SMT Solvers for Verification and Synthesis

Andrew Reynolds
VTSA Summer School
August 1 and 3, 2017
Acknowledgements

• Thanks to past and present members of development team of CVC4:
  • Cesare Tinelli, Clark Barrett, Tim King, Morgan Deters, Dejan Jovanovic, Liana Hadarean, Kshitij Bansal, Tianyi Liang, Nestan Tsiskardidze, Christopher Conway, Francois Bobot, Guy Katz, Andres Noetzli, Paul Meng, Alain Mebsout, Burak Ekici

• ...and external collaborators:
  • Viktor Kuncak, Amit Goel, Sava Krstic, Leonardo de Moura, Jasmin Blanchette, Thomas Wies, Radu Iosif, Haniel Barbosa, Pascal Fontaine, Chantal Keller
Satisfiability Modulo Theories (SMT) Solvers

• SMT solvers are:
  • Fully automated reasoners
  • Widely used in applications
Satisfiability Modulo Theories (SMT) Solvers

- Software Verification Tools
- Interactive Proof Assistants
- Symbolic Execution Engines
- Synthesis Tools, Planners

Verification Conditions → Conjectures → Path Constraints → Specifications

Expressed in first-order logic formulas over some fixed background theory $T$
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 $x_{out} = y_{in} \land y_{out} = x_{in}$?

Software Verification Tools
Contract-Based Software Verification

@precondition: $x_{in} > y_{in}$

```c
void swap(int x, int y) {
    x := x + y;
    y := x - y;
    x := x - y;
}
```

...does this function ensure that $x_{out} = y_{in} \land y_{out} = x_{in}$?

Software Verification Tools

<table>
<thead>
<tr>
<th>Pre-condition</th>
<th>Function Body</th>
<th>(Negated) Post-condition</th>
</tr>
</thead>
</table>
| $x_{in} > y_{in}$ | $x_{2} = x_{in} + y_{in} \land y_{2} = y_{in}$ | $x_{out} = x_{3} - y_{3} \land y_{out} = y_{3}$  
| $x_{3} = x_{2} \land y_{3} = x_{2} - y_{2}$ | $(x_{out} \neq y_{in} \lor y_{out} \neq x_{in})$ |
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
$x_{out} = y_{in} \land y_{out} = x_{in}$

Software Verification Tools

$\text{Pre-condition}$

$\text{Function Body}$

$(\text{Negated})$

$\text{Post-condition}$

SMT Solver
Interactive Proof Assistants

Theorem app_rev:
\[ \forall (x : \text{list}) (y : \text{list}), \text{rev} \ \text{append} \ x \ y = \text{append} \ (\text{rev} \ y) \ (\text{rev} \ x) \].

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

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

\exists xy: L. rev(append(x, y)) \neq append(rev(y), rev(x))
Interactive Proof Assistants

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

\[
\begin{align*}
\text{case is-\text{cons} } x: & \text{ rev append } x \ y = \text{ by rev-def} \\
\text{...} \\
\text{case is-\text{nil} } x: & \\
\text{append } x \ y = y \text{ by append-def} \\
\text{rev } x = \text{ nil by rev-def} \\
\therefore \text{ rev append } x \ y = \text{ append (rev } y) (\text{rev } x) \text{ by simplify}
\end{align*}
\]

QED.

Interactive Proof Assistant

SMT Solver

Signature

Axioms

(Negated) conjecture
Symbolic execution

```cpp
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';
    }
} else if(pass == 'Y') {
    grant_root_permission();
    Assert(strcmp(buff, 'PASSWORD') == 0);
}
```

Does this assertion hold for all executions?
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';
    }
} else {
    cout << "Correct Password";
    pass = 'Y';
}
if(pass == 'Y') {
    grant_root_permission();
    Assert(strcmp(buff,"PASSWORD")==0);
}
Symbolic execution

```c++
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();
    }
} else {
    cout << "Wrong 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"))
```

SMT Solver

```
(define-fun input () String "AAAAAAAAAAAAAAAAAY")
(define-fun buff () String "AAAAAAAAAAAAAAAAA")
(define-fun pass () String "Y")
```
Symbolic execution

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

Symbolic Execution Engine

```
(define-fun input () String "AAAAAAAAAAAAAAAAAY")
(define-fun buff () String "AAAAAAAAAAAAAAAA")
(define-fun pass () String "Y")
```
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: ∀i. (c_{out}[i]≥a[i]∧c_{out}[i]≥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: ∀i. \( (c_{out}[i]>=a[i] \land c_{out}[i]>=b[i]) \) ?

Find an \( x \) that satisfies specification
\( x\geq a[i] \land x\geq b[i] \)

Is \( \text{ite}(a[i]\geq b[i],a[i],b[i]) \) a solution?

\( \neg(\text{ite}(a[i]\geq b[i],a[i],b[i])\geq a[i] \land \text{ite}(a[i]\geq b[i],a[i],b[i])\geq b[i]) \)
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: ∀i.(c_out[i]≥a[i]∧c_out[i]≥b[i])

SMT Solver
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: \( a \geq b + 2 \times c + 3 \)
    • Arrays: \( \text{select}(A, i) = \text{select}(\text{store}(A, i+1, 3), i) \)
    • Bit-vectors: \( \text{bvule}(x, \#\text{xFF}) \)
    • Algebraic Datatypes: \( x, y: \text{List}; \text{tail}(x) = \text{cons}(0, y) \)
    • Unbounded Strings: \( x, y: \text{String}; y = \text{substr}(x, 0, \text{len}(x) - 1) \)
    • ... 
  • \( \forall \) over each of these
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]
  • …
  • ∀ over each of these
Satisfiability Modulo Theories (SMT) Solvers

- Software Verification Tools
- Interactive Proof Assistants
- Symbolic Execution Engines
- Synthesis Tools, Planners

- Verification Conditions
- Conjectures
- Path Constraints
- Specifications

- SMT Solvers
Satisfiability Modulo Theories (SMT) Solvers

This lecture will focus mostly on this portion.
Overview

• Satisfiability Modulo Theories (SMT) solvers: how they work
  • DPLL, DPLL(T), decision procedures, Nelson-Oppen combination, quantifier instantiation

• How to use SMT solvers
  • smt2 language, models, proofs, unsat cores, incremental mode

• Things that SMT solvers can (and cannot) do well
Overview

• **Part 1**: DPLL and DPLL(T) for SAT (modulo theories)
  • Applications: Contract-based program verification, Symbolic Execution

• **Part 2**: Extension to quantified formulas $\forall \exists$
  • Applications: Inductive theorem proving, Finite Model finding, Synthesis
  • Use development version on right hand side

• ...or clone from github: [https://github.com/CVC4/CVC4](https://github.com/CVC4/CVC4)

• Lecture material available: