq x \land f(x, y) \neq y)) \]

- **Single invocation** properties
  - Are properties such that:
    - All occurrences of \( f \) are of a particular form, e.g. \( f(x, y) \) above
  - Are a common class of properties useful for:
    - Software Synthesis (post-conditions describing the result of a function)
  - Given solution, it can be checked without replicating solution

**NOT** single invocation: “f is commutative”
Single Invocation Properties

\[ \forall f. \exists x y. ( f(x, y) < x \lor f(x, y) < y \lor (f(x, y) \neq x \land f(x, y) \neq y) ) \]

Push \( \forall \) quantification inwards

\[ \exists x y. \forall g. ( g < x \lor g < y \lor (g \neq x \land g \neq y) ) \]

- Occurrences of \( f(x, y) \) are replaced with integer variable \( g \)
- Resulting formula is equisatisfiable, and \textbf{first-order}
Single Invocation Properties

\[ \forall f. \exists x y. (f(x, y) < x \lor f(x, y) < y \lor (f(x, y) \neq x \land f(x, y) \neq y)) \]

\[ \exists x y. \forall g. (g < x \lor g < y \lor (g \neq x \land g \neq y)) \]

Skolemize, for fresh \( a \) and \( b \)

\[ \forall g. (g < a \lor g < b \lor (g \neq a \land g \neq b)) \]
Solving Max Example

\[ \forall g. (g < a \lor g < b \lor (g \neq a \land g \neq b)) \]

Ground solver

Quantifiers Module
Solving Max Example

∀g. ¬isMax(g,a,b)
Solving Max Example

\[ \neg \text{isMax}(a, a, b) \land \neg \text{isMax}(b, a, b) \]

Ground solver

\[ \forall g. \neg \text{isMax}(g, a, b) \]

Quantifiers Module

instances \( a/g, b/g \)
Solving Max Example

\[ a < b \land b < a \]

\[ \forall g. \neg \text{isMax}(g, a, b) \]

- **simplify**
  - a < b \land b < a
    - Ground solver
  - \forall g. \neg \text{isMax}(g, a, b)
    - Quantifiers Module
Solving Max Example

\[ a < b \land b < a \]

Ground solver

unsat \[ \Rightarrow \forall g. \neg \text{isMax}(g, a, b) \text{ is unsatisfiable} \]
by instances \( a/g, b/g, \)
implies original synthesis conjecture has a solution
Solving Max Example

Ground solver

\[\neg \text{isMax}(a, a, b) \land \neg \text{isMax}(b, a, b)\]

\[\exists f. \forall xy. \text{isMax}(f(x, y), x, y)\]

\[\forall g. \neg \text{isMax}(g, a, b)\]

\[f := \lambda xy. \text{ite}(\text{isMax}(a, a, b), a, b)[x/a][y/b]\]

\[\Rightarrow \text{Extract solution from unsatisfiable core of instantiations } a/g, b/g\]
Solving Max Example

\[ \exists f. \forall xy. \text{isMax}(f(x,y), x, y) \]

\[ \neg \text{isMax}(a, a, b) \land \neg \text{isMax}(b, a, b) \]

\[ \forall g. \neg \text{isMax}(g, a, b) \]

\[ f := \lambda xy. \text{ite}(x \geq y, x, y) \]

\[ \Rightarrow \text{Desired function, after simplification} \]
How do we Choose Relevant Instances?

... 

\[ \forall g. \neg \text{isMax}(g,a,b) \]

Ground solver

Quantifiers Module

?
Counterexample-Guided Quantifier Instantiation

• Instances chosen using *counterexample-guided quantifier instantiation*
  ⇒ Follows counterexample-guided inductive synthesis (CEGIS) approach

In work under submission, we provide framework where such CEGIS choices are **complete** for linear arithmetic

**Much better scalability than quantifier elimination approaches**
Quantifiers Module of CVC4

Specialized technique for quantified arithmetic

ARXiv:1510.02642
An Instantiation-Based Approach for Solving Quantified Linear Arithmetic
Andrew Reynolds, Tim King, Viktor Kuncak
CVC4 in Sygus Competition 2015

NSF Expedition on Computer Augmented Programm Engineering (ExCAPE) led by Rajeev Alur, involves UPenn, MIT, Berkeley, Rice, Illinois, Maryland, UCLA, Michigan organizes a competition of software synthesis tools

https://excape.cis.upenn.edu/

Our technique Won General and LIA tracks (= 2/3 tracks) in competition

<table>
<thead>
<tr>
<th>Solver</th>
<th>#solved</th>
<th>total-expr-size</th>
<th>average-expr-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVC4-1.5-syguscomp2015-v4</td>
<td>70</td>
<td>43726</td>
<td>624.66</td>
</tr>
<tr>
<td>AlchemistCSDT</td>
<td>47</td>
<td>6658</td>
<td>141.66</td>
</tr>
<tr>
<td>Alchemist CS</td>
<td>33</td>
<td>866</td>
<td>26.24</td>
</tr>
</tbody>
</table>

In LIA track, solved 70/73 benchmarks, 60 of these in <1 second
  • Nearest competitor AlchemistCSDT solved 47/73 in a timeout of 1 hour
Max example: Sygus Comp 2015

<table>
<thead>
<tr>
<th>$n$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvc4+si</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.6</td>
<td>1.0</td>
<td>1.9</td>
<td>3.2</td>
<td>5.3</td>
<td>6.5</td>
</tr>
<tr>
<td>AlchemistCSDT</td>
<td>0.2</td>
<td>0.6</td>
<td>1.5</td>
<td>6.4</td>
<td>20.8</td>
<td>132.8</td>
<td>877.9</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>AlchemistCS</td>
<td>0.0</td>
<td>3.7</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

- Outperforms existing approaches by an order of magnitude or more

⇒ Our approach is efficient for synthesizing non-recursive functions that are defined by cases

Implementation available in the main branch of CVC4 SMT solver:

http://cvc4.cs.nyu.edu/web/
Conclusions: Synthesis and SMT Solvers

Leon system for verifying and synthesizing Scala programs
http://leon.epfl.ch
SMT solvers are essential for synthesis and verification

Given support for quantifiers, SMT solver can perform synthesis on its own!

Key challenge: efficient techniques to instantiate quantifiers
  - CVC4 solution is state of the art for linear arithmetic: complete and fast

arXiv:1510.02642
An Instantiation-Based Approach for Solving Quantified Linear Arithmetic
Andrew Reynolds, Tim King, Viktor Kuncak