First-Order Theories & SMT Solvers

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2022/2023

Roadmap

- **First-Order Theories**
  - basic concepts; decidability issues;
  - several theories: equality, integers, linear arithmetic, arrays;
  - combining theories.
- **SMT solvers**
  - main features;
  - SMT-LIB; SMT's APIs;
  - applications.
- **SMT solvers algoritms** (extra)
  - SMT and SAT solvers integration: "eager" vs "lazy" approach;
  - the basic "lazy offline" approach and its enhancements;
  - DPLL($\mathcal{T}$) framework.

First-Order Theories

Introduction

- When judging the validity of first-order formulas we are typically interested in a particular domain of discourse, which in addition to a specific underlying vocabulary includes also properties that one expects to hold.
- For instance, in formal methods involving the integers, one is not interested in showing that the formula

$$\forall x, y.\ x < y \rightarrow x < y + y$$

is true for all possible interpretations of the symbols $<$ and $+$, but only for those interpretations in which $<$ is the usual ordering over the integers and $+$ is the addition function.

- We are not interested in validity in general but in validity with respect to some **background theory** – a logical theory that fixes the interpretations of certain predicates and function symbols.

Introduction

- Stated differently, we are often interested in moving away from pure logical validity (i.e. validity in all models) towards a more refined notion of validity restricted to a specific class of models.
- There are two ways for specifying such a class of models:
  - To provide a *set of axioms* (sentences that are expected to hold in them).
  - To pinpoint the models of interest.
- *First-order theories* provide a basis for the kind of reasoning just described.

Theories - basic concepts

Let $\mathcal{V}$ be a vocabulary of a first-order language.

- A first-order *theory* $\mathcal{T}$ is a set of $\mathcal{V}$-sentences that is closed under derivability (i.e., $\mathcal{T} \models \phi$ implies $\phi \in \mathcal{T}$).
- A *$\mathcal{T}$-structure* is a $\mathcal{V}$-structure that validates every formula of $\mathcal{T}$.
- A formula ϕ is *$\mathcal{T}$-valid* if every $\mathcal{T}$-structure validates ϕ.
- A formula ϕ is *$\mathcal{T}$-satisfiable* if some $\mathcal{T}$-structure validates ϕ.
- Two formulae ϕ and ψ are *$\mathcal{T}$-equivalent* if $\mathcal{T} \models \phi \leftrightarrow \psi$ (i.e, for every $\mathcal{T}$-structure $\mathcal{M}$, $\mathcal{M} \models \phi$ iff $\mathcal{M} \models \psi$).

Theories - basic concepts

Let $\mathcal{T}$ be a first-order theory.

- $\mathcal{T}$ is said to be a *consistent* theory if at least one $\mathcal{T}$-structure exists.
- $\mathcal{T}$ is said to be a *complete* theory if, for every $\mathcal{V}$-sentence ϕ, either $\mathcal{T} \models \phi$ or $\mathcal{T} \models \neg\phi$.
- $\mathcal{T}$ is said to be a *decidable* theory if there exists a decision procedure for checking $\mathcal{T}$-validity.
- A subset $\mathcal{A} \subseteq \mathcal{T}$ is called an *axiom set* for the theory $\mathcal{T}$, when $\mathcal{T}$ is the deductive closure of $\mathcal{A}$, i.e. $\phi \in \mathcal{T}$ iff $\mathcal{A} \models \phi$.
- A theory $\mathcal{T}$ is *finitely* (resp. *recursively*) *axiomatizable* if it possesses a finite (resp. recursive) set of axioms.
- A *fragment* of a theory is a syntactically-restricted subset of formulae of the theory.

Theories - some results

- For a given $\mathcal{V}$-structure $\mathcal{M}$, the theory $\mathrm{Th}(\mathcal{M}) = \{\phi \mid \mathcal{M} \models \phi, \text{for all } \}$ is complete.
  - These semantically defined theories are useful when one is interested in reasoning in some specific mathematical domain such as the natural numbers, rational numbers, etc.
  - Such theories may lack an axiomatisation, which seriously compromises its use in purely deductive reasoning.
- If a theory is complete and recursive axiomatizable, it can be shown to be decidable.

Theories - decidability problem

- The decidability criterion for $\mathcal{T}$-validity is crucial for mechanised reasoning in the theory $\mathcal{T}$.
- It may be necessary (or convenient) to restrict the class of formulas under consideration to a suitable *fragment* (i.e., syntactical constraint).
- The $\mathcal{T}$-validity problem in a fragment refers to the decision about whether or not $\phi \in \mathcal{T}$ when ϕ belongs to the fragment under consideration.
- A fragment of interest is the *quantifier-free (QF) fragment*.

Equality and uninterpreted functions $\mathcal{T}_\mathsf{E}$

- The **vocabulary** of the theory of *equality* $\mathcal{T}_\mathsf{E}$ consists of
  - equality $(=)$, which is the only interpreted symbol (whose meaning is defined via the axioms of $\mathcal{T}_\mathsf{E}$);
  - constant, function and predicate symbols, which are uninterpreted (except as they relate to $=$).
- **Axioms:**
  - *reflexivity:* $\forall x.\ x = x$
  - *symmetry:* $\forall x, y.\ x = y \rightarrow y = x$
  - *transitivity:* $\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$
  - *congruence for functions:* for every function $f \in \mathcal{T}$ with $\mathsf{ar}(f) = n$,
    $$\forall \overline{x}, \overline{y}.\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$
  - *congruence for predicates:* for every predicate $P \in \mathcal{T}$ with $\mathsf{ar}(P) = n$,
    $$\forall \overline{x}, \overline{y}.\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow (P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n))$$
- $\mathcal{T}_\mathsf{E}$-validity is undecidable, but efficiently decidable for the QF fragment.

