SMT Solvers for Verification and Synthesis

Andrew Reynolds

VTSA Summer School

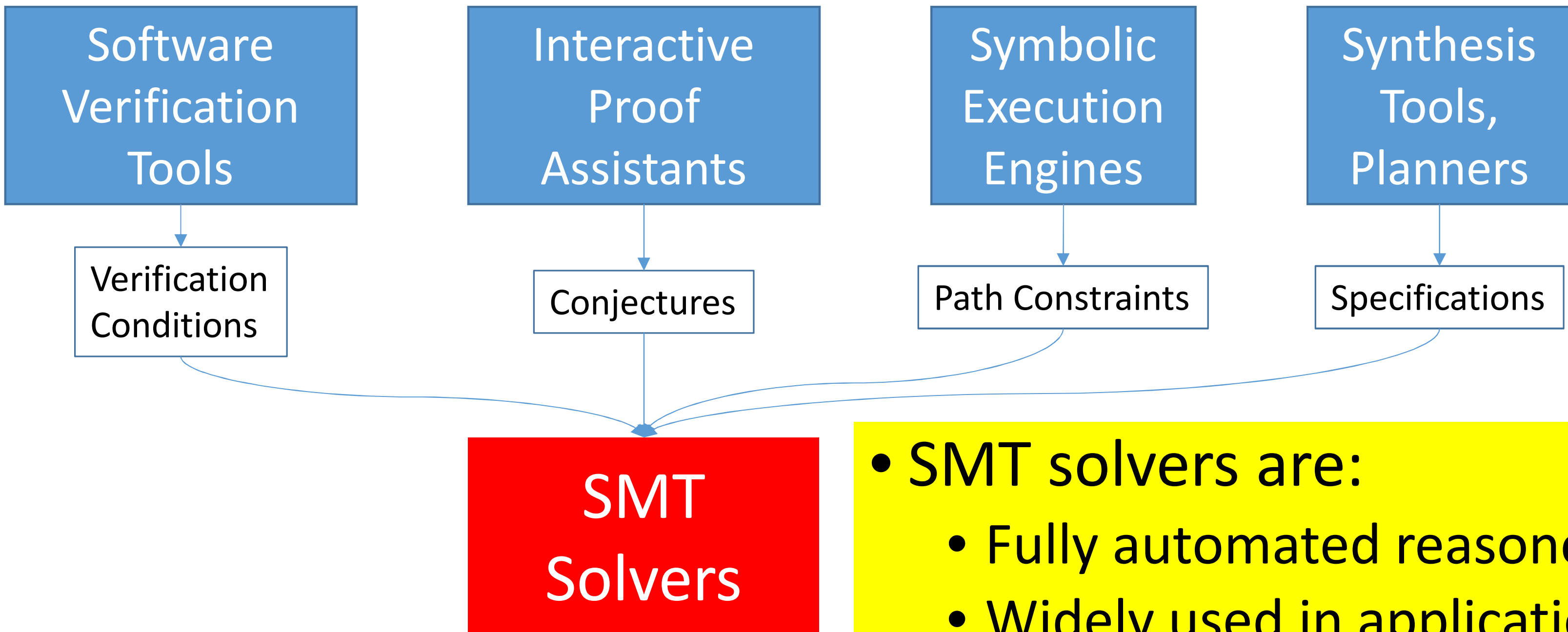
August 1 and 3, 2017

The University of Iowa

Acknowledgements

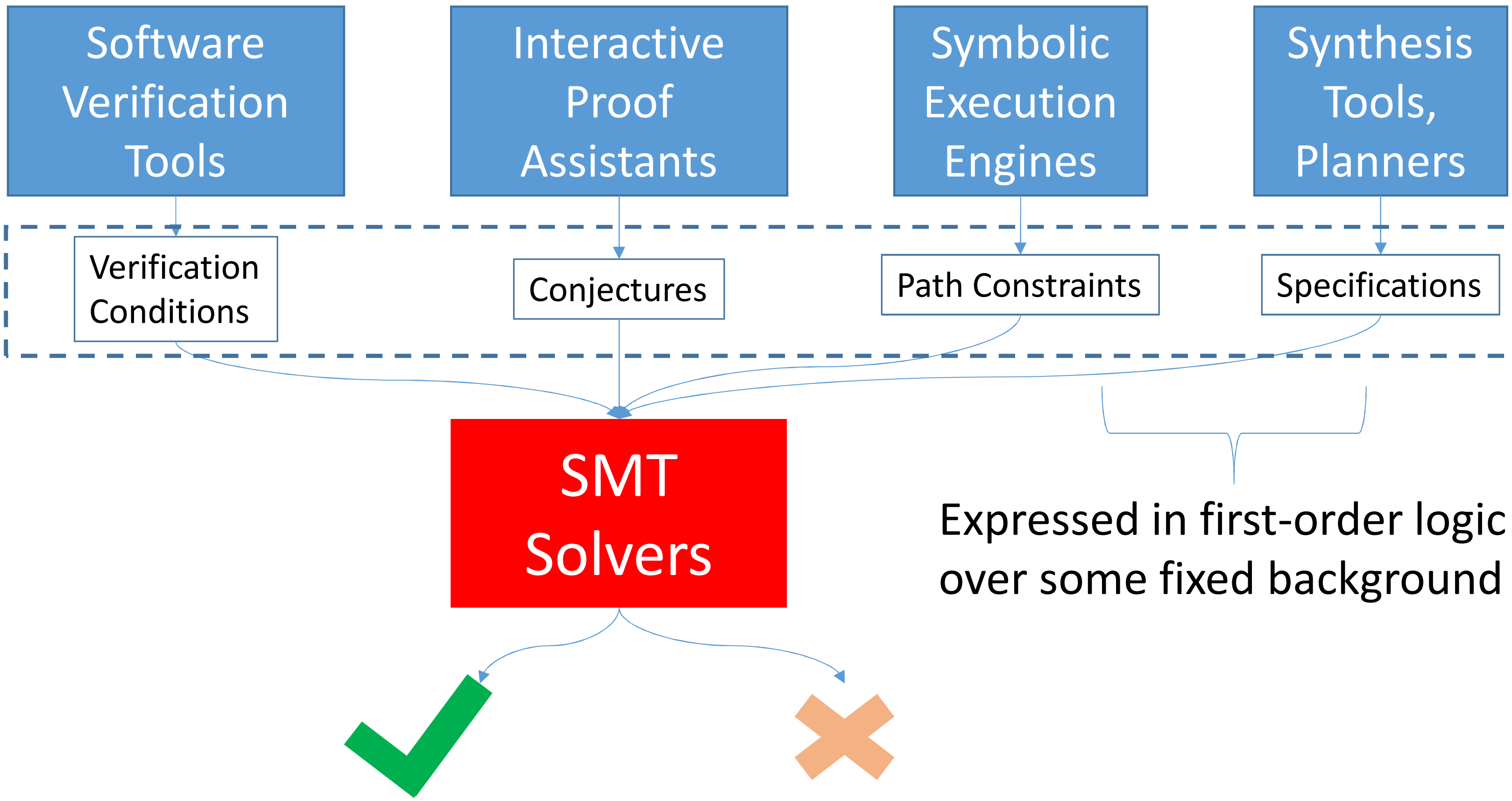
- Thanks to past and present members of development team of CVC4:
  - Cesare Tinelli, Clark Barrett, Tim King, Morgan Deters, Dejan Jovanovic, Liana Hadarean, Kshitij Bansal, Tianyi Liang, Nestan Tsiskardidze, Christopher Conway, Francois Bobot, Guy Katz, Andres Noetzli, Paul Meng, Alain Mebsout, Burak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



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 $x_{out}=y_{in} \wedge y_{out}=x_{in}$?

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 $x_{out}=y_{in} \wedge y_{out}=x_{in}$?

Software Verification Tools

Pre-condition

$$x_{in}>y_{in}$$

Function Body

$$x_2=x_{in}+y_{in} \;\wedge\; y_2=y_{in}$$

$$x_3=x_2 \;\wedge\; y_3=x_2-y_2$$

$$x_{out}=x_3-y_3 \;\wedge\; y_{out}=y_3$$

(Negated) Post-condition

$$(x_{out} \neq y_{in} \vee y_{out} \neq x_{in})$$

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

$x_{out}=y_{in}$ ė $y_{out}=x_{in}$

Software Verification Tools

$x_{in}>y_{in}$ — **Pre-condition**

$x_2=x_{in}+y_{in} \wedge y_2=y_{in}$

$x_3=x_2 \wedge y_3=x_2-y_2$

$x_{out}=x_3-y_3 \wedge y_{out}=y_3$ — **Function Body**

$(x_{out}\neq y_{in} \vee y_{out}\neq x_{in})$ — **(Negated) Post-condition**

SMT Solver

✓

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 Assistant

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 Assistant

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

Signature

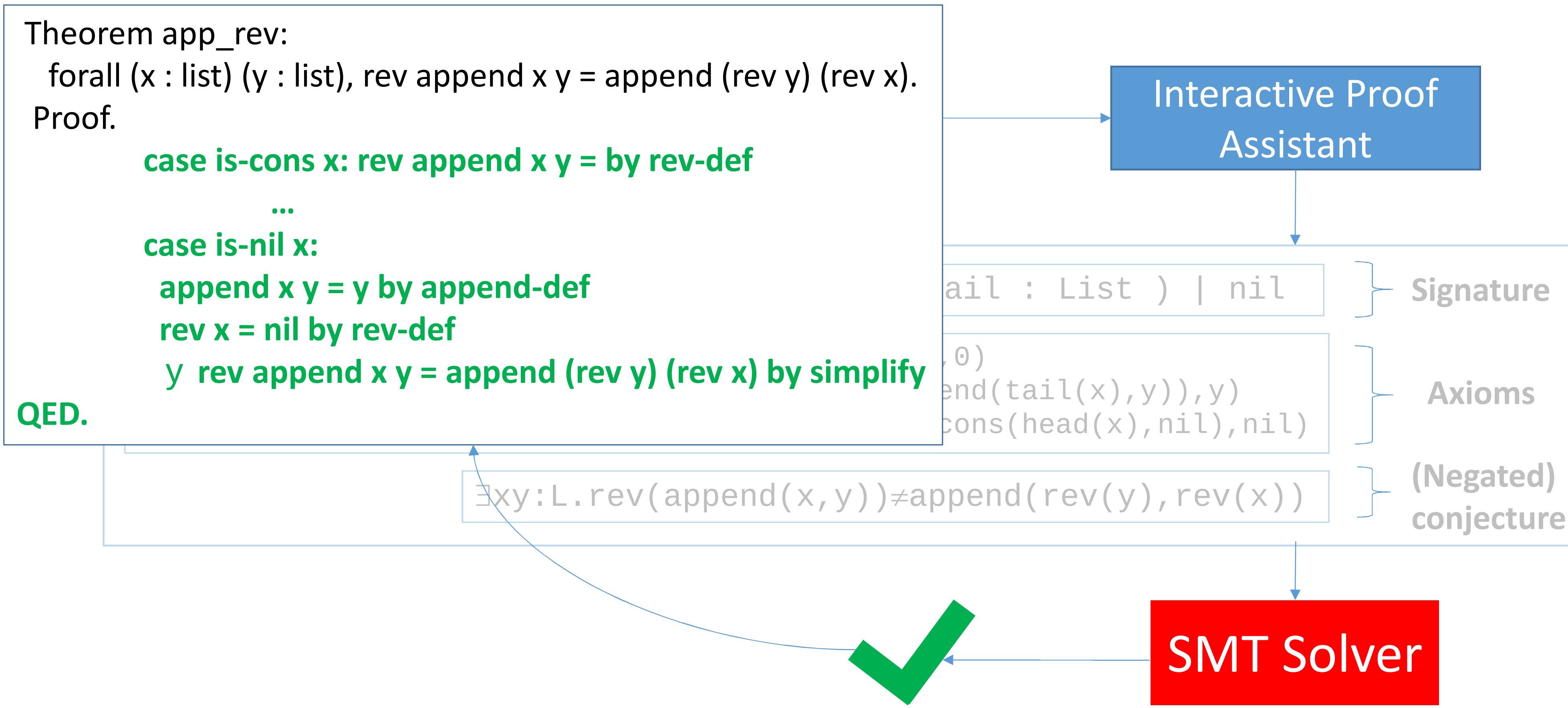
```
∀x:L.length(x)=ite(is-cons(x),1+length(tail(x)),0)
∀xy:L.append(x)=ite(is-cons(x),cons(head(x),append(tail(x),y)),y)
∀x:L.rev(x)=ite(is-cons(x),append(rev(tail(x)),cons(head(x),nil),nil)
```

Axioms

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

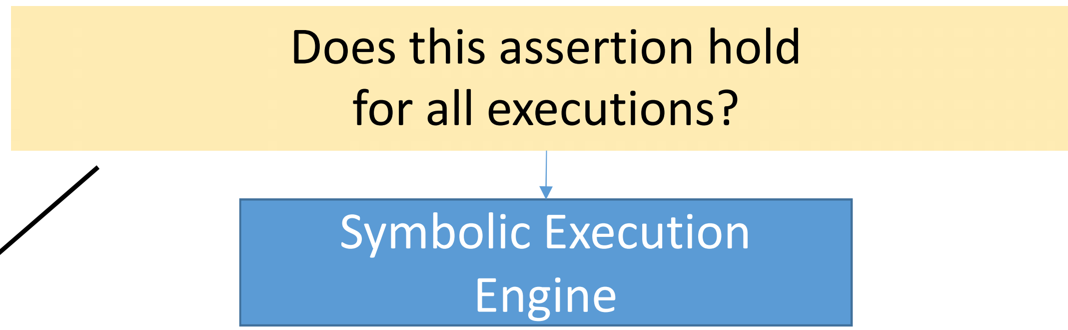
(Negated) conjecture

Interactive Proof Assistants



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);
    }
}
```



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?

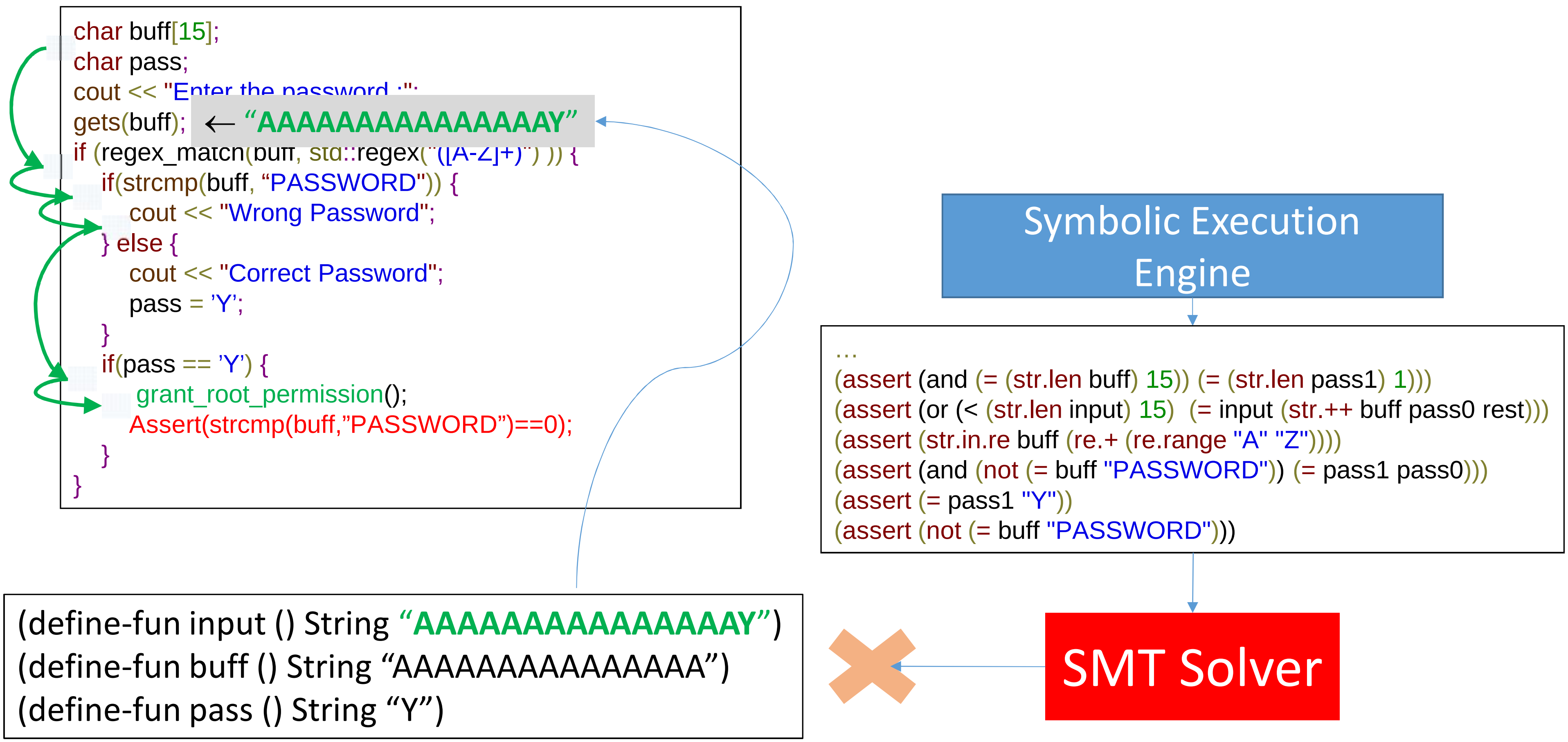
Symbolic Execution Engine

