PennState

CSE 597: Security of Emerging Technologies

Module: Formal Verification

Prof. Syed Rafiul Hussain

Systems and Network Security (SyNSec) Research Group

Department of Computer Science and Engineering

The Pennsylvania State University

Computer Systems and Correctness

- Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers.
- All of these components are using software.
- We have requirements on how the system should function, for example safety, reliability, security, availability, absence of deadlocks etc.
- How can one ensure that the system satisfies these requirements?
  - Modern computer systems are unreliable.

Small Example (Software)

- Consider the following C code fragment

```
int* allocateArray(int length)
{
        int i;
        int* array;
        array = malloc(sizeof(int)*length);
        for (i = 0;i <= length;i++) array[i]= 0;
        return array;
}
```

Writing to unallocated memory

Is the program correct?

Small Example (Software)

- Consider the following C code fragment

```
int* allocateArray(int length)
{
        int i;
        int* array;
        array = malloc(sizeof(int)*length);
        for (i = 0;i < length;i++) array[i] = 0;
        return array;
}
```

Returns NULL when allocation fails

Writing to NULL memory

Is the program correct?

Small Example (Software)

```
/*Returns a new array of integers of a given length
   initialized by a non-zero value */
   int* allocateArray(int length){
        int i;
        int* array;
        array = malloc(sizeof(int)*length);
        if(!array) return NULL ;
        for (i = 0;i < length;i++) array[i] = 0;
        return array;
   }
```

Is the program correct?

Program Correctness

- We discussed program correctness without ever defining it

- What is program correctness?

Program Correctness

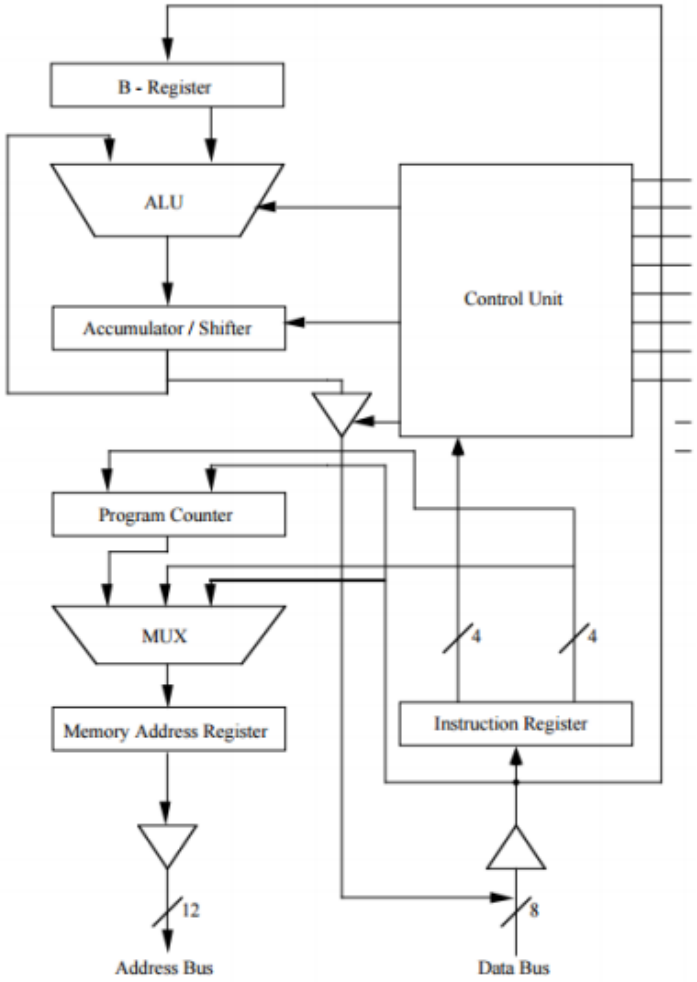
- We did not need to understand the *intended meaning* of the program to identify the first two types of errors.

- We just needed to understand the meaning of C programs in general and some specific properties of programming in C.

Program Correctness

- To understand the last error, however, we needed to understand the intended behavior of the C program.

Example: Circuit Design



We used a circuit ***C1*** in a processor and would like to replace it by another circuit ***C2***.

For example, we may believe that the use of ***C2*** results in a lower energy consumption.

We ***want to be sure*** that ***C2*** is correct, that is, it will behave according to some specification.

If we know that ***C1*** is **correct**, it is sufficient to **prove** that ***C2*** is functionally **equivalent** to ***C1***.

Another Example (Vending Machine)

1. The vending machine contains a drink storage, a coin slot, and a drink dispenser. The drink storage stores drinks of two kinds: beer and coffee. We are only interested in whether a particular kind of drink is currently being stored or not, but not interested in the amount of it.
2. The coin slot can accommodate up to ***three coins***.
3. The drink dispenser can store **at most one drink**. If it contains a drink, this drink should be removed before the next one can be dispensed.
4. A can of beer costs **two coins**. A cup of coffee costs **one coin**.
5. There are two kinds of customers: **students** and **professors**. Students drink only beer, professors drink only coffee.
6. From time to time the drink storage can be recharged.

Property: The vending machine always dispenses the right drink.

How to Establish Correctness?

1. Consider the system (or a computer program) as a mathematical object. To do this, we will have to build a formal model of the system (or the program).
2. Use a formal language for expressing intended properties. The language must have a semantics that explains what are possible interpretations of the sentences of the formal language. The semantics is normally based on the notions *is true*, *is false*, *satisfies*.
3. Write a specification, that is, intended properties of the system in this language.
4. Prove formally that the model satisfies the specification.

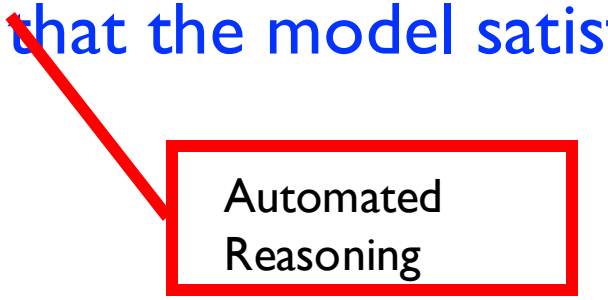
How to Prove Properties of Programs or Systems?

- Hire all people with PhD in mathematical logic in the world;

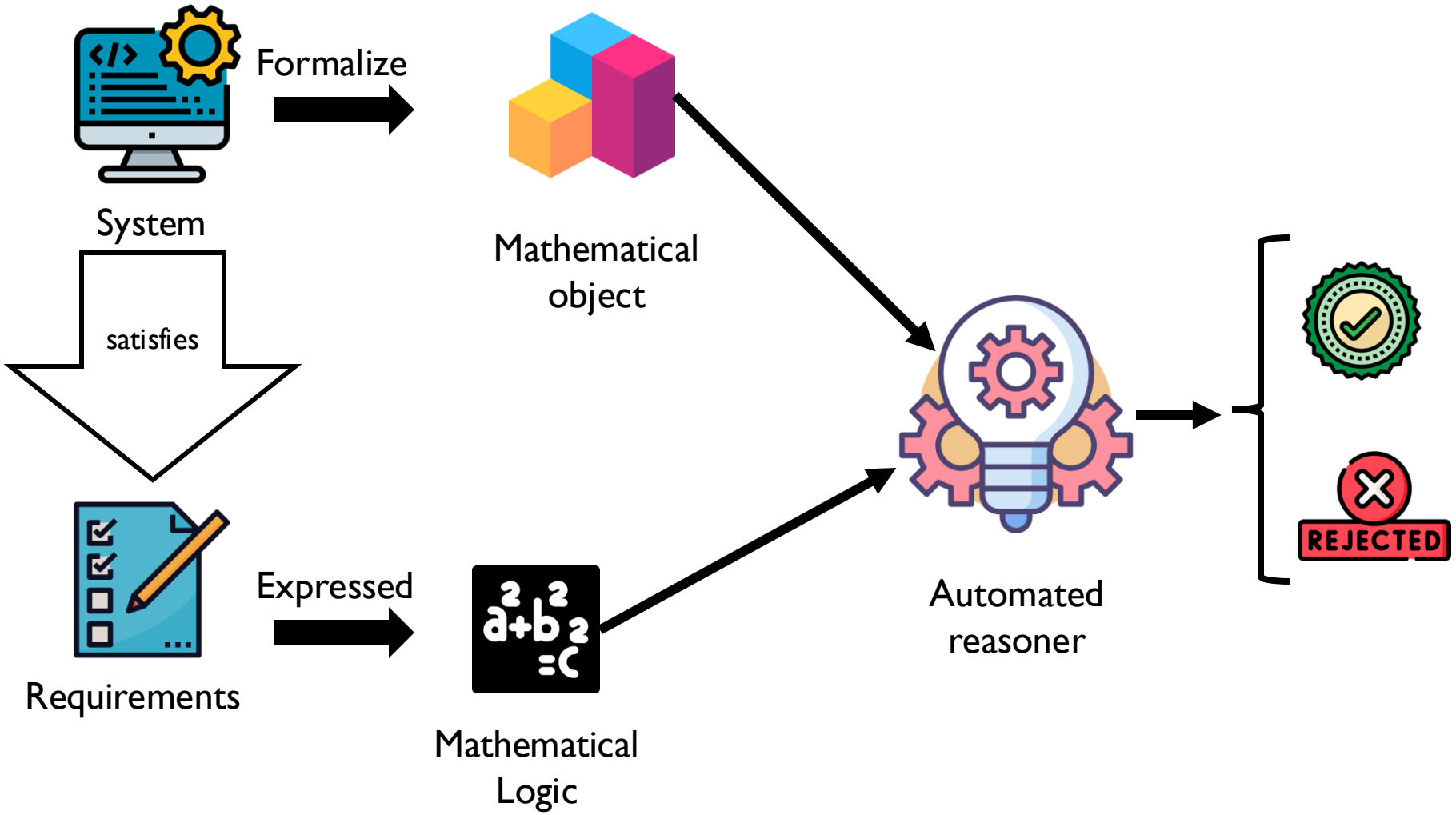
- *Delegate the problem of proving to a computer program.*

How to Establish Correctness?

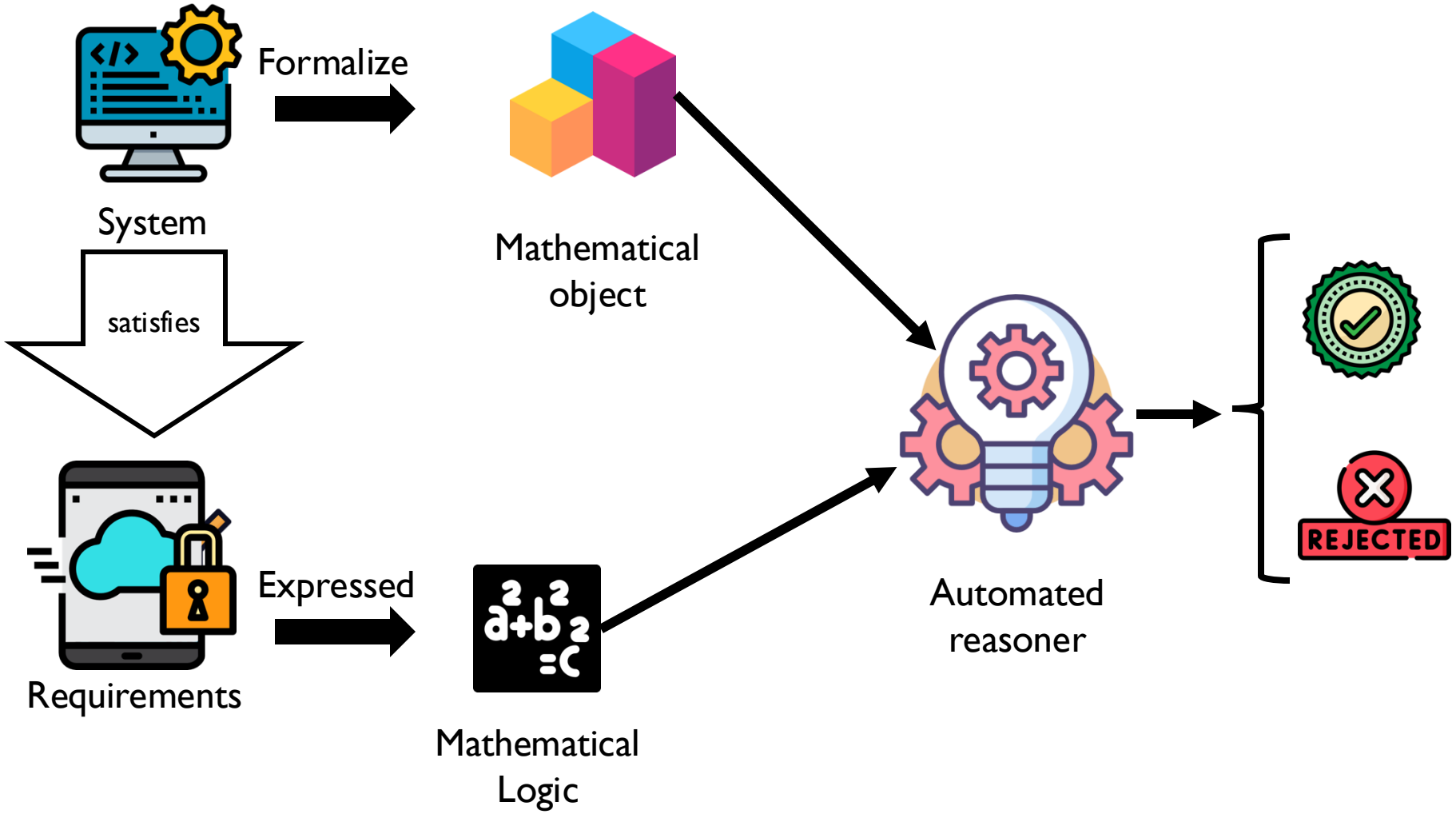
1. Consider the system (or a computer program) as a mathematical object. To do this, we will have to build a formal model of the system (or the program).
2. Use a formal language for expressing intended properties. The language must have a semantics that explains what are possible interpretations of the sentences of the formal language. The semantics is normally based on the notions *is true*, *is false*, *satisfies*.
3. Write a specification, that is, intended properties of the system in this language.
4. Prove formally that the model satisfies the specification.