Presburger arithmetic $\mathcal{T}_\mathbb{N}$

- The theory of *Presburger arithmetic* $\mathcal{T}_\mathbb{N}$ is the additive fragment of the theory of Peano.
- **Vocabulary:** $\mathcal{V}_\mathbb{N} = \{0, 1, +, =\}$
- **Axioms:**
  - axioms of $\mathcal{T}_\mathsf{E}$
  - $\forall x.\ \neg(x + 1 = 0)$ *(zero)*
  - $\forall x, y.\ x + 1 = y + 1 \rightarrow x = y$ *(successor)*
  - $\forall x.\ x + 0 = x$ *(plus zero)*
  - $\forall x, y.\ x + (y + 1) = (x + y) + 1$ *(plus successor)*
  - for every formula ϕ with $\mathsf{FV}(\phi) = \{x\}$ *(axiom schema of induction)*
    $$\phi[0/x] \wedge (\forall x.\ \phi \rightarrow \phi[x + 1/x]) \rightarrow \forall x.\ \phi$$
- $\mathcal{T}_\mathbb{N}$ is both complete and decidable (Presburger, 1929), but it has double exponential complexity.

Linear integer arithmetic $\mathcal{T}_\mathbb{Z}$

- **Vocabulary:** $\mathcal{V}_\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots, -3\cdot, -2\cdot, 2\cdot, 3\cdot, \ldots, +, -, >, =\}$
- Each symbol is interpreted with its standard mathematical meaning in $\mathbb{Z}$.
  - Note: $\ldots, -3\cdot, -2\cdot, 2\cdot, 3\cdot, \ldots$ are unary functions. For example, the intended meaning of $3 \cdot x$ is $x + x + x$, and of $-2 \cdot x$ is $-x - x$.

$\mathcal{T}_\mathbb{Z}$ and $\mathcal{T}_\mathbb{N}$ have the same expressiveness

- For every formula of $\mathcal{T}_\mathbb{Z}$ there is an equisatisfiable formula of $\mathcal{T}_\mathbb{N}$.
- For every formula of $\mathcal{T}_\mathbb{N}$ there is an equisatisfiable formula of $\mathcal{T}_\mathbb{Z}$.

Let ϕ be a formula of $\mathcal{T}_\mathbb{Z}$ and ψ a formula of $\mathcal{T}_\mathbb{N}$. ϕ and ψ are *equisatisfiable* if

$$\phi \text{ is } \mathcal{T}_\mathbb{Z}\text{-satisfiable} \quad \text{iff} \quad \psi \text{ is } \mathcal{T}_\mathbb{N}\text{-satisfiable}$$

- $\mathcal{T}_\mathbb{Z}$ is both complete and decidable via the rewriting of $\mathcal{T}_\mathbb{Z}$-formulae into $\mathcal{T}_\mathbb{N}$-formulae.

$\mathcal{T}_{\mathbb{Z}}$ versus $\mathcal{T}_{\mathbb{N}}$

Consider the $\mathcal{T}_{\mathbb{Z}}$-formula $\forall x, y. \exists z.\ y + 3x - 4 > -2z$

- For each variable v ranging over the integers, introduce two variables, v_p and v_n ranging over the non-negative integers.

$$\forall x_p, x_n, y_p, y_n. \exists z_p, z_n.\ (y_p - y_n) + 3(x_p - x_n) - 4 > -2(z_p - z_n)$$

- Eliminate negation.

$$\forall x_p, x_n, y_p, y_n. \exists z_p, z_n.\ y_p + 3x_p + 2z_p > 2z_n + y_n + 3x_n + 4$$

- Eliminate $>$ and numbers.

$$\forall x_p, x_n, y_p, y_n. \exists z_p, z_n. \exists u.\ \neg(u = 0) \ \wedge\ y_p + x_p + x_p + x_p + z_n + z_p = z_n + z_n + y_n + x_n + x_n + x_n + 1 + 1 + 1 + 1 + u$$

This is a $\mathcal{T}_{\mathbb{N}}$-formula equisatisfiable to the original one.

$\mathcal{T}_{\mathbb{N}}$ versus $\mathcal{T}_{\mathbb{Z}}$

The $\mathcal{T}_{\mathbb{N}}$-formula

$$\forall x. \exists y.\ x = y + 1$$

is equisatisfiable to the $\mathcal{T}_{\mathbb{Z}}$-formula

$$\forall x.\ x > -1 \rightarrow \exists y.\ y > -1 \wedge x = y + 1$$

To decide $\mathcal{T}_{\mathbb{Z}}$-validity for a $\mathcal{T}_{\mathbb{Z}}$-formula ϕ

- transform $\neg\phi$ to an equisatisfiable $\mathcal{T}_{\mathbb{N}}$-formula $\neg\psi$
- decide $\mathcal{T}_{\mathbb{N}}$-validity of ψ

Linear rational arithmetic $\mathcal{T}_{\mathbb{Q}}$

- The full theory of rational numbers (with addition and multiplication) is *undecidable*, since the property of being a natural number can be encoded in it.
- But the theory of *linear arithmetic over rational numbers $\mathcal{T}_{\mathbb{Q}}$* is decidable, and actually more efficiently than the corresponding theory of integers.
- **Vocabulary:** $\mathcal{V}_{\mathbb{Q}} = \{0, 1, +, -, =, \geq\}$
- **Axioms:** 10 axioms (see Manna's book)
- Rational coefficients can be expressed in $\mathcal{T}_{\mathbb{Q}}$.

The formula $\frac{5}{2}x + \frac{4}{3}y \leq 6$ can be written as the $\mathcal{T}_{\mathbb{Q}}$-formula

$$36 \geq 15x + 8y$$

- $\mathcal{T}_{\mathbb{Q}}$ is decidable and its quantifier-free fragment is efficiently decidable.

Reals $\mathcal{T}_{\mathbb{R}}$

- Surprisingly, the *theory of reals $\mathcal{T}_{\mathbb{R}}$* is decidable even in the presence of multiplication and quantifiers.
- **Vocabulary:** $\mathcal{V}_{\mathbb{R}} = \{0, 1, +, \times, -, =, \geq\}$
- **Axioms:** 17 axioms (see Manna's book)

The inclusion of multiplication allows a formula like $\exists x.\ x^2 = 3$ to be expressed (x^2 abbreviates $x \times x$). This formula should be $\mathcal{T}_{\mathbb{R}}$-valid, since the assignment $x \mapsto \sqrt{3}$ satisfies $x^2 = 3$.

- $\mathcal{T}_{\mathbb{R}}$ is decidable (Tarski, 1949). However, it has a high time complexity (doubly exponential).