```
…
(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 "AAAAAAAAAAAAAAAY")
(define-fun buff () String "AAAAAAAAAAAAAAA")
(define-fun pass () String "Y")
```

Symbolic execution



Synthesis Tools

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

@ensures: $\forall i.(c_{out}[i] \geq a[i] \wedge c_{out}[i] \geq b[i])$?

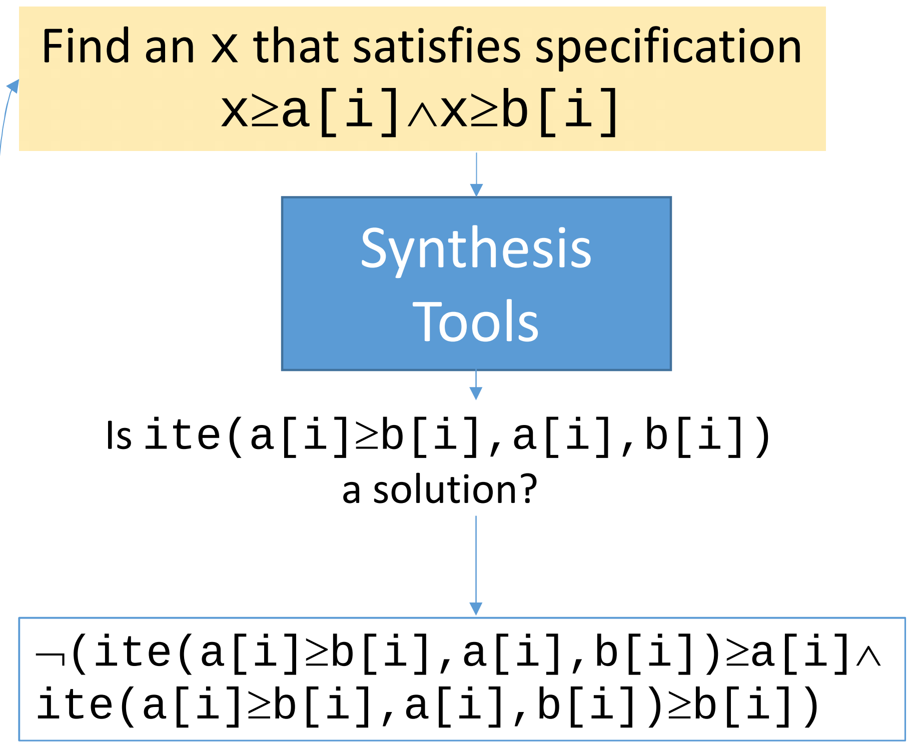
Find an x that satisfies specification

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

Synthesis Tools

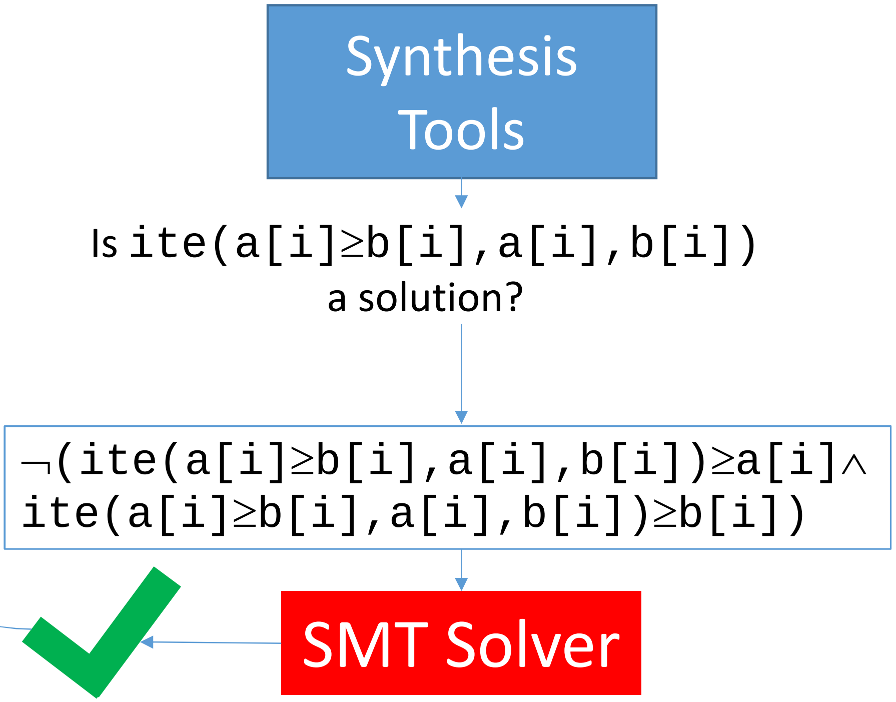
Synthesis Tools