Automated Reasoning



Why teach automated reasoning in security?



What is Logic?

- Mathematical logic is a branch of science that deals with notions such as
  - Syntax and semantics;
  - Proof theory and model theory;
  - Reasoning.

Computational Logic

- Computational logic deals with applications of logic in computer science and computer engineering, including:
  - Software and hardware verification;
  - Circuit design;
  - Constraint satisfaction;
  - Knowledge representation and reasoning;
  - Semantic web;
  - Planning;
  - Databases (semantics and query optimization);
  - Theorem proving in mathematics.

Proposition

- Propositional Logic formalizes the notion of proposition, that is a statement that can be either true or false.
- There are simple propositions called atomic. For example:
  - 0 < 1;
  - Alan Turing was born in Manchester;
  - 1+1=10.
- More complex propositions are built from simpler ones using a small number of constructs. Examples of more complex propositions:
  - If 0 < 1, then Alan Turing was born in Manchester;
  - 1+1=10 or1+1≠ 10.

Truth

- Each proposition is either true or false.
- The truth value of an atomic proposition, that is, either true or false depends on an interpretation of such propositions.
- For example, 1 + 1 = 10 is false, if we interpret sequences of digits as the decimal notation for numbers and true if we use the binary notation.
- If a complex proposition C is build from simpler propositional $S1, \ldots, Sn$ using a construct, then the truth value of C is determined by the truth value of $S1, \ldots, Sn$. More precisely, it is a function of truth values of $S1, \ldots, Sn$ defined by this construct.
- For example, 1 + 1 = 10 or 1 + 1 ≠10 is true if 1 + 1 ≠ 10 is true.

Propositional Logic: Syntax

- Assume a countable set of boolean variables. Propositional formula:
  - Every boolean variable is a formula, also called atomic formula, or simply atom.
  - $\top$ and $\bot$ are formulas.
  - If $A1,...,An$ are formulas,where $n \geq 2$,then$(A1 \wedge ... \wedge An)$and
  - $(A1 \vee ... \vee An)$ are formulas.
  - If A is a formula, then $(\neg A)$ is a formula.
  - If A and B are formulas, then $(A \rightarrow B)$ and $(A \leftrightarrow B)$ are formulas.
- The symbols $\top$, $\bot$, $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$ are called connectives.

Semantics and Interpretation

- Consider an arithmetical expression, for example

$$x \cdot y + 2 \cdot z.$$

- In arithmetic, the meaning of expressions with variables is defined as follows. Take a mapping from variables to (integer) values, for example

$$\{x \mapsto 1, y \mapsto 7, z \mapsto -3\}.$$

- Then, under this mapping, the expression has the value 1. In other words, when we interpret variables as values, we can compute the value of any expression built using these variables.

Semantics and Interpretation

- Likewise, the semantics of propositional formulas can be defined by assigning values to variables.

- There are two boolean values, also called truth values: true (denoted 1) and false (denoted 0).

- An interpretation for a set P of boolean variables is a mapping $I : P \mapsto \{1, 0\}$.
- Interpretations are also called truth assignments.

Interpreting Formulas

- The truth value of a complex formula is determined by the truth values of its components.
- Given an interpretation I, extend I to a mapping from all formulas to truth values as follows.
  - 1. $I(\top)=1$ and $I(\bot)=0$.
  - 2. $I(A1 \wedge ... \wedge An)=1$ if and only if $I(Ai)=1$ for all i.
  - 3. $I(A1 \vee ... \vee An)=1$ if and only if $I(Ai)=1$ for some i.
  - 4. $I(\neg A)=1$ if and only if $I(A)=0$.
  - 5. $I(A1 \rightarrow A2)=1$ if and only if $I(A1)=0$ or $I(A2)=1$.
  - 6. $I(A1 \leftrightarrow A2)=1$ if and only if $I(A1)=I(A2)$.

Satisfiability, Validity and Equivalence

- Let A be a formula.
  - If $I(A)=1$, then we say that the formula A is true in I and that I satisfies A and that I is a model of A, denoted by $I \models A$.
  - If $I(A)=0$, then we say that the formula A is false in I.
  - A is satisfiable if it is true in some interpretation.
  - A is valid (or a tautology) if it is true in every interpretation.
  - Two formulas A and B are called equivalent, denoted by $A \equiv B$ if they have the same models.

Connections between these notions

- A formula A is valid if and only if $\neg A$ is unsatisfiable.
- A formula A is satisfiable if and only if $\neg A$ is not valid.

- A formula A is valid if and only if A is equivalent to $\top$.
- Formulas A and B are equivalent if and only if the formula $A \leftrightarrow B$ is valid.

- Formulas A and B are equivalent if and only if the formula $\neg(A \leftrightarrow B)$ is unsatisfiable.
- A formula A is satisfiable if and only if A is not equivalent to $\bot$

Evaluate a formula

Let us evaluate the formula

$$(p \to q) \wedge (p \wedge q \to r) \to (p \to r)$$

in the interpretation

$$\{p \mapsto 1, q \mapsto 0, r \mapsto 1\}.$$

The value of this formula is equal to the value of

$$(\top \to \bot) \wedge (\top \wedge \bot \to \top) \to (\top \to \top).$$

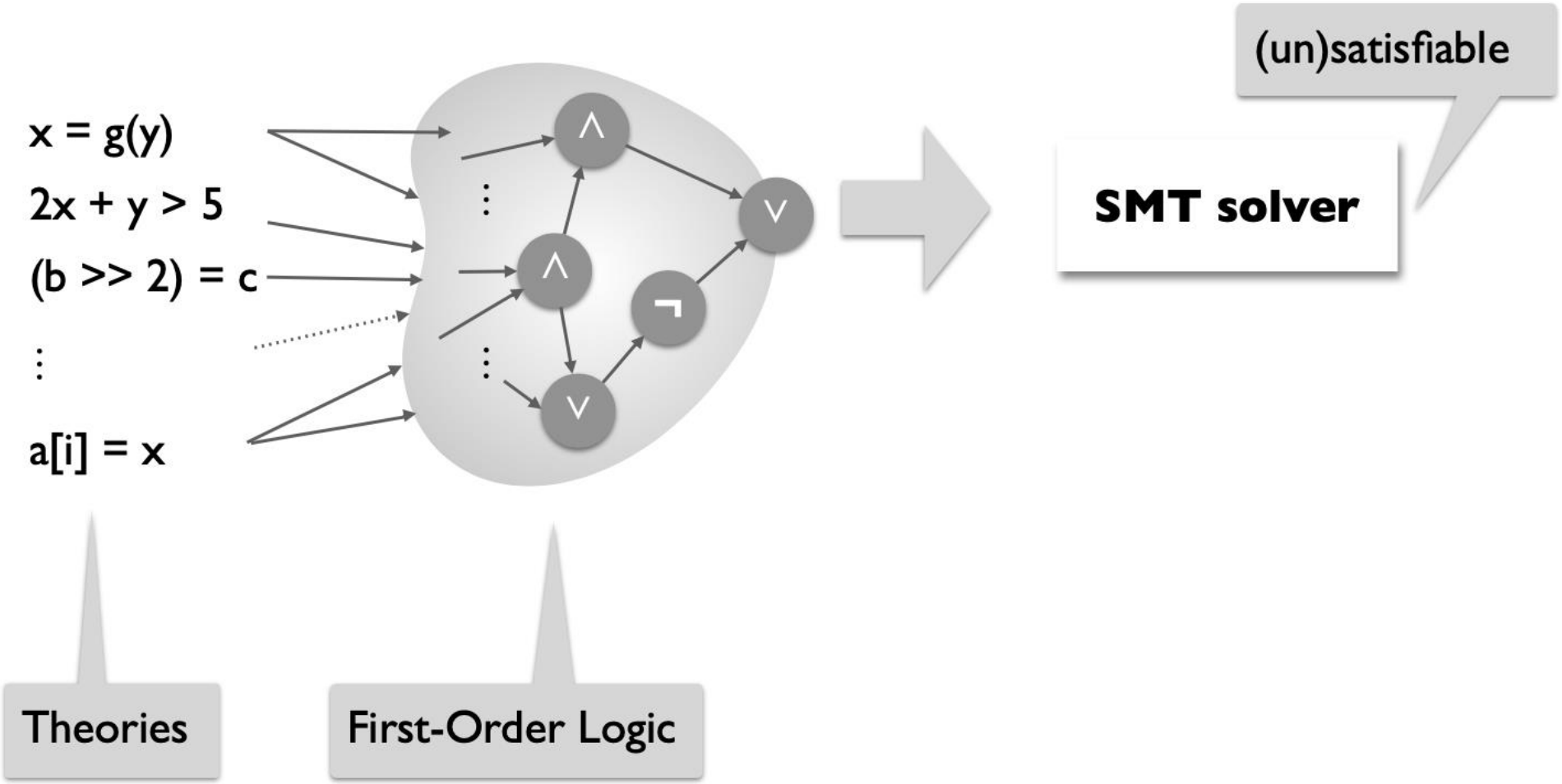
Apply rewrite rules

Inside-out, left-to-right:

$$\begin{aligned}
&(\top \to \bot) \wedge (\top \wedge \bot \to \top) \to (\top \to \top) \Rightarrow \\
&\bot \wedge (\top \wedge \bot \to \top) \to (\top \to \top) \Rightarrow \\
&\bot \wedge (\bot \to \top) \to (\top \to \top) \Rightarrow \\
&\bot \wedge \top \to (\top \to \top) \Rightarrow \\
&\bot \to (\top \to \top) \Rightarrow \\
&\bot \to \top \Rightarrow \\
&\top
\end{aligned}$$

$$\begin{aligned}
&A \wedge \bot \Rightarrow \bot \\
&\top \to \bot \Rightarrow \bot \\
&A \to \top \Rightarrow \top
\end{aligned}$$

Satisfiability Modulo Theories (SMT)

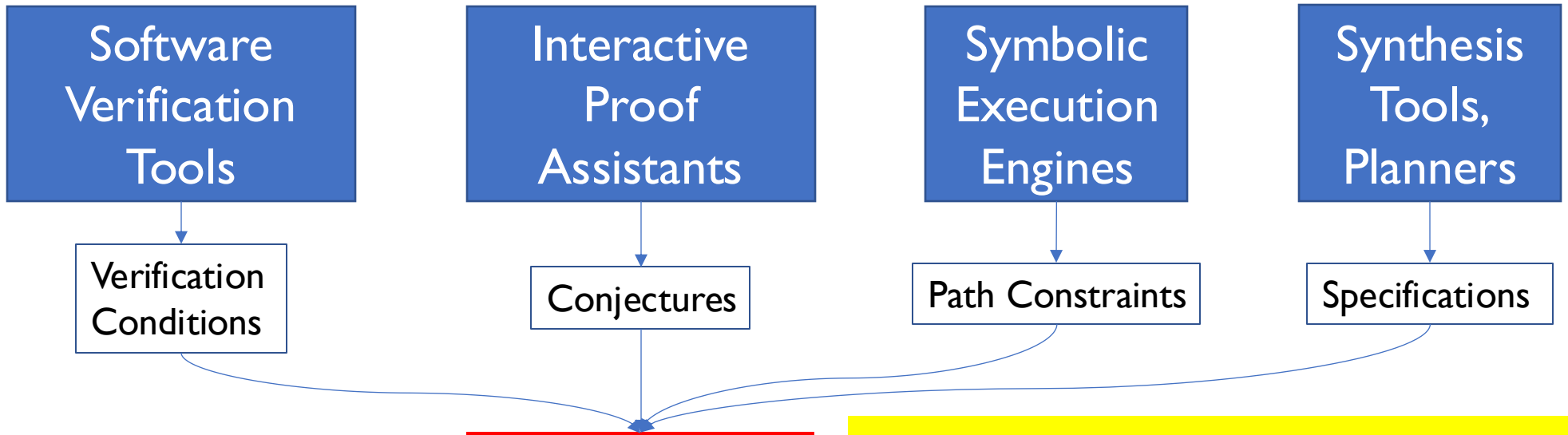


First Order Logic (FOL)

- **Logical symbols**
  - Connectives: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
  - Parentheses: ()
  - Quantifiers: $\forall, \exists$ ✗
- **Non-logical symbols**
  - Constants: x,y,z
  - N-ary functions: f, g
  - N-ary predicates: p, q
  - Variables: u,v,w ✗