Difference arithmetic

- *Difference logic* is a fragment (a sub-theory) of linear arithmetic.
- Atomic formulas have the form $x - y \leq c$, for variables x and y and constant c.
- Conjunctions of difference arithmetic inequalities can be checked very efficiently for satisfiability by searching for negative cycles in weighted directed graphs.

  Graph representation: each variable corresponds to a node, and an inequality of the form $x - y \leq c$ corresponds to an edge from y to x with weight c.
- The quantifier-free satisfiability problem is solvable in $\mathcal{O}(|V||E|)$.

Arrays $\mathcal{T}_{\mathsf{A}}$ and $\mathcal{T}_{\mathsf{A}}^{=}$

- Arrays are modeled in logic as applicative data structures.
- **Vocabulary:** $\mathcal{V}_{\mathsf{A}} = \{read, write, =\}$
- **Axioms:**
  - (reflexivity), (symmetry) and (transitivity) of $\mathcal{T}_{\mathsf{E}}$
  - $\forall a, i, j.\ i = j \to read(a, i) = read(a, j)$
  - $\forall a, i, j, v.\ i = j \to read(write(a, i, v), j) = v$
  - $\forall a, i, j, v.\ \neg(i = j) \to read(write(a, i, v), j) = read(a, j)$
- $=$ is only defined for array elements.
- $\mathcal{T}_{\mathsf{A}}^{=}$ is the theory $\mathcal{T}_{\mathsf{A}}$ plus an axiom (extensionality) to capture $=$ on arrays.
  - $\forall a, b.\ (\forall i.\ read(a, i) = read(b, i)) \leftrightarrow a = b$
- Both $\mathcal{T}_{\mathsf{A}}$ and $\mathcal{T}_{\mathsf{A}}^{=}$ are undecidable. But their quantifier-free fragments are decidable.
- Alternative fragments are often preferred that subsume the quantifier-free fragment (allowing restricted forms of index quantification).

Other theories

- *Fixed-size bit-vectors*
  - Model bit-level operations of machine words, including 2^n-modular operations (where n is the word size), shift operations, etc.
  - Decision procedures for the theory of fixed-size bit vectors often rely on appropriate encodings in propositional logic.
- *Algebraic data structures*
  - The theories describe data structures that are ubiquitous in programming like lists, stacks, binary trees, etc.
  - These theories are built around the theory of equality with uninterpreted functions, and are normally efficiently decidable for the quantifier-free fragment.

Combining theories

- In practice, the most of the formulae we want to check need a combination of theories.

Checking $x + 2 = y \to f(read(write(a, x, 3), y - 2)) = f(y - x + 1)$
involves 3 theories: equality and uninterpreted functions, arrays and arithmetic.

- Given theories $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathcal{V}_1 \cap \mathcal{V}_2 = \{=\}$, the *combined theory* $\mathcal{T}_1 \cup \mathcal{T}_2$ has vocabulary $\mathcal{V}_1 \cup \mathcal{V}_2$ and axioms $A_1 \cup A_2$

  [Nelson&Oppen, 1979] showed that if
  - satisfiability of the quantifier-free fragment of $\mathcal{T}_1$ is decidable,
  - satisfiability of the quantifier-free fragment of $\mathcal{T}_2$ is decidable, and
  - certain technical requirements are met,

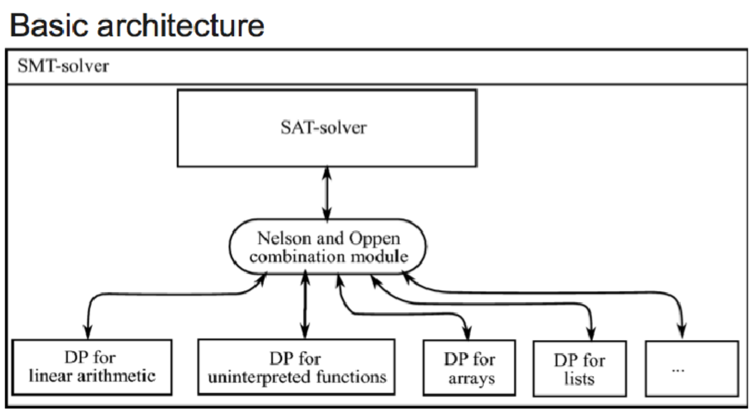
  then the satisfiability in the quantifier-free fragment of $\mathcal{T}_1 \cup \mathcal{T}_2$ is decidable.
- Most methods available are based on the Nelson-Oppen combination method.

SMT solvers

Satisfiability Modulo Theories

- The *Satisfiability Modulo Theories (SMT) problem* is a variation of the SAT problem for first-order logic, with the interpretation of symbols constrained by (a combination of) specific theories (i.e., it is the problem of determining, for a theory $\mathcal{T}$ and given a formula ϕ, whether ϕ is $\mathcal{T}$-satisfiable).
- **SMT solvers** address this problem by using as building blocks a propositional **SAT solver**, and state-of-the-art **theory solvers**
  - theories need not be finitely or even first-order axiomatizable
  - specialized inference methods are used for each theory
- The underlying logic of SMT solvers is many-sorted first-order logic with equality.

SMT-solvers basic architecture



SMT solvers

- In the last two decades, SMT procedures have undergone dramatic progress. There has been enormous improvements in efficiency and expressiveness of SMT procedures for the more commonly occurring theories.
  - The annual competition[1] for SMT procedures plays an important rule in driving progress in this area.
  - A key ingredient is SMT-LIB[2], an online resource that proposes, as a standard, a unified notation and a collection of benchmarks for performance evaluation and comparison of tools.
- Some SMT solvers: Z3, CVC4, Alt-Ergo,Yices 2, MathSAT, Boolector, ...
- Usually, SMT solvers accept input either in a proprietary format or in SMT-LIB format.