```
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]∧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 = if(a[i]ñb[i]{a[i]}else{b[i]};
    c := c.append(max);
  }
  return c;
}
@ensures: ?i.(c_out[i]ña[i]ėc_out[i]ñb[i])
```



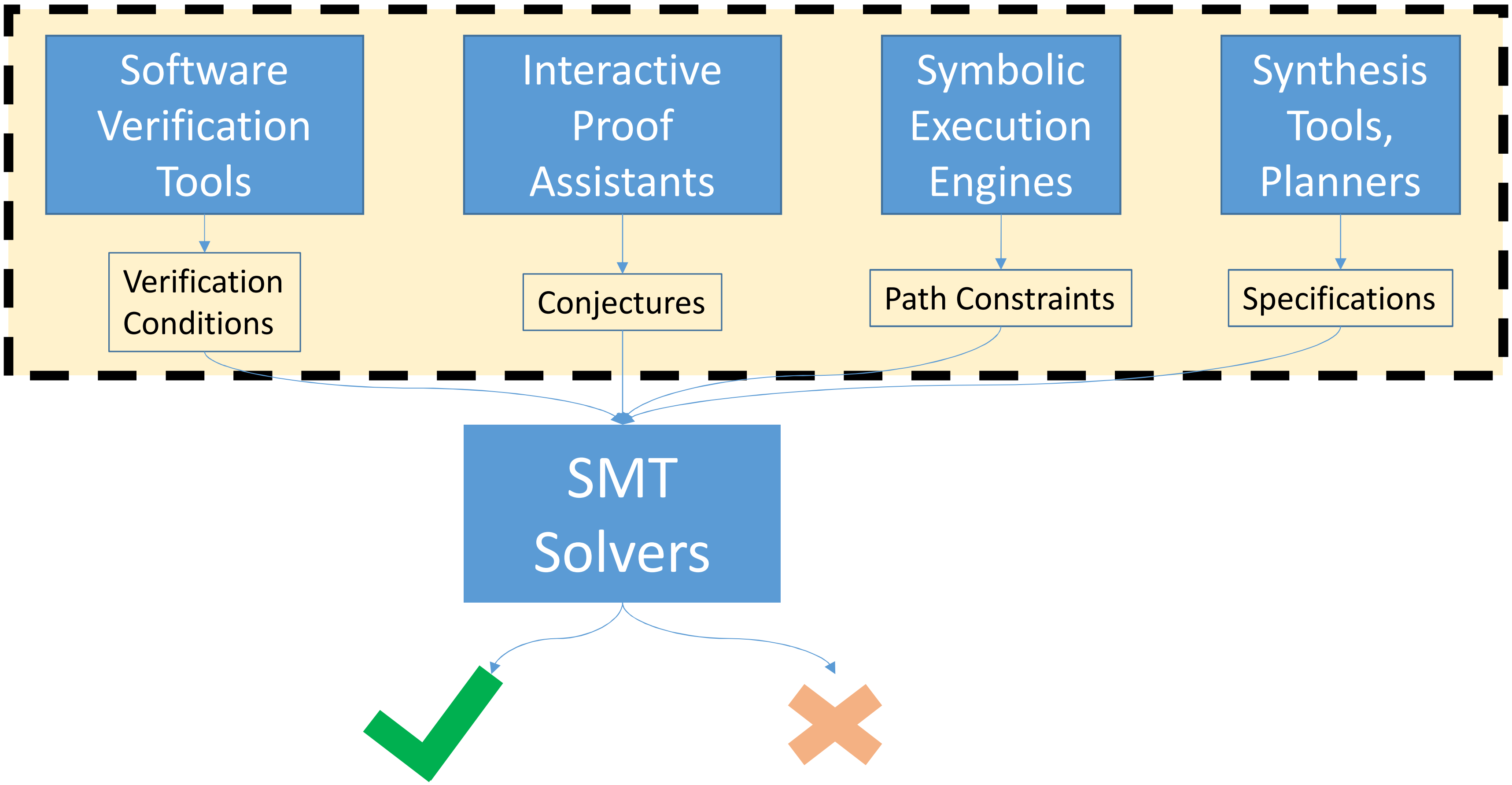
Constraints Supported by SMT Solvers

- 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)`
    - …
  - ∀ over each of these

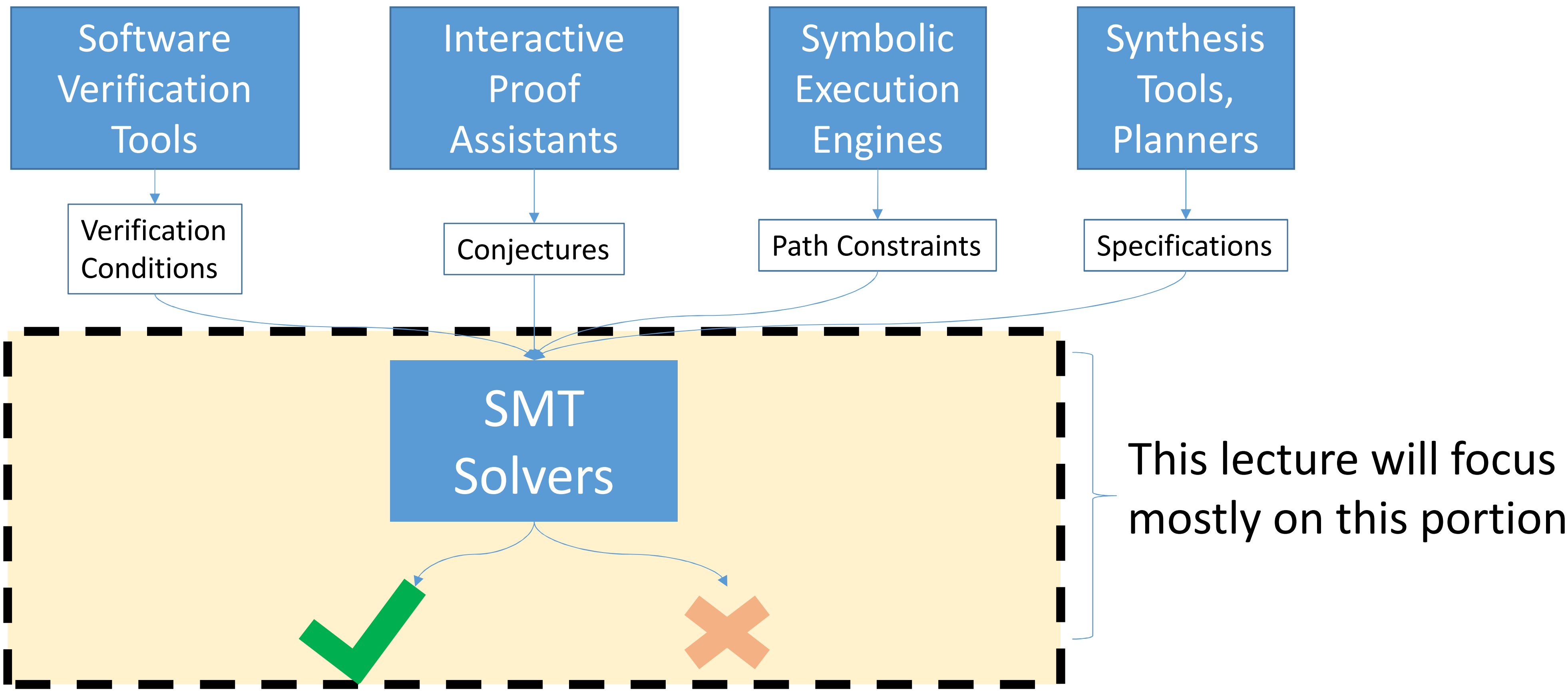
Constraints Supported by SMT Solvers

- SMT solvers support:
  - Arbitrary Boolean combinations of theory constraints
  - Examples of supported theories $\Rightarrow$ decision procedures
    - Uninterpreted functions: $\Rightarrow$ Congruence Closure **[Nieuwenhuis/Oliveras 2005]**
    - Linear real/integer arithmetic: $\Rightarrow$ Simplex **[deMoura/Dutertre 2006]**
    - Arrays: $\Rightarrow$ **[deMoura/Bjorner 2009]**
    - Bit-vectors: $\Rightarrow$ Bitblasting, lazy approaches **[Bruttomesso et al 2007,Hadarean et al 2014]**
    - Algebraic Datatypes: $\Rightarrow$ **[Barrett et al 2007,Reynolds/Blanchette 2015]**
    - Unbounded Strings: $\Rightarrow$ **[Zheng et al 2013,Liang et al 2014, Abdulla et al 2014]**
    - …
  - $\forall$ over each of these

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, 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

- Can download CVC4 binary: http://cvc4.cs.stanford.edu/downloads/
  - Use development version on right hand side

- …or clone from github: https://github.com/CVC4/CVC4

- Lecture material available:

  http://homepage.cs.uiowa.edu/~ajreynol/VTSA2017/