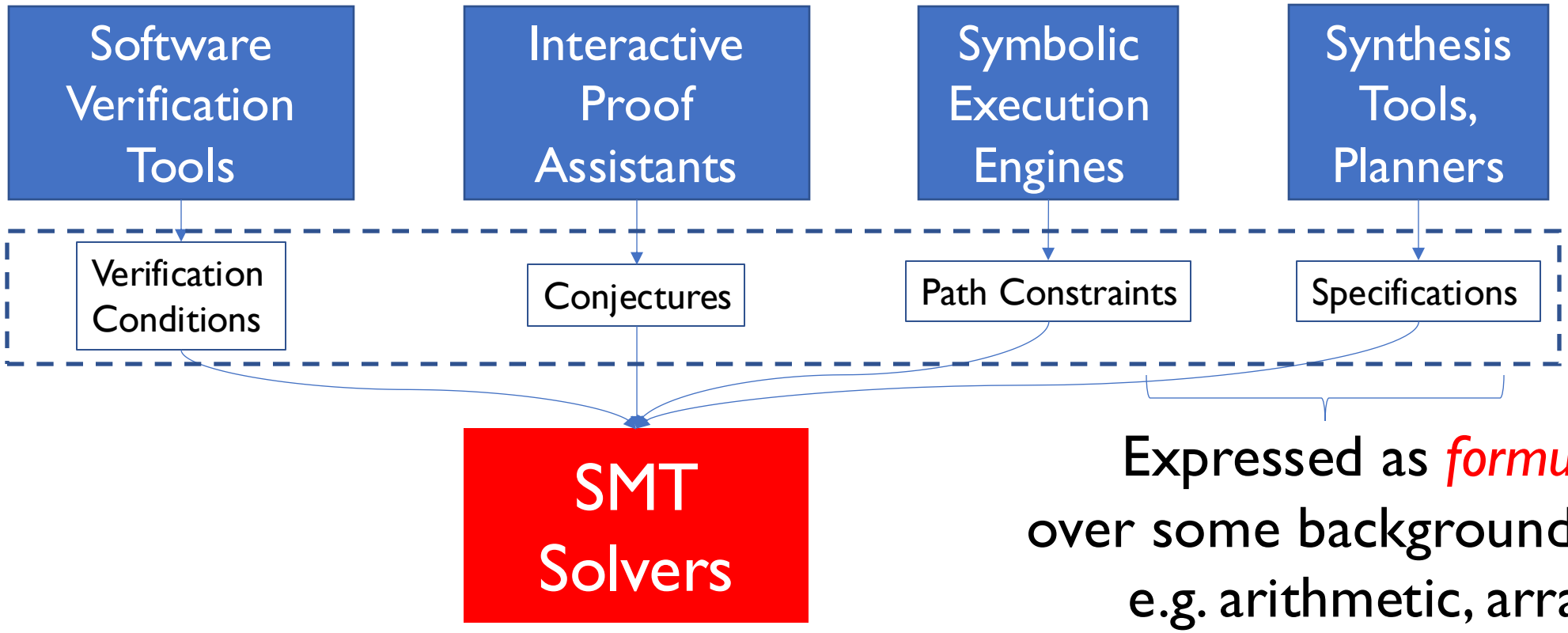
- We will only consider the quantifier free fragment of FOL.

Satisfiability Modulo Theories (SMT) Solvers



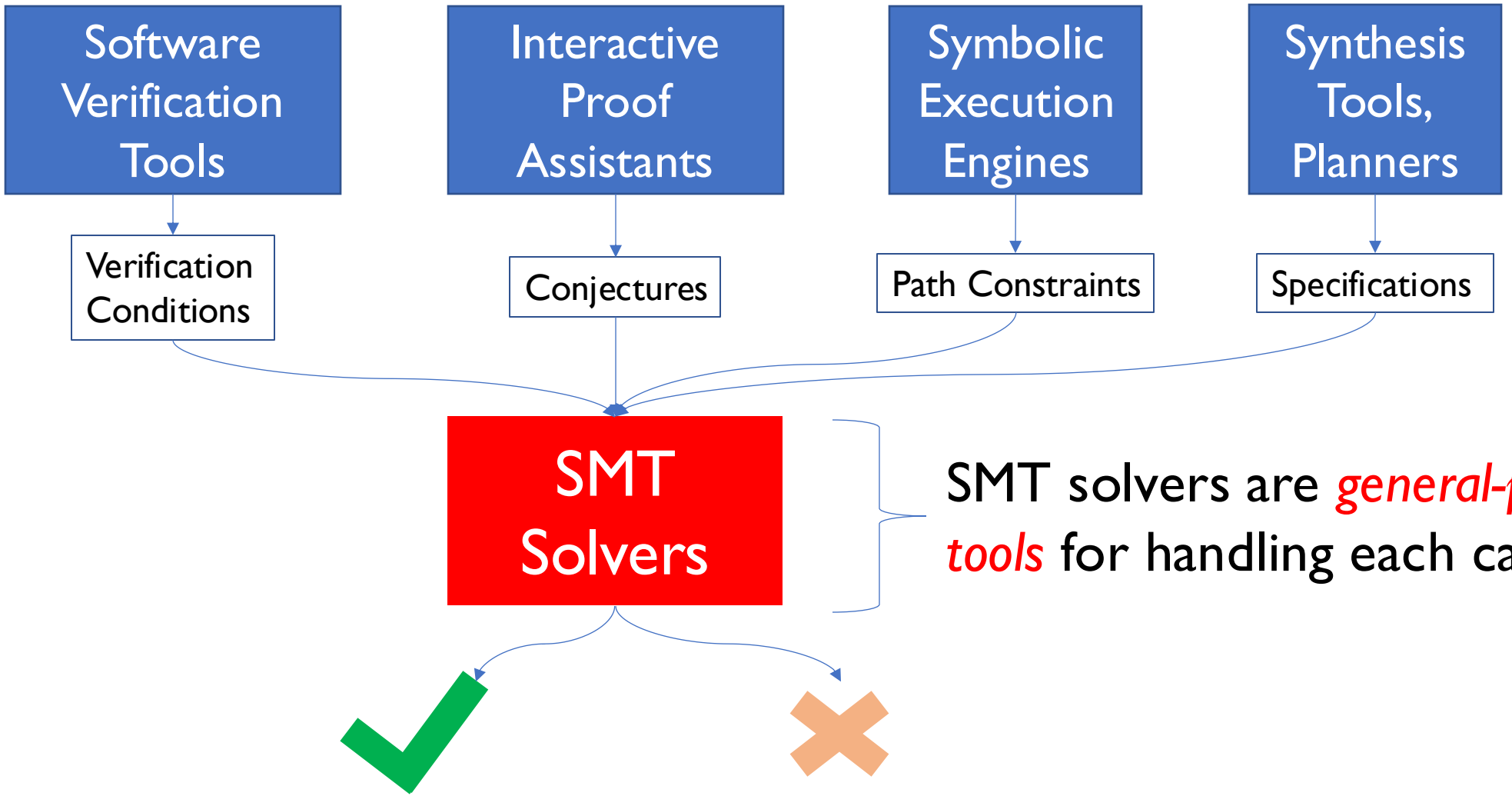
- SMT solvers are:
  - Fully automated reasoners
  - Widely used in applications

Satisfiability Modulo Theories (SMT) Solvers



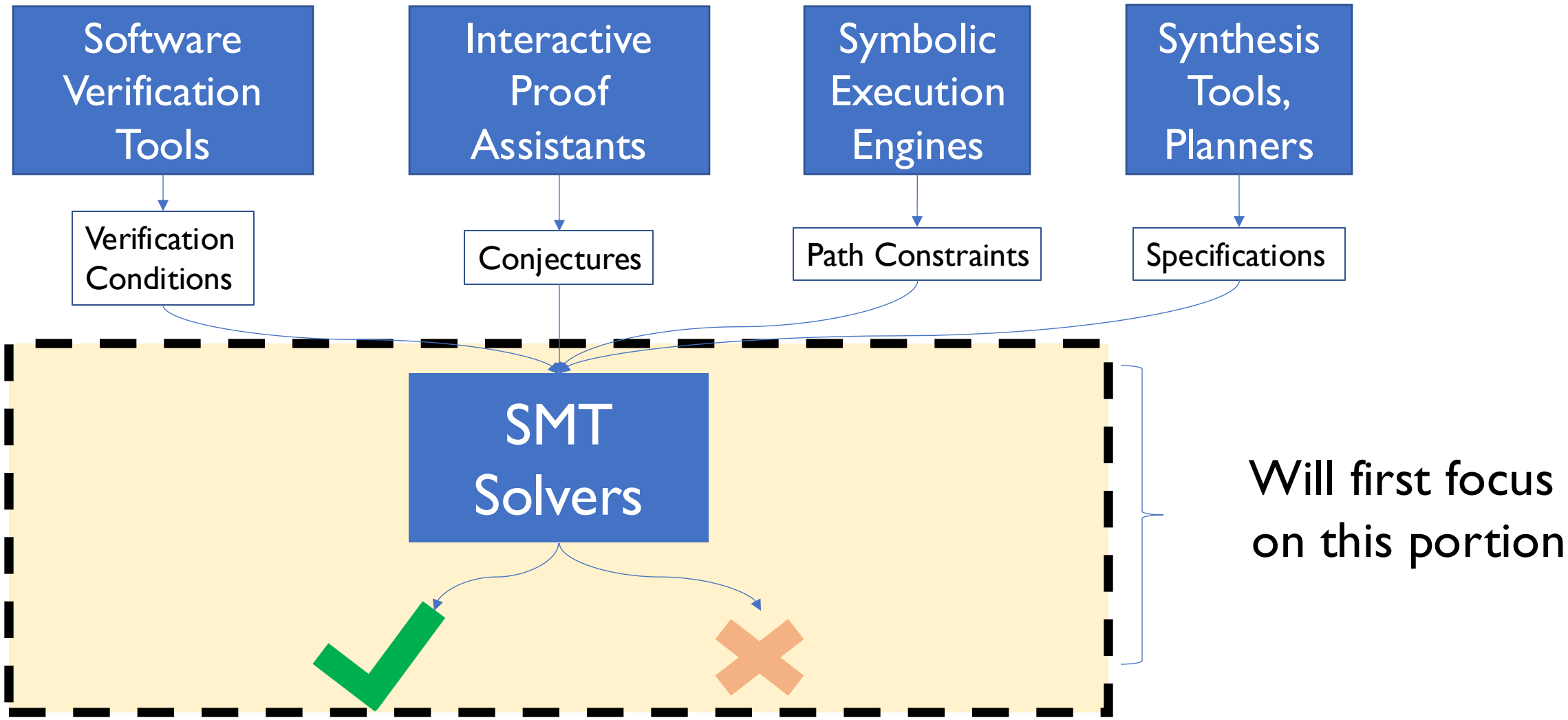
Expressed as *formulas*
over some background *theory*
e.g. arithmetic, arrays

Satisfiability Modulo Theories (SMT) Solvers



SMT solvers are *general-purpose tools* for handling each case

Satisfiability Modulo Theories (SMT) Solvers

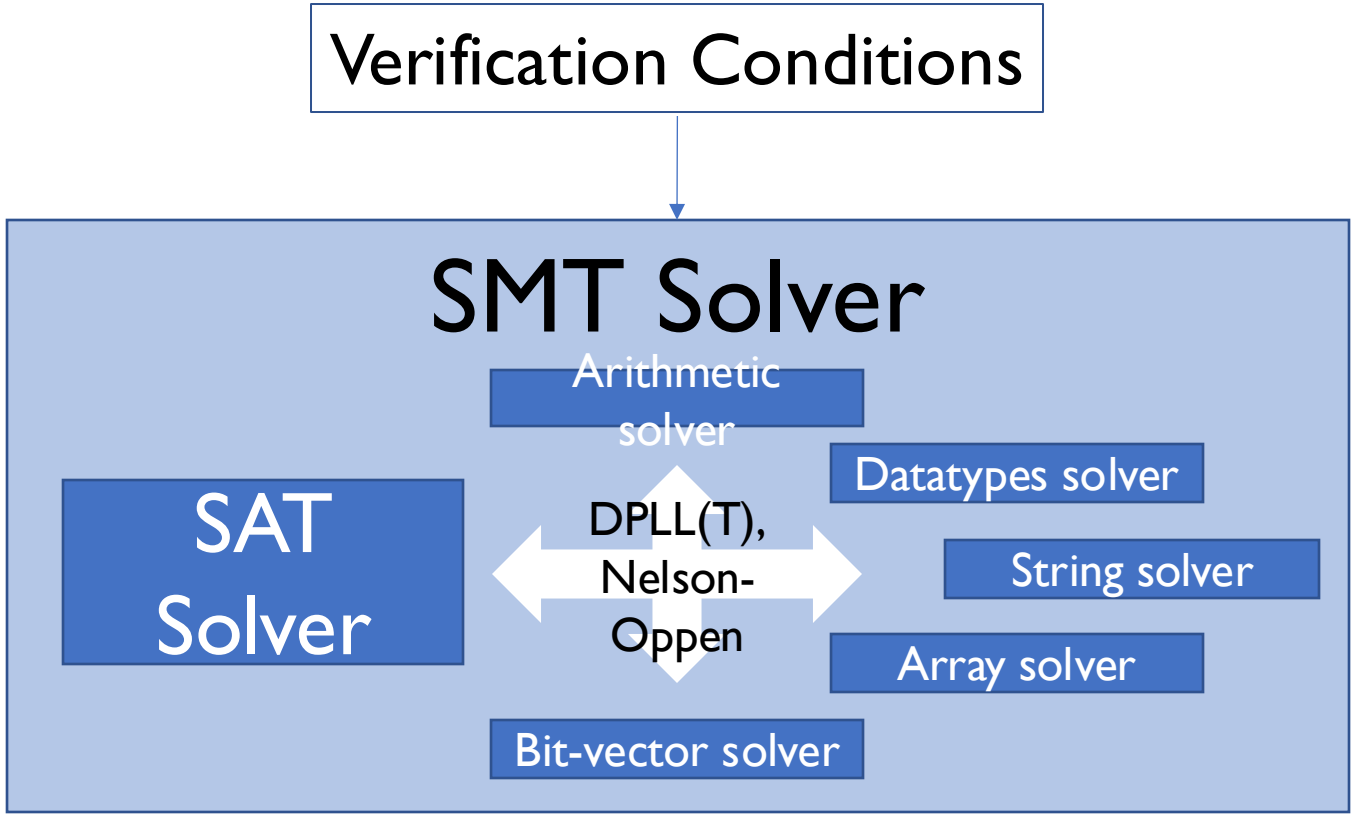


Overview

- Satisfiability Modulo Theories (SMT) solvers: how they work
  - DPLL, DPLL(T), decision procedures,