[1] http://www.smtcomp.org
[2] http://smtlib.cs.uiowa.edu

The SMT-LIB repository

- Catalog of theory declarations - semi-formal specification of theories of interest
  - A theory defines a vocabulary of sorts and functions. The meaning of the theory symbols are specified in the theory declaration.
- Catalog of logic declarations - semi-formal specification of fragments of (combinations of) theories
  - A logic consists of one or more theories, together with some restrictions on the kinds of expressions that may be used within that logic.
- Library of benchmarks
- Utility tools (parsers, converters, ...)
- Useful links (documentation, solvers, ...)
- See http://smtlib.cs.uiowa.edu

The SMT-LIB language

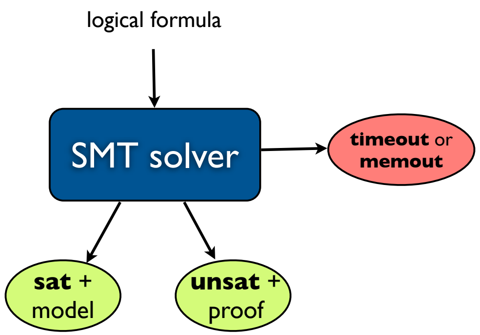
- Textual, command-based I/O format for SMT solvers.
  - Two versions: SMT-LIB 1, SMT-LIB 2 (last version: 2.6)
- Intended mostly for machine processing.
- All input to and output from a conforming solver is a sequence of one or more *S-expressions*

$$\langle\text{S-exp}\rangle \quad ::= \quad \langle\text{token}\rangle \mid (\langle\text{S-exp}\rangle^*)$$

- SMT-LIB language expresses logical statements in a many-sorted first-order logic. Each well-formed expression has a unique *sort* (type).
- Typical usage:
  - Asserting a series of logical statements, in the context of a given logic.
  - Checking their satisfiability in the logic.
  - Exploring resulting models (if SAT) or proofs (if UNSAT)

Theorem provers / SAT checkers

ϕ is valid iff $\neg\phi$ is unsatisfiable



It may happen that, for a given formula, a SMT solver returns a timeout, while another SMT solver returns a concrete answer.

SMT-LIB 2 example

```
(set-logic QF_UFLIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (distinct x y z))
(assert (> (+ x y) (* 2 z)))
(assert (>= x 0))
(assert (>= y 0))
(assert (>= z 0))
(check-sat)
(get-model)
(get-value (x y z))
```

```
sat
(model (define-fun z () Int 1)
       (define-fun y () Int 0)
       (define-fun x () Int 3) )
( (x 3) (y 0) (z 1) )
```

SMT-LIB 2 example

```
(set-logic QF_UFLIA)
(set-option :produce-unsat-cores true)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (! (distinct x y z) :named a1))
(assert (! (> (+ x y) (* 2 z)) :named a2))
(assert (! (>= x 0) :named a3))
(assert (! (>= y 0) :named a4))
(assert (! (>= z 0) :named a5))
(assert (! (>= z x) :named a6))
(assert (! (> x y) :named a7))
(assert (! (> y z) :named a8))
(check-sat)
(get-unsat-core)
```

```
unsat
(a7 a2 a6)
```


SMT-LIB 2 example

Logical encoding of the C program:

```
x = x + 1;
a[i] = x + 2;
y = a[i];
```

- We use the logic QF_AUFLIA (*quantifier-free linear formulas over the theory of integer arrays extended with free sort and function symbol*).
- An access to array `a[i]` is encoded by `(select a i)`.
- An assigment `a[i] = v` is encoded by `(store a i v)`. The result is a new array in everything equal to array `a` except in position `i` which now has the value `v`.
- Assignments such as `x = x+1` are encoded by introducing variables (e.g. `x0` and `x1`) which represent the value of `x` before and after the assignment. The logical encoding would be in this case `(= x1 (+ x0 1))`.

SMT-LIB 2 example

```
(set-logic QF_AUFLIA)
;; Logical encoding of the C program:
;;                                  x = x + 1;
;;                                  a[i] = x + 2;
;;                                  y = a[i];
(declare-const a0 (Array Int Int))
(declare-const a1 (Array Int Int))
(declare-const i0 Int)
(declare-const x0 Int)
(declare-const x1 Int)
(declare-const y1 Int)

(assert (= x1 (+ x0 1)))
(assert (= a1 (store a0 i0 (+ x1 2))))
(assert (= y1 (select a1 i0)))
;; Is it true that after the execution of the program y>x holds?
(assert (not (> y1 x1)))
(check-sat)                         ;; Yes!
```

```
unsat
```


Logical enconding of a (branching) program

The logical enconding of a (branching) program should performed as follows:

1. Convert the program into single-assignment (SA) form.
2. Convert the SA program into conditional normal form (CNF).
3. Convert each CNF statement into a logical formula.

Let us see each of these steps...

Single-assignment (SA) form

- A program is in *single-assignment (SA) form* if all its variables satisfies the following proprety:
  - *Once a variable has been used (i.e., read or assigned) it is not assigned again.*
- To encode a (branching) program into a logical formula, one should first convert the program into a SA form in wich multiple indexed version of each variable are used (a new version for each assignment made).

original program

$x = y + z;$
if $(x > y)$
  $x = x + 10$
else
  $y = x + 5;$
$z = x + y;$

$\Longrightarrow$

single-assignment form

$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$
  $x_2 = x_1 + 10;$
else
  $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ?\ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ?\ y_0 : y_1;$
$z_1 = x_3 + y_2;$

Conditional normal form

The second step for the logical enconding of the program is to convert the SA program into *conditional normal form*: a sequence of statements of the form `(if b then S)`, where `S` is an atomic statement.

- The idea is that every atomic statement is guarded by the conjunction of the conditions in the execution path leading to it.

single-assignment form

$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$
  $x_2 = x_1 + 10;$
else
  $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ?\ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ?\ y_0 : y_1;$
$z_1 = x_3 + y_2;$

$\Longrightarrow$

conditional normal form

$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$ $x_2 = x_1 + 10;$
if $(!(x_1 > y_0))$ $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ?\ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ?\ y_0 : y_1;$
$z_1 = x_3 + y_2;$

Program model in SMT-LIB 2

```
(set-logic QF_LIA)
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun y0 () Int)
(declare-fun y1 () Int)
(declare-fun y2 () Int)
(declare-fun z0 () Int)
(declare-fun z1 () Int)

(assert (= x1 (+ y0 z0)))
(assert (=> (> x1 y0) (= x2 (+ x1 10))))
(assert (=> (not (> x1 y0)) (= y1 (+ x1 5))))
(assert (= x3 (ite (> x1 y0) x2 x1)))
(assert (= y2 (ite (> x1 y0) y0 y1)))
(assert (= z1 (+ x3 y2)))
```


Checking properties of the program

original program

$x = y + z;$
if $(x > y)$ {
  $x = x + 10;$
  assert $z > 0;$
} else
  $y = x + 5;$
$z = x + y;$
assert $x == 0$

$\Longrightarrow$

single-assignment form

$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$ {
  $x_2 = x_1 + 10;$
  assert $z_0 > 0;$
} else
  $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ?\ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ?\ y_0 : y_1;$
$z_1 = x_3 + y_2;$
assert $x_3 == 0;$

Checking properties of the program

single-assignment form

$x_1 = y_0 + z_0$;
if $(x_1 > y_0)$ {
  $x_2 = x_1 + 10$;
  assert $z_0 > 0$;
} else
  $y_1 = x_1 + 5$;
$x_3 = x_1 > y_0\ ?\ x_2 : x_1$;
$y_2 = x_1 > y_0\ ?\ y_0 : y_1$;
$z_1 = x_3 + y_2$;
assert $x_3 == 0$;

$\Longrightarrow$

conditional normal form

$x_1 = y_0 + z_0$;
if $(x_1 > y_0)$ $x_2 = x_1 + 10$;
if $(x_1 > y_0)$ assert $z_0 > 0$;
if $(!(x_1 > y_0))$ $y_1 = x_1 + 5$;
$x_3 = x_1 > y_0\ ?\ x_2 : x_1$;
$y_2 = x_1 > y_0\ ?\ y_0 : y_1$;
$z_1 = x_3 + y_2$;
assert $x_3 == 0$;

Checking properties of the program

The properties to be checked are: $(x_1 > y_0 \to z_0 > 0)$ and $(x_3 = 0)$.

Let $\mathcal{C}$ be the set of formulas of the logical encoding of the program and $\mathcal{P}$ be the set of properties to be checked.

$\mathcal{C} \models_{\mathcal{T}} \bigwedge \mathcal{P}$ iff no computation path of the program violates any assert statement in it.

- If $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$ is satisfiable, a counter-example is show.
- The $\mathcal{T}$-models of $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$ (if any) correspond to the execution paths of the program that lead to an assertion violation.

Program model in SMT-LIB 2

```
(set-logic QF_LIA)
...
(assert (= x1 (+ y0 z0)))
(assert (=> (> x1 y0) (= x2 (+ x1 10))))
(assert (=> (not (> x1 y0)) (= y1 (+ x1 5))))
(assert (= x3 (ite (> x1 y0) x2 x1)))
(assert (= y2 (ite (> x1 y0) y0 y1)))
(assert (= z1 (+ x3 y2)))
(push)
(assert (not (=> (> x1 y0) (> z0 0))))
(check-sat)
(pop)
(push)
(assert (not (= x3 0)))
(check-sat)
(get-model)
```


Program model in SMT-LIB 2

```
unsat
sat
(
(define-fun y2 () Int
4)
(define-fun z0 () Int
0)
(define-fun x2 () Int
10)
(define-fun y1 () Int
4)
(define-fun x1 () Int
(- 1))
(define-fun x3 () Int
(- 1))
(define-fun y0 () Int
(- 1))
(define-fun z1 () Int
3)
)
```

Therefore $(x_1 > y_0 \to z_0 > 0)$ is valid.
Therefore $(x_3 = 0)$ doesn't hold.
Here is a conter-example:

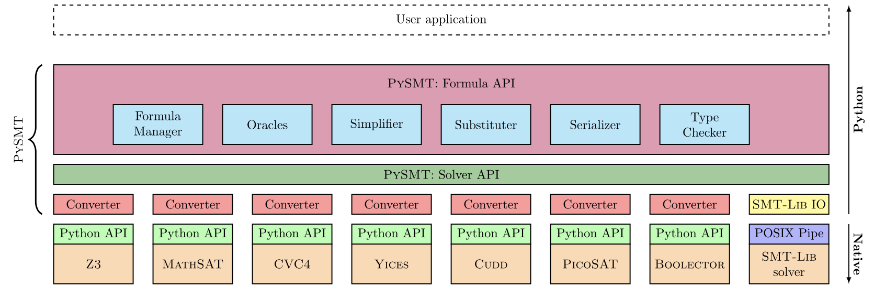
SMT solvers APIs

- Several SAT solvers have APIs for different programming languages that allow an incremental use of the solver.
- For instance, Z3Py: the Z3 Python API.

```
from z3 import *

s = Solver()
x = Int('x')
y = Int('y')
z = Int('z')

s.add(Distinct(x,y,z))
s.add(x+y>2*z)
s.add(x>=0, y>=0, z>=0)

if s.check() == sat:
    m = s.model()
    print(m)
else:
  print('There is no solution.')
```


pySMT library

- The pySMT library allows a Python program to communicate with several SMT solvers based on a common language.
- This makes it possible to code a problem independently of the SMT solver, and run the same problem with several SMT solvers.



Choosing a SMT solver

- There are many available SMT solvers:
  - some are targeted to specific theories;
  - many support SMT-LIB format;
  - many provide non-standard features.
- Features to have into account:
  - the efficiency of the solver for the targeted theories;
  - the solver's license;
  - the ways to interface with the solver;
  - the "support" (is it being actively developed?).
- See https://smtlib.cs.uiowa.edu

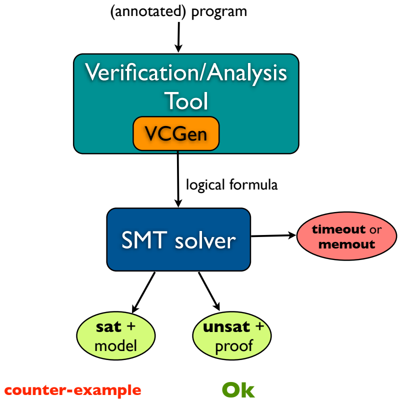
Applications

SMT solvers are the core engine of many tools for

- program analysis
- program verification
- test-cases generation
- bounded model checking of SW
- modeling
- planning and scheduling
- ...

Program verification/analysis

The general architecture of program verification/analysis tools is powered by a *Verification Conditions Generator (VCGen)* that produces verification conditions (also called "proof obligations") that are then passed to a SMT solver to be "discharged". Examples of such tools: Boogie, Why3, Frama-C, ESC/JAVA2.