SMT solvers

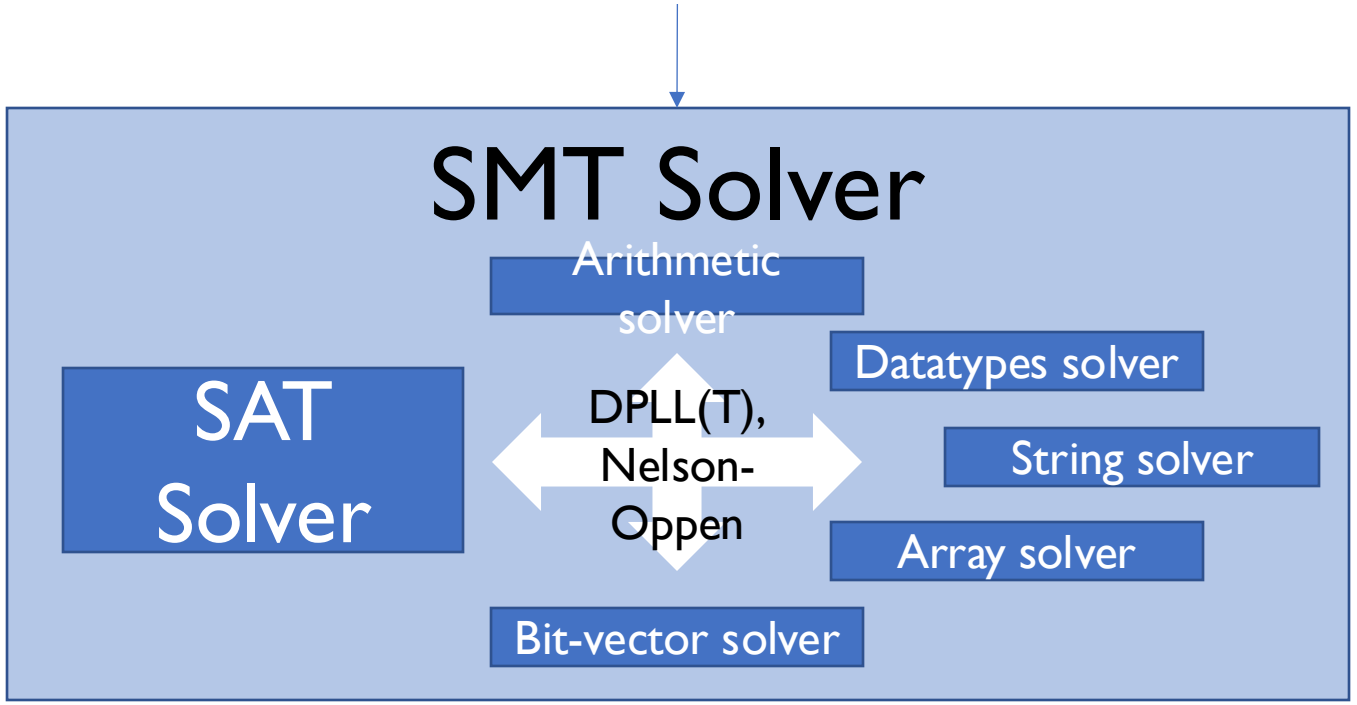
- Efficient tools for satisfiability and satisfiability *modulo theories*



SMT solvers

- Efficient tools for satisfiability and satisfiability *modulo theories*

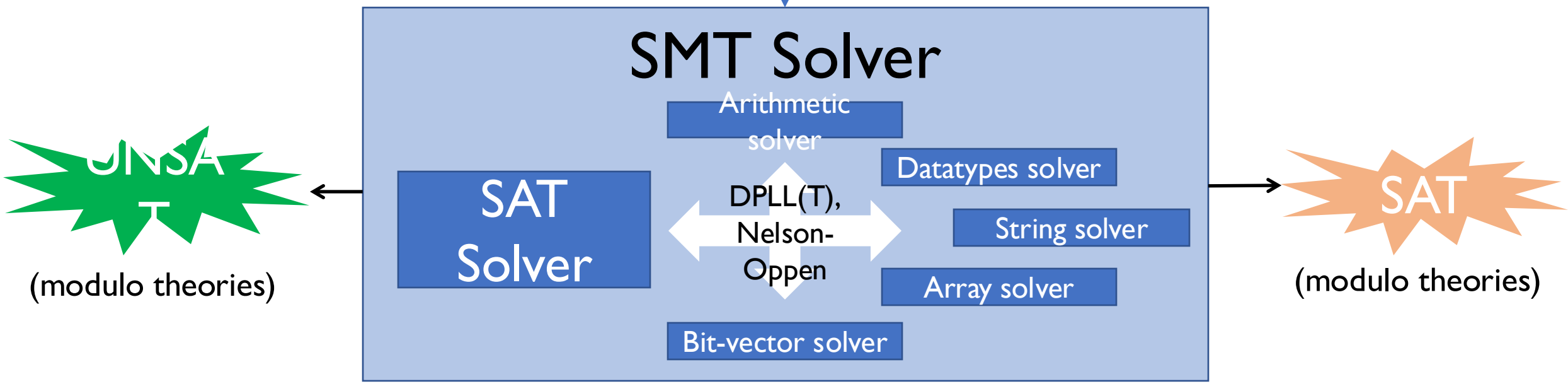
$$(A[x]+B[x]>0 \vee x+y>0) \wedge (cons(\text{“abc”},d_1)\neq d_2 \vee x<0)$$



SMT solvers

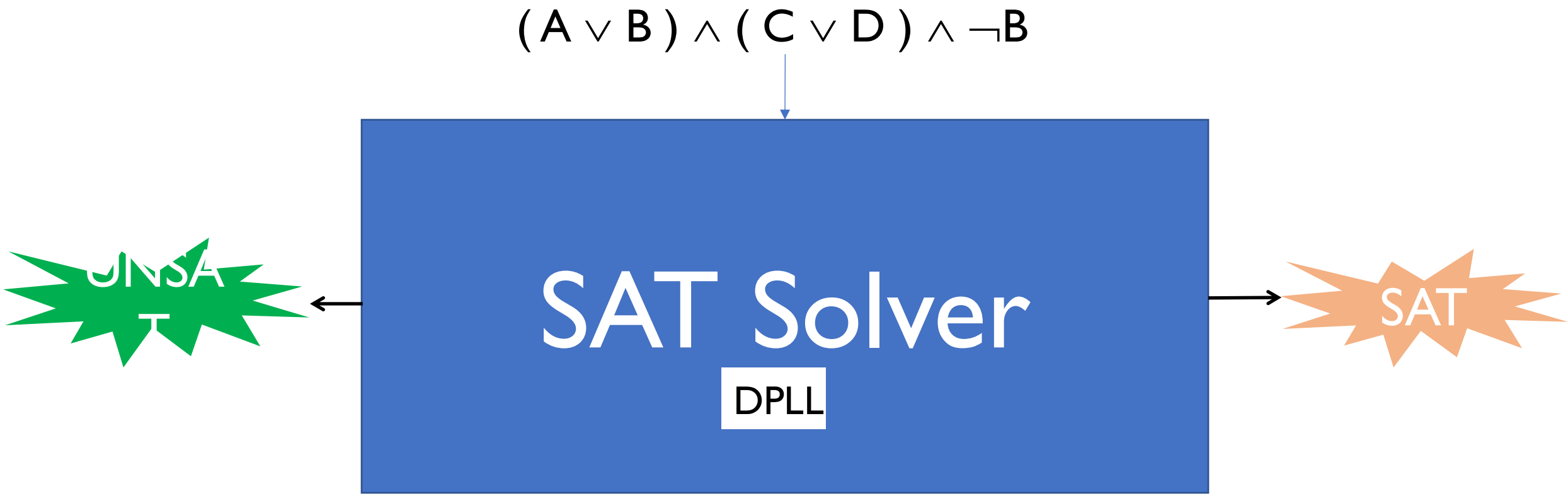
- Efficient tools for satisfiability and satisfiability *modulo theories*

$$(A[x]+B[x]>0 \vee x+y>0) \wedge (cons(\text{“abc”},d_1)\neq d_2 \vee x<0)$$



…but first : SAT solvers

- Efficient tools for *satisfiability*



DPLL (Davis–Putnam–Logemann–Lovelan)

$$(\neg A \Rightarrow B) \wedge (C \vee D) \wedge \neg B$$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

Convert to clausal normal form (CNF)

- A formula is CNF if it is a conjunction of clauses
- A *clause* is a disjunction of literals e.g. $(A \vee B)$
- A *literal* is an atom or its negation e.g. A, $\neg B$,…

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- Alternate between:
  - Propagations : assign values to atoms whose value is forced
  - Decisions : choose an arbitrary value for an unassigned atom

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

Context

$B \rightarrow \bot$

- DPLL algorithm
  - Propagate : $B \rightarrow$ false

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

Context

$B \rightarrow \bot$

$A \rightarrow T$

- DPLL algorithm
  - Propagate : $B \rightarrow$ false
  - Propagate : $A \rightarrow$ true

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

Context

$B \rightarrow \bot$

$A \rightarrow T$

$C \rightarrow T^d$

- DPLL algorithm
  - Propagate : $B \rightarrow$ false
  - Propagate : $A \rightarrow$ true
  - Decide : $C \rightarrow$ true

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

Context

$B \rightarrow \bot$
$A \rightarrow T$
$C \rightarrow T^d$

- DPLL algorithm
  - Propagate : $B \rightarrow$ false
  - Propagate : $A \rightarrow$ true
  - Decide : $C \rightarrow$ true

$\Rightarrow$ Input is 

by interpretation where

$\{A \rightarrow T, B \rightarrow \bot, C \rightarrow T\}$

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

Context

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Decide : $A \rightarrow$ true

Context

$A \rightarrow T^{d}$

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

Alternatively, can view context as set of literals

- DPLL algorithm
  - Decide : $A \rightarrow$ true

Context

A^d

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Decide : $A \rightarrow$ true
  - Propagate : $B \rightarrow$ true

Context

A^d

B

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Decide : $A \rightarrow$ true
  - Propagate : $B \rightarrow$ true
  - Propagate : $C \rightarrow$ false

Context

A^d

B

$\neg C$

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Decide : $A \rightarrow$ true
  - Propagate : $B \rightarrow$ true
  - Propagate : $C \rightarrow$ false

$\Rightarrow$ Conflicting clause!
(all literals are false)

Context

A^d
B
$\neg C$

DPLL

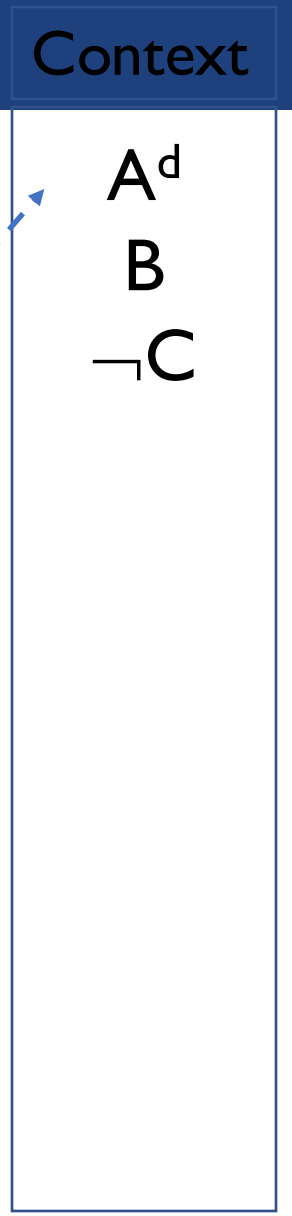
$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true
  - Propagate : C $\rightarrow$ false

$\Rightarrow$ Conflicting clause!