Deductive program verification

Program with pre-condition, post-condition and loop invariant

PRE: $n \geq 1 \wedge i = 1 \wedge m = A[0]$
```
while (i < n)
```
INV: $i <= n \wedge \forall j. 0 \leq j < i \to m \geq A[j]$
```
  if (A[i] > m)
     m = A[i];
  i = i+1;
```
POS: $\forall j. 0 \leq j < n \to m \geq A[j]$

Proof obligations

- **Initialization:** PRE $\to$ INV
- **Preservation:** $(i < n \wedge \text{INV}) \to (A[i] > m \to \text{INV}[(i+1)/i][A[i]/m]) \wedge (A[i] \leq m \to \text{INV}[(i+1)/i])$
- **Utility:** $(\text{INV} \wedge i \geq n) \to$ POS

Scheduling

Job-shop-scheduling decision problem

- Consider n jobs.
- Each job has m tasks of varying duration that must be performed consecutively on m machines.
- The start of a new task can be delayed as long as needed in order for a machine to become available, but tasks cannot be interrupted once they are started.

Given a total maximum time max and the duration of each task, the problem consists of deciding whether there is a schedule such that the end-time of every task is less than or equal to max time units.

Two types of constraints:

- Precedence between two tasks in the same job.
- Resource: a machine cannot run two different tasks at the same time.

Scheduling

- d_{ij} - duration of the j-th task of the job i
- t_{ij} - start-time for the j-th task of the job i
- Constraints
  - Precedence: for every i, j, $t_{i\,j+1} \geq t_{ij} + d_{ij}$
  - Resource: for every $i \neq i'$, $(t_{ij} \geq t_{i'j} + d_{i'j}) \vee (t_{i'j} \geq t_{ij} + d_{ij})$
  - The start time of the first task of every job i must be greater than or equal to zero $t_{i1} \geq 0$
  - The end time of the last task must be less than or equal to max $t_{im} + d_{im} \leq max$

Find a solution for this problem

d_{ij}	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

and $max = 8$

SMT solvers algorithms (extra)

Solving SMT problems

- For a lot of theories one has (efficient) decision procedures for a limited kind of input problems: **sets (or conjunctions) of literals.**
- In practice, we do not have just sets of literals.
  - We have to deal with arbitrary Boolean combinations of literals.

How to extend theory solvers to work with arbitrary quantifier-free formulas?

- Naive solution: convert the formula in DNF and check if any of its disjuncts (which are conjunctions of literals) is $\mathcal{T}$-satisfiable.
  - In reality, this is completely impractical, since DNF conversion can yield exponentially larger formula.
- Current solution: **exploit propositional SAT technology.**

Lifting SAT technology to SMT

How to deal efficiently with boolean complex combinations of atoms in a theory?

- Two main approaches:
  - **Eager approach**
    - translate into an equisatisfiable propositional formula
    - feed it to any SAT solver
  - **Lazy approach**
    - abstract the input formula to a propositional one
    - feed it to a (DPLL-based) SAT solver
    - use a theory decision procedure to refine the formula and guide the SAT solver
- According to many empirical studies, lazy approach performs better than the eager approach.
- We will only focus on the lazy approach.

The "eager" approach

- **Methodology:**
  - Translate into an equisatisfiable propositional formula.
  - Feed it to any SAT solver.
- **Why "eager"?** Search uses all theory information from the beginning.
- **Characteristics:** Sophisticated encodings are needed for each theory.
- **Tools:** UCLID, STP, Boolector, Beaver, Spear, ...

The "lazy" approach

- **Methodology:**
  - Abstract the input formula to a propositional one.
  - Feed it to a (DPLL-based) SAT solver.
  - Use a theory decision procedure to refine the formula and guide the SAT solver.
- **Why "lazy"?** Theory information used lazily when checking $\mathcal{T}$-consistency of propositional models.
- **Characteristics:**
  - SAT solver and theory solver continuously interact.
  - Modular and flexible.
- **Tools:** Z3, CVC4,Yices 2, MathSAT, Barcelogic, ...

Boolean abstraction

- Define a bijective function prop, called *boolean abstraction function*, that maps each SMT formula to a overapproximate SAT formula.

Given a formula ψ with atoms $\{a_1, \dots, a_n\}$ and a set of propositional variables $\{P_1, \dots, P_n\}$ not occurring in ψ,

- The *abstraction mapping*, prop, from formulas over $\{a_1, \dots, a_n\}$ to propositional formulas over $\{P_1, \dots, P_n\}$, is defined as the homomorphism induced by $\text{prop}(a_i) = P_i$.
- The inverse prop^{-1} simply replaces propositional variables P_i with their associated atom a_i.

$$\psi : \quad \underbrace{g(a) = c}_{P_1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg P_2} \vee \underbrace{g(a) = d}_{P_3}) \wedge \underbrace{c \neq d}_{\neg P_4}$$

$$\text{prop}(\psi) : \quad P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$$

Boolean abstraction

$$\psi : \quad \underbrace{g(a) = c}_{P_1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg P_2} \vee \underbrace{g(a) = d}_{P_3}) \wedge \underbrace{c \neq d}_{\neg P_4}$$

$$\text{prop}(\psi) : \quad P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$$

- The boolean abstraction constructed this way over-approximates satisfiability of the formula.
  - Even if ψ is not $\mathcal{T}$-satisfiable, $\text{prop}(\psi)$ can be satisfiable.
- However, if boolean abstraction $\text{prop}(\psi)$ is unsatisfiable, then ψ is also unsatisfiable.

Boolean abstraction

For an assignment $\mathcal{A}$ of $\text{prop}(\psi)$, let the set $\Phi(\mathcal{A})$ of first-order literals be defined as follows

$$\Phi(\mathcal{A}) = \{\text{prop}^{-1}(P_i) \mid \mathcal{A}(P_i) = 1\} \cup \{\neg\text{prop}^{-1}(P_i) \mid \mathcal{A}(P_i) = 0\}$$

$$\psi : \quad \underbrace{g(a) = c}_{P_1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg P_2} \vee \underbrace{g(a) = d}_{P_3}) \wedge \underbrace{c \neq d}_{\neg P_4}$$

$$\text{prop}(\psi) : \quad P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$$

- Consider the SAT assignment for $\text{prop}(\psi)$,

$$\mathcal{A} = \{P_1 \mapsto 1, P_2 \mapsto 0, P_4 \mapsto 0\}$$

$\Phi(\mathcal{A}) = \{g(a) = c, f(g(a)) \neq f(c), c \neq d\}$ is not $\mathcal{T}$-satisfiable.

- This is because $\mathcal{T}$-atoms that may be related to each other are abstracted using different boolean variables.

The "lazy" approach (simplest version)

- Given a CNF F, **SAT-Solver**(F) returns a tuple $(r, \mathcal{A})$ where r is SAT if F is satisfiable and UNSAT otherwise, and $\mathcal{A}$ is an assignment that satisfies F if r is SAT.
- Given a set of literals S, **T-Solver**(S) returns a tuple (r, J) where r is SAT if S is $\mathcal{T}$-satisfiable and UNSAT otherwise, and J is a justification if r is UNSAT.
- Given an $\mathcal{T}$-unsatisfiable set of literals S, a *justification* (a.k.a. *unsat core*) for S is any unsatisfiable subset J of S. A justification J is *non-redundant* (or *minimal*) if there is no strict subset J' of J that is also unsatisfiable.

The "lazy" approach (simplest version)

Basic SAT and theory solver integration

```
SMT-Solver (ψ) {
  F ← prop(ψ)
  loop {
    (r, A) ← SAT-Solver(F)
    if r = UNSAT then return UNSAT
    (r, J) ← T-Solver(Φ(A))
    if r = SAT then return SAT
    C ← ⋁_{B∈J} ¬prop(B)
    F ← F ∧ C
  }
}
```

If a valuation $\mathcal{A}$ satisfying F is found, but $\Phi(\mathcal{A})$ is $\mathcal{T}$-unsatisfiable, we add to F a clause C which has the effect of excluding $\mathcal{A}$ when the SAT solver is invoked again in the next iteration. This clause is called a *"theory lemma"* or a *"theory conflict clause"*.

SMT-Solver($g(a) = c \land (f(g(a)) \neq f(c) \lor g(a) = d) \land c \neq d$)

- $F = \text{prop}(\psi) = P_1 \land (\neg P_2 \lor P_3) \land \neg P_4$
- SAT-Solver(F) = SAT, $\mathcal{A} = \{P_1 \mapsto 1, P_2 \mapsto 0, P_4 \mapsto 0\}$
- $\Phi(\mathcal{A}) = \{g(a) = c, f(g(a)) \neq f(c), c \neq d\}$
  T-Solver$(\Phi(\mathcal{A}))$ = UNSAT, $J = \{g(a) = c, f(g(a)) \neq f(c), c \neq d\}$
- $C = \neg P_1 \lor P_2 \lor P_4$

- $F = P_1 \land (\neg P_2 \lor P_3) \land \neg P_4 \land (\neg P_1 \lor P_2 \lor P_4)$
  SAT-Solver(F) = SAT, $\mathcal{A} = \{P_1 \mapsto 1, P_2 \mapsto 1, P_3 \mapsto 1, P_4 \mapsto 0\}$
- $\Phi(\mathcal{A}) = \{g(a) = c, f(g(a)) = f(c), g(a) = d, c \neq d\}$
  T-Solver$(\Phi(\mathcal{A}))$ = UNSAT, $J = \{g(a) = c, f(g(a)) = f(c), g(a) = d, c \neq d\}$
- $C = \neg P_1 \lor \neg P_2 \lor \neg P_3 \lor P_4$

- $F = P_1 \land (\neg P_2 \lor P_3) \land \neg P_4 \land (\neg P_1 \lor P_2 \lor P_4) \land (\neg P_1 \lor \neg P_2 \lor \neg P_3 \lor P_4)$
  SAT-Solver(F) = **UNSAT**

SMT-Solver($x = 3 \land (f(x + y) = f(y) \lor y = 2) \land x = y$)

- $F = \text{prop}(\psi) = P_1 \land (P_2 \lor P_3) \land P_4$
- SAT-Solver(F) = SAT, $\mathcal{A} = \{P_1 \mapsto 1, P_2 \mapsto 0, P_3 \mapsto 1, P_4 \mapsto 1\}$
- $\Phi(\mathcal{A}) = \{x = 3, f(x + y) \neq f(y), y = 2, x = y\}$
  T-Solver$(\Phi(\mathcal{A}))$ = UNSAT, $J = \{x = 3, y = 2, x = y\}$
- $C = \neg P_1 \lor \neg P_3 \lor \neg P_4$

- $F = P_1 \land (P_2 \lor P_3) \land P_4 \land (\neg P_1 \lor \neg P_3 \lor \neg P_4)$
  SAT-Solver(F) = SAT, $\mathcal{A} = \{P_1 \mapsto 1, P_2 \mapsto 1, P_3 \mapsto 0, P_4 \mapsto 1\}$
- $\Phi(\mathcal{A}) = \{x = 3, f(x + y) = f(y), y \neq 2, x = y\}$
  T-Solver$(\Phi(\mathcal{A}))$ = **SAT**

The "lazy" approach (enhancements)

Several enhancements are possible to increase efficiency of this basic algorithm:

- If $\Phi(\mathcal{A})$ is $\mathcal{T}$-unsatisfiable, identify a small justification (or unsat core) of it and add its negation as a clause.
- Check $\mathcal{T}$-satisfiability of partial assignment $\mathcal{A}$ as it grows.
- If $\Phi(\mathcal{A})$ is $\mathcal{T}$-unsatisfiable, backtrack to some point where the assignment was still $\mathcal{T}$-satisfiable.