(all literals are false)

…*backtrack* on a decision



DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Backtrack : $A \rightarrow$ false

Context

$\neg A$

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Backtrack : $A \rightarrow$ false
  - Propagate : $D \rightarrow$ true

Context

$\neg A$
D

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

- DPLL algorithm
  - Backtrack : $A \rightarrow$ false
  - Propagate : $D \rightarrow$ true
  - Decide : $B \rightarrow$ false

Context

$\neg A$

D

B^d

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : $A \rightarrow$ false
  - Propagate : $D \rightarrow$ true
  - Decide : $B \rightarrow$ false

$\Rightarrow$ Input is 

by interpretation where

$\{A \rightarrow \bot, B \rightarrow \bot, D \rightarrow T\}$

Context

$\neg A$

D

B^d

DPLL

- Important optimizations:
  - Two watched literals
  - Non-chronological backtracking
  - Conflict-driven clause learning (CDCL)
  - Decision heuristics
  - Preprocessing / in-processing

SAT

- Using an encoding of problems into propositional logic:
  - **Pros** : Decidable, very efficient CDCL-based SAT solvers available
  - **Cons** : Not expressive, may require exponentially large encoding

    $\Rightarrow$ Motivation for Satisfiability *Modulo Theories*

SMT solvers handle formulas like:

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

SMT solvers handle formulas like:

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- …using DPLL(**T**) algorithm for satisfiability modulo T
  - Extends DPLL algorithm to incorporate reasoning about a theory **T**
  - Combines:
    - Off-the-shelf CDCL-based SAT solver
    - *Theory Solver for T*

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- DPLL(LIA) algorithm

  Invoke DPLL(T) for theory T = LIA (linear integer arithmetic)

DPLL(T)

Context

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- DPLL(LIA) algorithm

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false

Context

$\neg x+y>0$

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true

Context

$\neg x+y>0$

$x+1>0$

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true

Context

$\neg x+y>0$

$x+1>0$

$x<0^d$

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true

$\Rightarrow$ ***Unlike propositional SAT case, we must check T-satisfiability of context***

Context

$\neg x+y>0$

$x+1>0$

$x<0^d$

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }

Context

$\neg x+y>0$

$x+1>0$

$x<0^d$

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }

Context is LIA-unsatisfiable!
$\Rightarrow$ one of $x+1>0$, $x<0$ must be false

Context

$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }
    - Add *theory lemma* ($\neg x+1>0 \vee \neg x<0$)

Context

$\neg x+y>0$

$x+1>0$

$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0)$ $\Rightarrow$ Conflicting clause!

…*backtrack* on a decision

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : { $x+1>0$, $\neg x+y>0$, $x<0$ }
    - Add *theory lemma* ($\neg x+1>0 \vee \neg x<0$)

Context

$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true

Context

$\neg x+y>0$

$x+1>0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - *Propagate* : $x<0 \rightarrow$ false

Context

$\neg x+y>0$
$x+1>0$
$\neg x<0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true

Context

$\neg x+y>0$

$x+1>0$

$\neg x<0$

$x+y>4$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0, \neg x+y>0, \neg x<0, x+y>4$ }

Context

$\neg x+y>0$

$x+1>0$

$\neg x<0$

$x+y>4$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }

Context is LIA-unsatisfiable!
$\Rightarrow$ one of $\neg x+y>0$, $x+y>4$ must be false

Context

$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$$
$$(\neg x+1>0 \vee \neg x<0) \wedge (x+y>0 \vee \neg x+y>4)$$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }
    - Add *theory lemma* ($x+y>0 \vee \neg x+y>4$)

Context

$\neg x+y>0$

$x+1>0$

$\neg x<0$

$x+y>4$

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$$
$$(\neg x+1>0 \vee \neg x<0) \wedge (x+y>0 \vee \neg x+y>4)$$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }
    - Add *theory lemma* ($x+y>0 \vee \neg x+y>4$)

$\Rightarrow$ Conflicting clause!

…no decision to backtrack

Context

$\neg x+y>0$

$x+1>0$

$\neg x<0$

$x+y>4$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0) \wedge (x+y>0 \vee \neg x+y>4)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: { $x+1>0$, $\neg x+y>0$, $\neg x<0$, $x+y>4$ }
    - Add *theory lemma* ($x+y>0 \vee \neg x+y>4$)

$\Rightarrow$ Conflicting clause!

…no decision to backtrack

$\Rightarrow$ *Input is* LIA-unsat

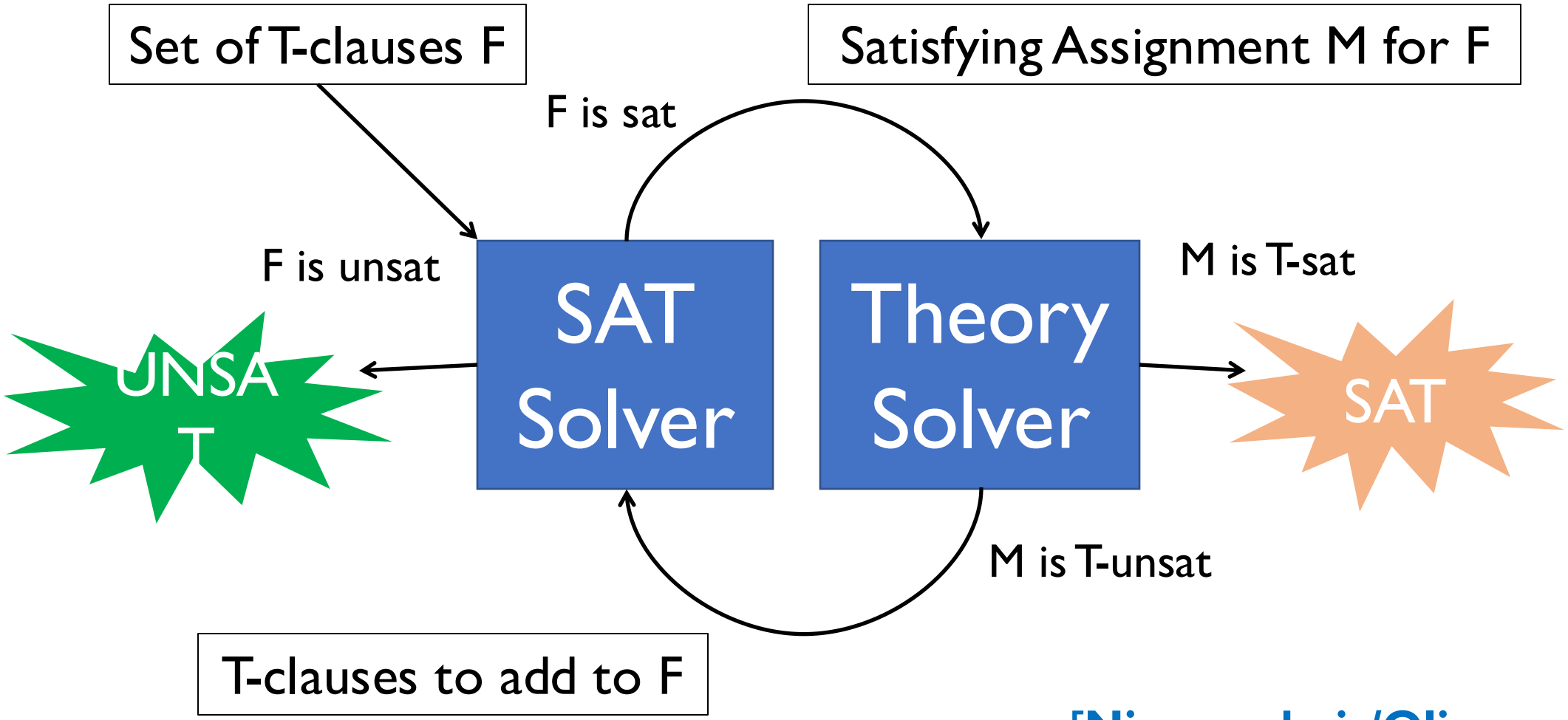
Context

$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

Encoding in *.smt2 format

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (or (> (+ x 1) 0) (> (+ x y) 0)))
(assert (or (< x 0) (> (+ x y) 4)))
(assert (not (> (+ x y) 0)))
(check-sat)
```

DPLL(T)



[Nieuwenhuis/Oliveras/Tinelli 2006]

Verification vs. Falsification

PennState

- An automated verification tool
  - can report that the system is verified (with a proof);
  - or that the system was not verified.

- When the system was not verified, it would be helpful to explain why
  - Model checkers can output an error counterexample: a concrete execution scenario that demonstrates the error.

- Can view a model checker as a falsification tool –
  - The main goal is to find bugs

- So what can we verify or falsify?

Temporal Properties

PennState

- Temporal Property
  - A property with time-related operators such as "invariant" or "eventually"

- Invariant(p)
  - is true in a state if property p is true in every state on all execution paths starting at that state
  - G, AG, ("globally" or "box" or "forall")

- Eventually(p)
  - is true in a state if property p is true at some state on every execution path starting from that state F, AF, ◊ ("future" or "diamond" or "exists")

An Example Concurrent Program

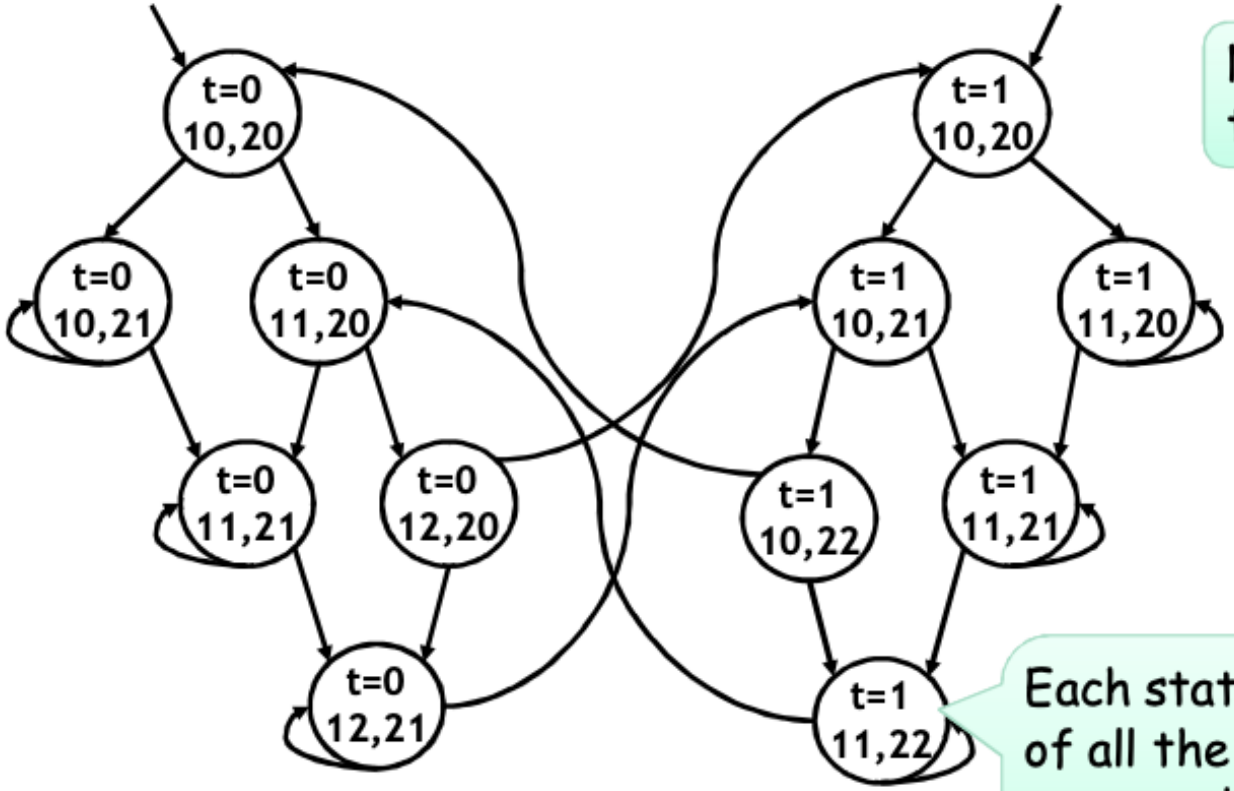
PennState

- A simple concurrent mutual exclusion program
- Two processes execute asynchronously
- There is a shared variable turn
- Two processes use the shared variable to ensure that they are not in the critical section at the same time
- Can be viewed as a "fundamental" program: any bigger concurrent one would include this one

```
10: while (true){
11:     wait(turn == 0);
        // critical section
12:     work(); turn = 1;
13: }

// concurrently with

20: while (true) {
21:     wait(turn == 1);
        // critical section
22: work(); turn = 0;
23: }
```

Reachable States of the Example Program


```
10: while (true){
11:     wait(turn == 0);
        // critical section
12:     work(); turn = 1;
13: }