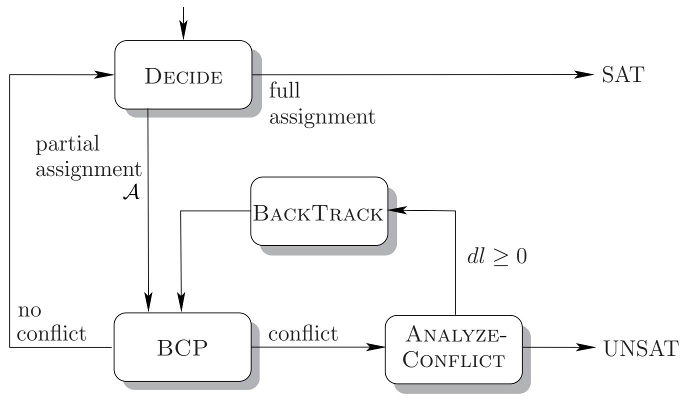
Unsat cores

- Given a $\mathcal{T}$-unsatisfiable set of literals S, a justification (a.k.a. unsat core) for S is any unsatisfiable subset J of S.
- So, the easiest justification S is the set S itself.
- However, conflict clauses obtained this way are too weak.
  - Suppose $\Phi(\mathcal{A}) = \{x = 0, x = 3, l_1, l_2, \ldots, l_{50}\}$. This set is unsat.
  - Theory conflict clause $C = \bigvee_{B \in \Phi(\mathcal{A})} \neg\mathsf{prop}(B)$ prevents that exact same assignment. But it doesn't prevent many other bad assignments involving $x = 0$ and $x = 3$.
  - In fact, there are 2^{50} unsat assignments containing $x = 0$ and $x = 3$, but C just prevents one of them!
- Efficiency can be improved if we have a more precise justification. Ideally, a minimal unsat core. This way we block many assignments using just one theory conflict clause.

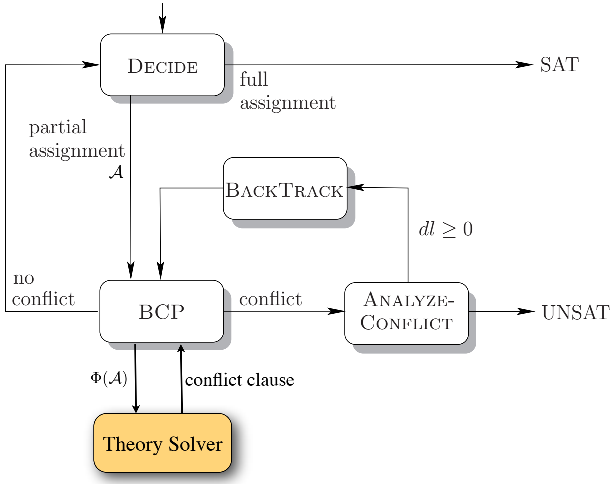
Integration with DPLL

- Lazy SMT solvers are based on the integration of a SAT solver and one (or more) theory solver(s).
- The basic architectural schema described by the SMT-solver algorithm is also called **"lazy offline"** approach, because the SAT solver is re-invoked from scratch each time an assignment is found $\mathcal{T}$-unsatisfiable.
- Some more enhancements are possible if one does not use the SAT solver as a "blackbox".
  - Check $\mathcal{T}$-satisfiability of partial assignment $\mathcal{A}$ as it grows.
  - If $\Phi(\mathcal{A})$ is $\mathcal{T}$-unsatisfiable, backtrack to some point where the assignment was still $\mathcal{T}$-satisfiable.
- To this end we need to integrate the theory solver right into the DPLL algorithm of the SAT solver. This architectural schema is called **"lazy online"** approach.
- Combination of DPLL-based SAT solver and decision procedure for conjunctive $\mathcal{T}$ formula is called *DPLL($\mathcal{T}$) framework*.

DPLL framework for SAT solvers



DPLL($\mathcal{T}$) framework for SMT solvers



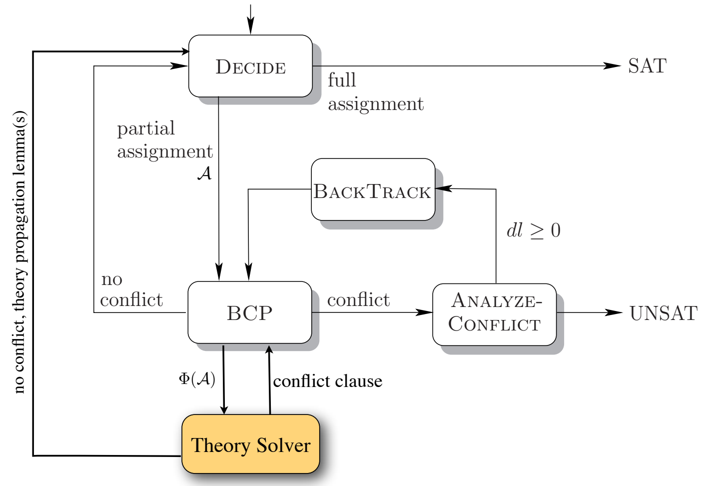
DPLL($\mathcal{T}$) framework

- Suppose SAT solver has made partial assignment $\mathcal{A}$ in Decide step and performed BCP (Boolean Constraints Propagation, i.e. in Deduce step).
- If no conflict detected, immediately invoke theory solver.
- Use theory solver to decide if $\Phi(\mathcal{A})$ is $\mathcal{T}$-unsatisfiable.
- If $\Phi(\mathcal{A})$ is $\mathcal{T}$-unsatisfiable, add the negation of its unsat core (the conflict clause) to clause database and continue doing BCP, which will detect conflict.
- As before, Analyze-Conflict decides what level to backtrack to.

DPLL($\mathcal{T}$) framework

- We can go further in the integration of the theory solver into the DPLL algorithm:
  - Theory solver can communicate which literals are implied by current partial assignment.
  - These kinds of clauses implied by theory are called *theory propagation lemmas*.
  - Adding theory propagation lemmas prevents bad assignments to boolean abstraction.

DPLL($\mathcal{T}$) framework



Main benefits of lazy approach

- Every tool does what it is good at:
  - SAT solver takes care of Boolean information.
  - Theory solver takes care of theory information.
- Modular approach:
  - SAT and theory solvers communicate via a simple API.
  - SMT for a new theory only requires new theory solver.
- Almost all competitive SMT solvers integrate theory solvers use DPLL($\mathcal{T}$) framework.

Solving SMT problems

- The theory solver works only with sets of literals.
- In practice, we need to deal not only with
  - arbitrary Boolean combinations of literals,
  - but also with formulas with quantifiers
- Some more sophisticated SMT solvers are able to handle formulas involving quantifiers. But usually one loses decidability...