// concurrently with

20: while (true) {
21:     wait(turn == 1);
        // critical section
22: work(); turn = 0;
23: }
```



Next: formalize this intuition …

Each state is a valuation of all the variables: `turn` and the two program counters for two processes

Analyzed System is a Transition System

PennState

- Labeled transition system

  $T = (S, I, R, L)$ –

  S = Set of states // standard FSM

  $I \subseteq S$ = Set of initial states // standard FSM

  $R \subseteq S \times S$ = Transition relation // standard FSM

  $L: S \rightarrow 2^{AP}$ = Labeling function // this is new!

- AP: Set of atomic propositions (e.g., "x=5"∈AP)
  - Atomic propositions capture basic properties
  - For software, atomic props depend on variable values
  - The labeling function labels each state with the set of propositions true in that state

Example Properties of the Program

PennState

- "In all the reachable states (configurations) of the system, the two processes are never in the critical section at the same time"
  - "pc1=12","pc2=22" are atomic properties for being in the critical section
  - Invariant (¬ (PC1=12 ∧ PC2 = 22)

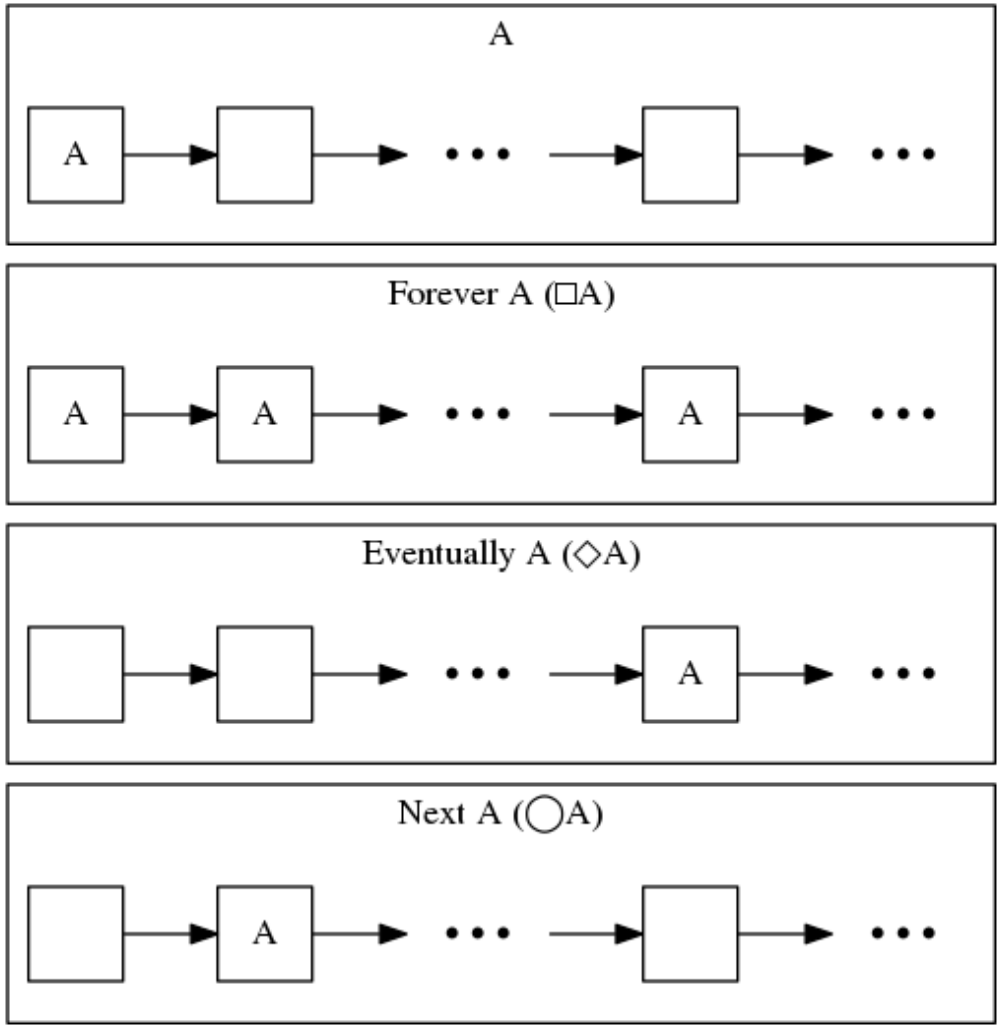
- "Eventually the first process enters the critical section
  - Eventually (PC1 = 12)

```
10: while (true){
11:     wait(turn == 0);
        // critical section
12:     work(); turn = 1;
13: }

// concurrently with

20: while (true) {
21:     wait(turn == 1);
        // critical section
22: work(); turn = 0;
23: }
```

Temporal Logics

PennState

- There are four basic temporal operators:
- X p Next p, p holds in the next state
- G p: Globally p, p holds in every state, p is an invariant
- F p: Future p, p will hold in a future state, p holds eventually
- p U q: p Until q, assertion p will hold until q holds
- Precise meaning of these temporal operators is defined on execution paths



Execution Paths

PennState

- A path in a transition system is an infinite sequence of states
  - $(s0, s1, s2, ...)$, such that $\forall i \geq 0. (si, si+1) \in R$
- A path $(s0, s1, s2, ...)$ is an execution path if $s0 \in I$
- Given a path $x = (s0, s1, s2, ...)$
  - h_i denotes the i-th state: s_i
  - h^i denotes the i-th suffix: $(s_i, s_{i+1}, s_{i+2}, ...)$
  - In some temporal logics one can quantify paths starting from a state using path quantifiers
    - A : for all paths
    - E : there exists a path

Paths and Predicates

PennState

- We write

$$h \models p$$

"the path x makes the predicate p true"

- h is a path in a transition system
- p is a temporal logic predicate •

- Example: A h. $h \models G\ (\neg(pc1=12 \wedge pc2=22))$

Linear Temporal Logic (LTL)

PennState

- LTL properties are constructed from atomic propositions in AP; logical operators $\wedge$, $\vee$, $\neg$ and temporal operators X, G, F, U.
- The semantics of LTL is defined on **paths**
- Given a path h: $h \models p$

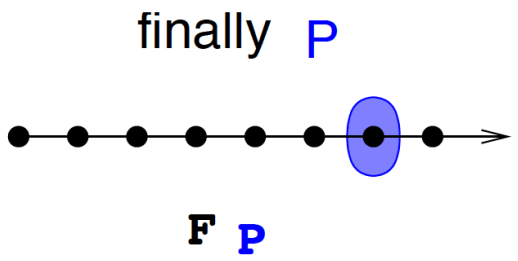
$h \models p$ iff $L(h^0, ap)$ atomic prop

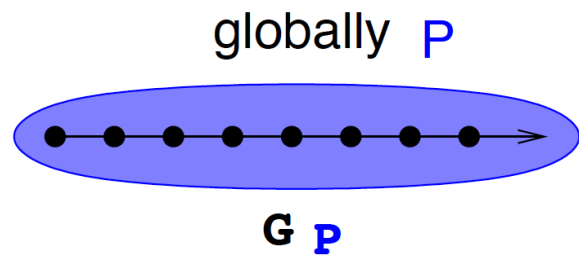
$h \models X\ p$ iff h^1 p next

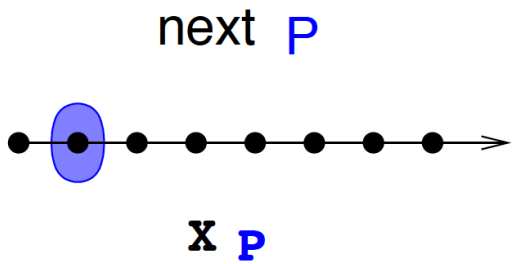
$h \models F\ p$ iff $\exists i \geq 0.\ h^i$ p future

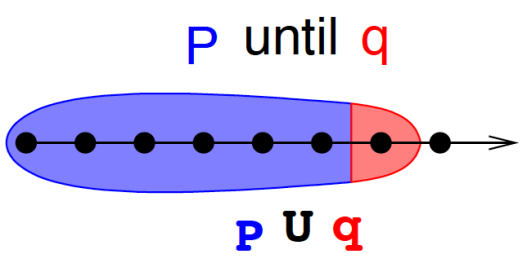
$h \models G\ p$ iff $\forall i \geq 0.\ h^i$ p globally

$h \models p\ U\ q$ iff $\exists i \geq 0.\ h^i$ q and $\forall j < i.\ h^j$ p until









Satisfying Linear Time Logic

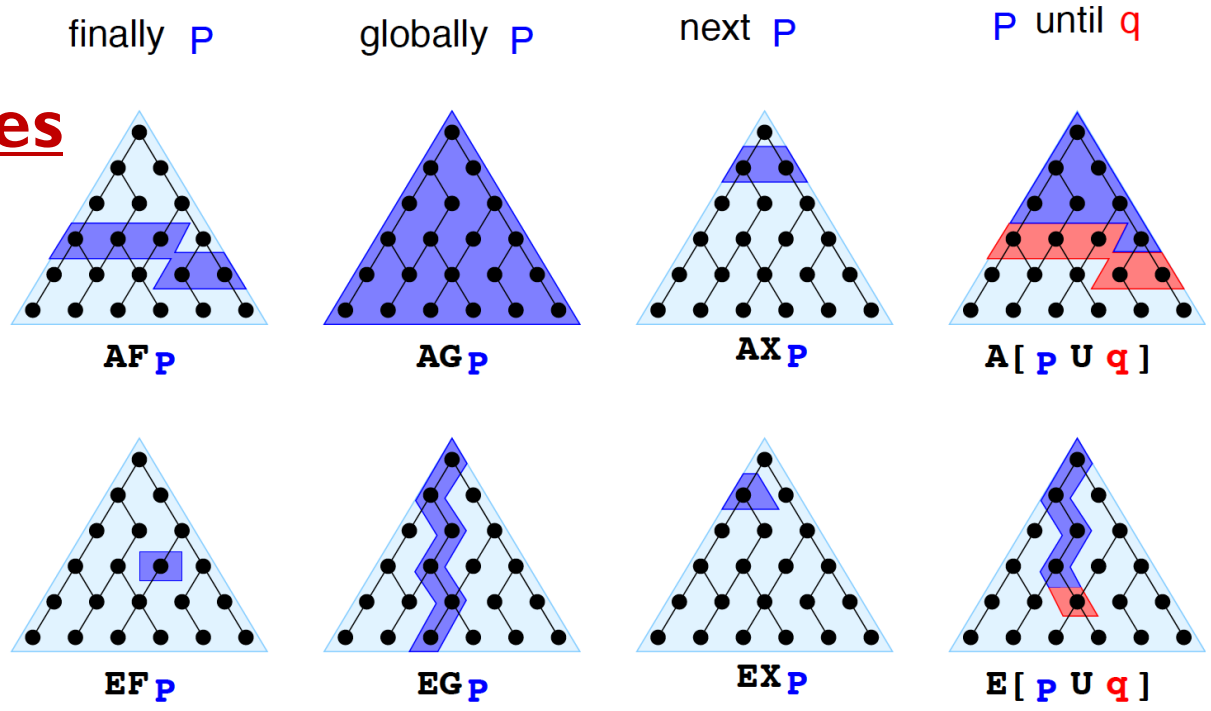
PennState

- Given a transition system T = (S, I, R, L) and an LTL property p, T satisfies p if all paths starting from all initial states I satisfy p

Computation Tree Logic

PennState

- In CTL, temporal properties use path quantifiers:
  - A : for all paths, E : there exists a path
- The semantics of CTL is defined on **states**
- Given a state s
- s $\models$ ap iff L(s, ap)
- $s_0 \models$ EX p iff $\exists$ a path $(s_0, s_1, s_2, \ldots)$. $s_1 \models p$
- $s_0 \models$ AX p iff $\forall$ paths $(s_0, s_1, s_2, \ldots)$. $s_1 \models p$
- $s_0 \models$ EG p iff $\exists$ a path $(s_0, s_1, s_2, \ldots)$. $\forall i \geq 0$. $s_i \models p$
- $s_0 \models$ AG p iff $\forall$ paths $(s_0, s_1, s_2, \ldots)$. $\forall i \geq 0$. $s_i \models p$



Examples of CTL formulas

PennState

- EF φ
  - It is possible to get to a state where φ is true
- AG AF enabled
  - A certain process is enabled infinitely often on every computation path
- AG (requested → AF acknowledged)
  - for any state, if a request ocurs, then it will eventually be acknowledged
- AG (φ → E[φ U ψ])
  - for any state, if φ holds, then there is a future where ψ eventually holds, and φ holds for all points in between
- AG (φ → EG ψ)
  - for any state, if φ holds then there is a future where ψ always holds

Linear vs. Branching Time

PennState

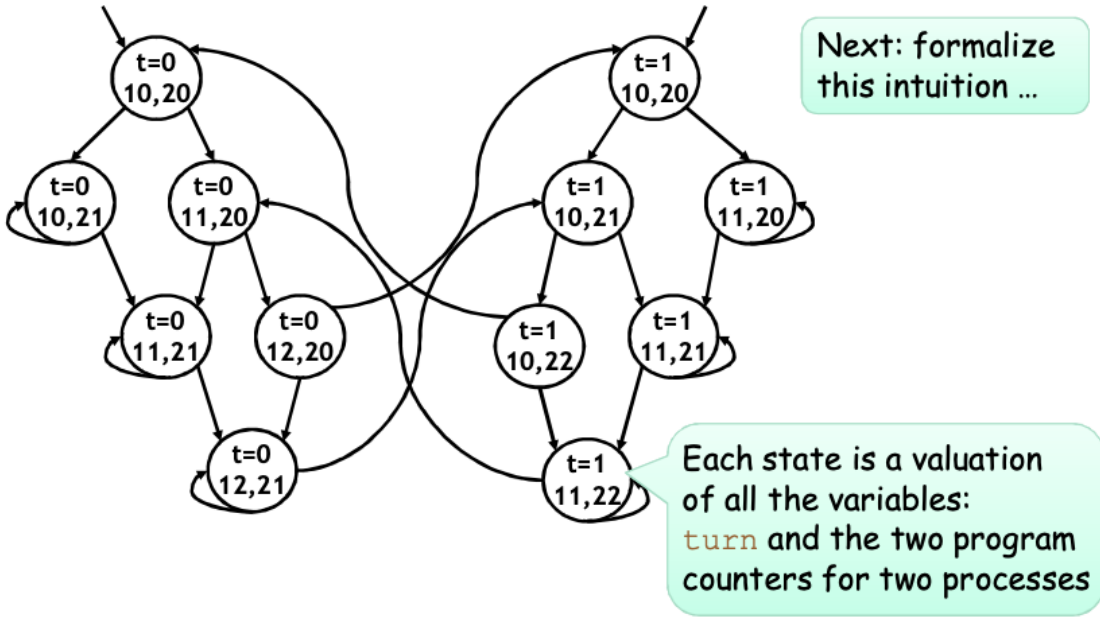
- LTL is a linear time logic
  - When determining if a path satisfies an LTL formula, we are only concerned with a single path
- CTL is a branching time logic
  - When determining if a state satisfies a CTL, formula we are concerned with multiple paths
  - In CTL the computation is instead viewed as a computation tree which contains all the paths
- The expressive powers of CTL and LTL are incomparable incomparable
  - LTL $\subseteq$ CTL*, CTL $\subseteq$ CTL*
  - Basic temporal properties can be expressed in both logics
  - Not in this lecture, sorry! (Take a class on Modal Logics)

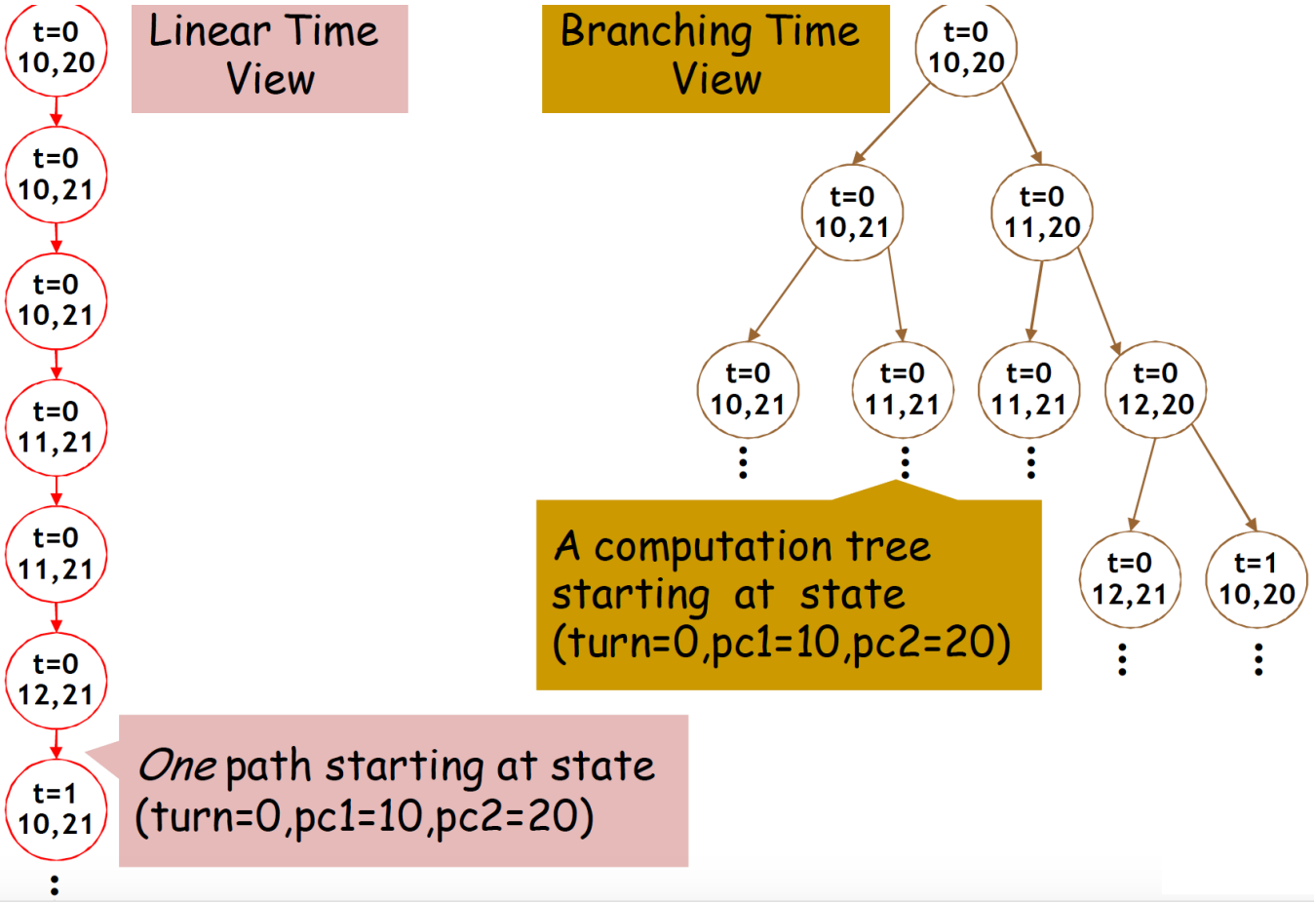
LTL vs. CTL

PennState

- Some LTL formulae cannot be translated into CTL formaulae.
  - FG s - This formula denotes the **property of stability** : in each execution of the program, s will finally be true until the end of the program (or forever if the program never stops).
  - CTL can only provide a formula that is too strict (AF AG s) or too permissive (AF EG s).
  - (AF EG s) is clearly wrong. It is not so straightforward for the first.
  - But AF AG s is erroneous. Consider a system that loops on A1, can go from A1 to B and then will go to A2 on the next move. Then the system will stay in A2 state forever. Then "the system will finally stay in a A state" is a property of the type FGs. It is obvious that this property holds on the system. However, AF AG s cannot capture this property since the opposite is true.

Linear vs. Branching Time





State Space Explosion

PennState

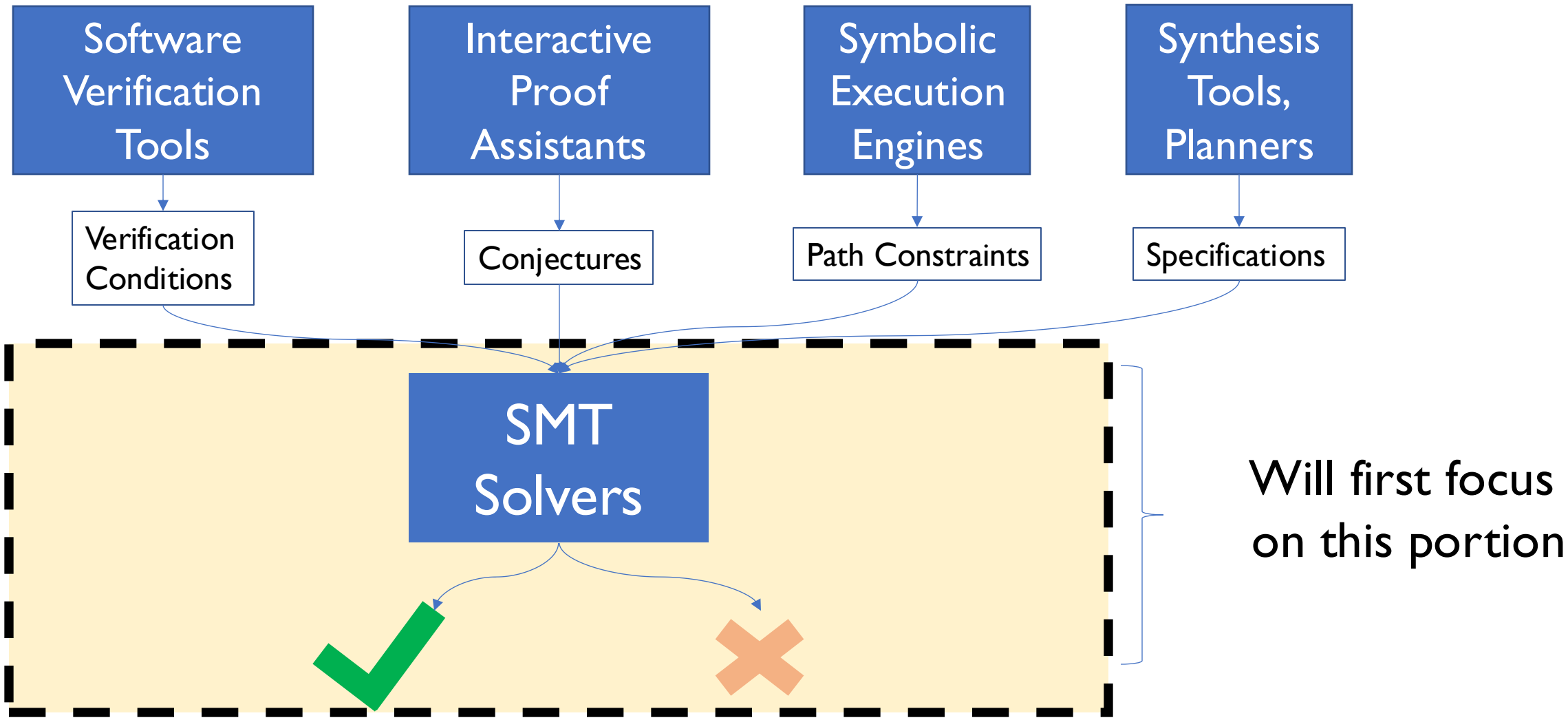
- The complexity of model checking increases linearly with respect to the size of the transition system (|S| + |R|)
- However, the size of the transition system (|S| + |R|) is exponential in the number of variables and number of concurrent processes
- This exponential increase in the state space is called the state space explosion
  - Dealing with it is one of the major challenges in model checking research

Symbolic Model Checking

PennState

- Symbolic model checking represents state sets and the transition relation as Boolean logic formulas
  - Fixed point computations manipulate sets of states rather than individual states
- Use an efficient data structure for manipulation of Boolean logic formulas
  - Binary Decision Diagrams (BDDs)
- SMV (Symbolic Model Verifier) was the first CTL model checker to use BDDs

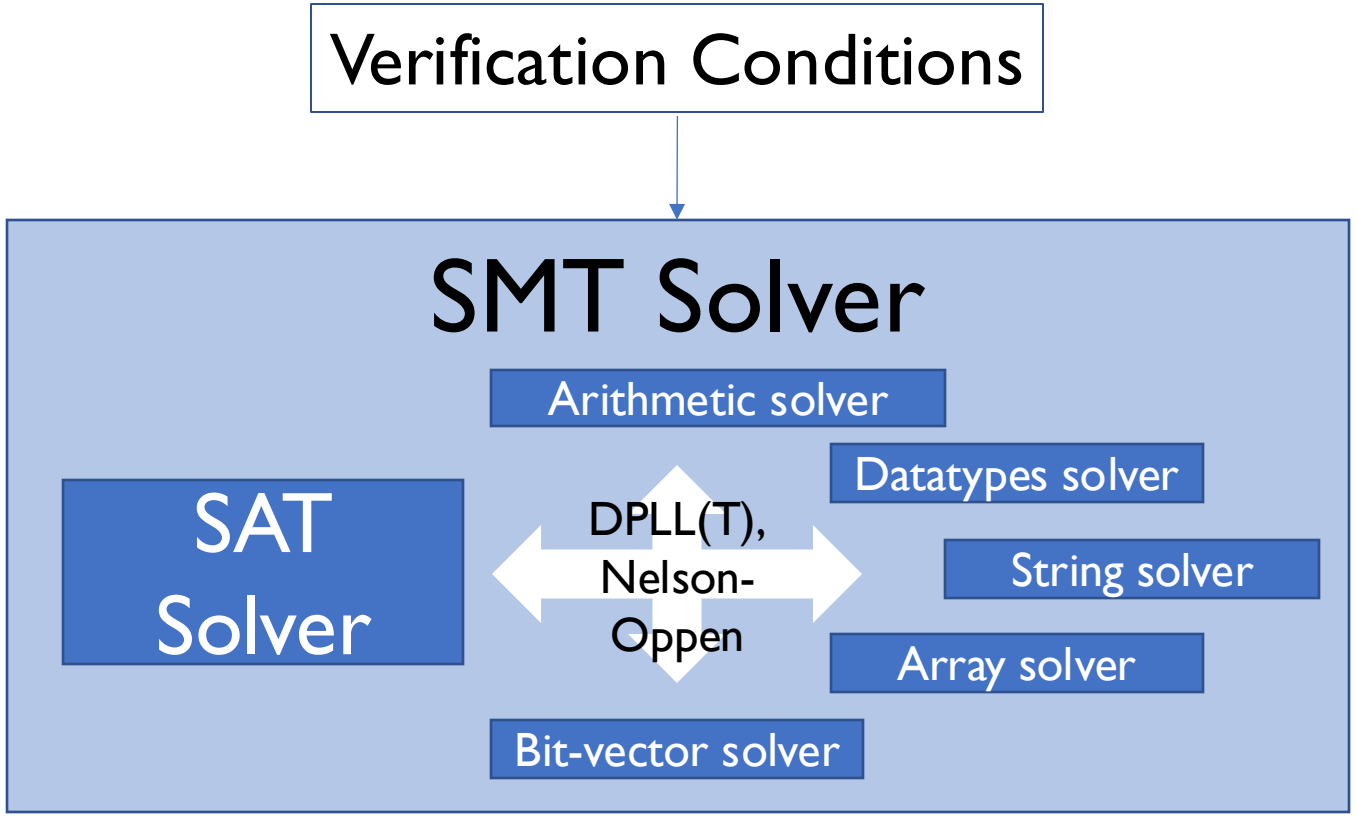
Satisfiability Modulo Theories (SMT) Solvers



SMT solvers

PennState

- Efficient tools for satisfiability and satisfiability *modulo theories*

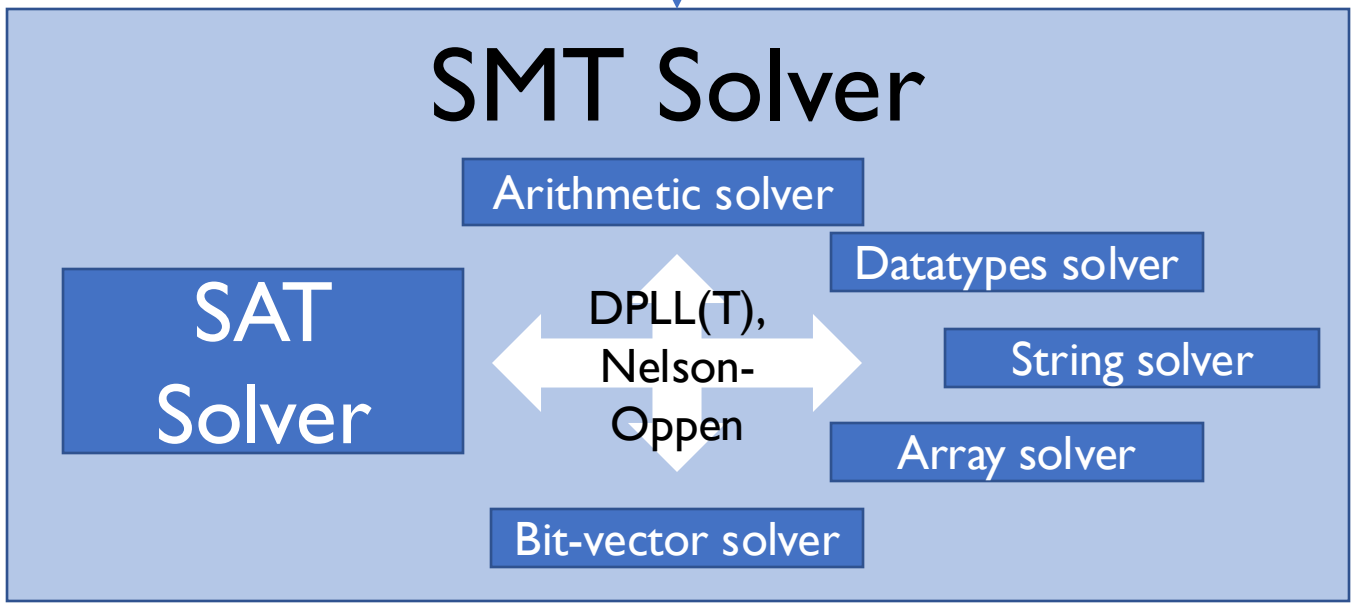


SMT solvers

PennState

- Efficient tools for satisfiability and satisfiability *modulo theories*

$$(A[x]+B[x]>0 \vee x+y>0) \wedge (cons(\text{“abc”},d_1)\neq d_2 \vee x<0)$$

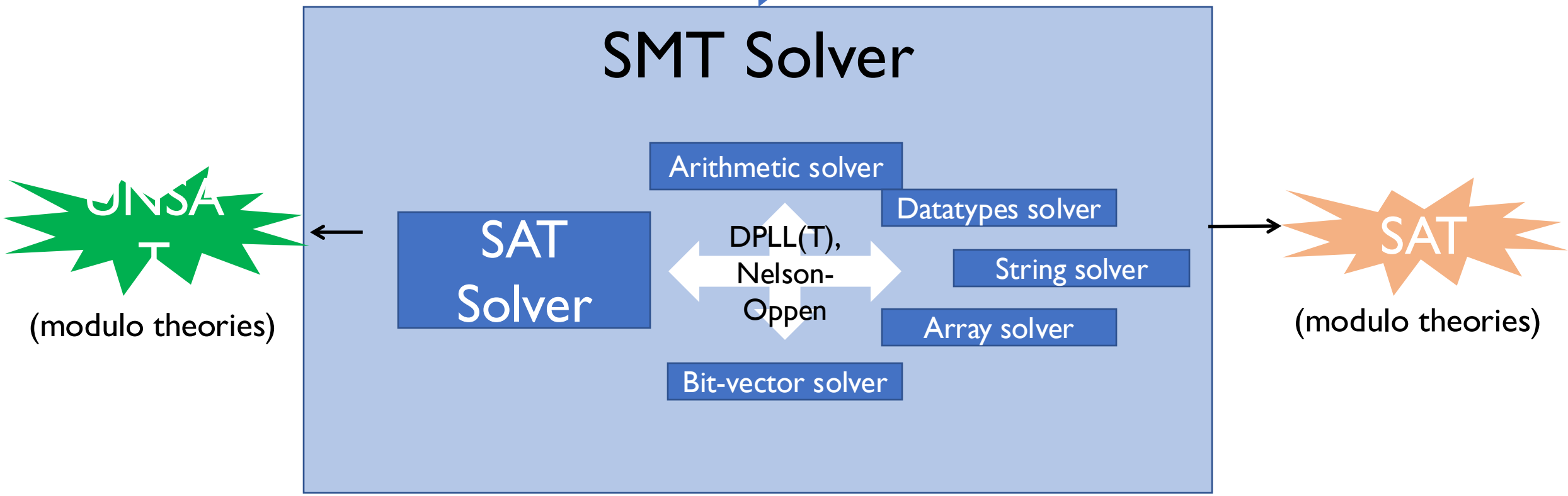


SMT solvers

PennState

- Efficient tools for satisfiability and satisfiability *modulo theories*

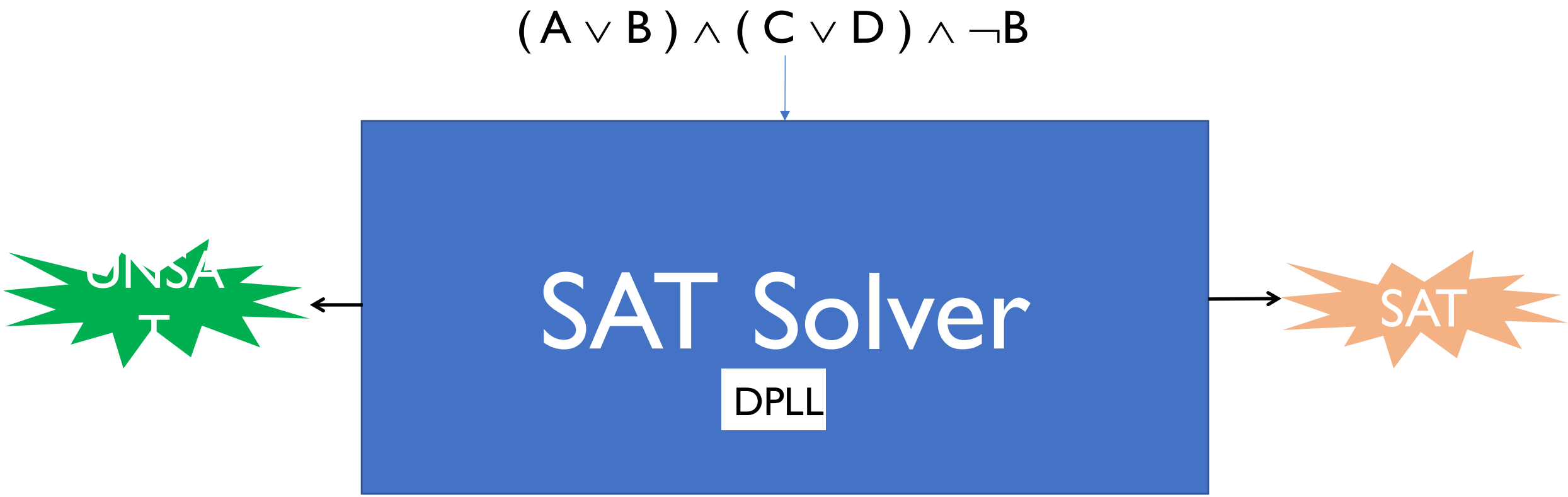
$$(A[x]+B[x]>0 \vee x+y>0) \wedge (cons(\text{“abc”},d_1)\neq d_2 \vee x<0)$$



…but first : SAT solvers

PennState

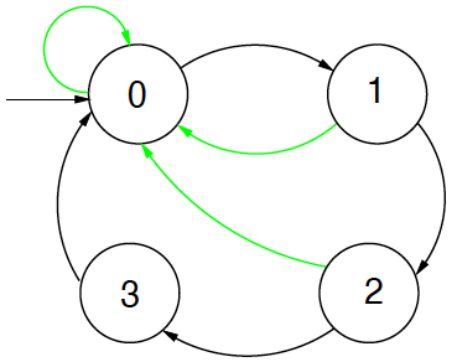
- Efficient tools for *satisfiability*



NuXmv Example: Modulo 4 counter with reset

```
MODULE main
VAR   b0    : boolean; b1    : boolean;
      reset : boolean;
ASSIGN
  init(b0) := FALSE;
  next(b0) := case  reset  : FALSE;
                    !reset : !b0;
              esac;
  init(b1) := FALSE;
  next(b1) := case  reset : FALSE;
                    TRUE  : ((!b0 & b1) |
                             (b0 & !b1));
              esac;
DEFINE out := toint(b0) + 2*toint(b1);

INVARSPEC out < 2
```

- recall:



- The invariant is false

```
nuXmv > read_model -i counter4reset.smv;
nuXmv > go; check_invar
-- invariant out < 2  is false
...
  -> State: 1.1 <-
    b0 = FALSE
    b1 = FALSE
    reset = FALSE
    out = 0
  -> State: 1.2 <-
    b0 = TRUE
    out = 1
  -> State: 1.3 <-
    b0 = FALSE
    b1 = TRUE
    out = 2
```

LTL Specifications

PennState

- Specications Examples:
  - A state in which out = 3 is eventually reached
  - LTLSPEC F out = 3
- Condition out = 0 holds until reset becomes false
  - LTLSPEC (out = 0) U (!reset)
- Every time a state with out = 2 is reached, a state with out = 3 is reached afterward
  - LTLSPEC G (out = 2 -> F out = 3)

LTL Specifications

PennState

All the previous specifications are false:

```
NuSMV > check_ltlspec
-- specification  F out = 3 is false ...
-- loop starts here --
-> State 1.1 <-
    b0 = FALSE
    b1 = FALSE
    reset = TRUE
    out = 0
-> State 1.2 <-
-- specification (out = 0 U (!reset)) is false ...
-- loop starts here --
-> State 2.1 <-
    b0 = FALSE
    b1 = FALSE
    reset = TRUE
    out = 0
-> State 2.2 <-
-- specification  G (out = 2 ->  F out = 3) is false ...
```

Q: why?

Model Programs in NuXmv

PennState

```
    void main() {
        ... // initialization of a and b
l1:     while (a!=b) {
l2:         if (a>b)
l3:             a=a-b;
            else
l4:             b=b-a;
        }
l5:     ... // GCD=a=b
    }
```

```
MODULE main()
VAR  a: 0..100;  b: 0..100;
  pc: {l1,l2,l3,l4,l5};
ASSIGN
  init(pc):=l1;
  next(pc):=
    case
      pc=l1 & a!=b    : l2;
      pc=l1 & a=b     : l5;
      pc=l2 & a>b     : l3;
      pc=l2 & a<=b    : l4;
      pc=l3 | pc=l4   : l1;
      pc=l5           : l5;
    esac;
```

```
next(a):=
  case
    pc=l3 & a > b: a - b;
    TRUE: a;
  esac;

next(b):=
  case
    pc=l4 & b >= a: b-a;
    TRUE: b;
  esac;
```

Takeaways

PennState

- A system can be modeled as a Labeled Transition System (LTS).
- Based on the expressiveness of the property, we use LTL or CTL property.
- Need to take care of state explosion problem with different types abstractions.
- Model checking is useful for testing many safety critical systems.

Thanks

PennState

Thanks to Bor-Yuh Evan Chang, Andrew Reynolds, and Patrick Trentin for some slides.