3 - SMT

Satisfiability Modulo Theories

We already know SAT problems, which are NP-Complete. SAT is based on boolean logic (propositional): this makes it “efficient”; often, though, we need something more expressive (e.g. FOL, where we quantify over variables + you can encode relations).

Many applications do not require general-purpose FOL satisfiability, but we have some theory of reference that we’re interested in (fixed background theory). We won’t see quantified formulas (work on quantifier-free formulas). Restricting to a particular theory enables a more efficient solving via specialized decision procedures, especially for quantifier-free formulas.

A formula can be undecidable (e.g. Peano arithmetic), but we can restrict to decidable fragments of an undecidable theory.

This is what SMT (Satisfiability Modulo Theory) does: study satisfiability of formulas w.r.t. background theories. Born and mainly used for software analysis.

SMT extends SAT and has an orthogonal perspective w.r.t. CP. We pass through SAT, by defining some abstractions to the input formulas and use solvers that decide if the formula is good or not. No propagation, nor global constraints. The explanations for the failures is given. Chuffed solver for CP (or OR-Tools) uses SAT actually (looks for explanations for the failures).

The two main approaches are the same for a CP solver:

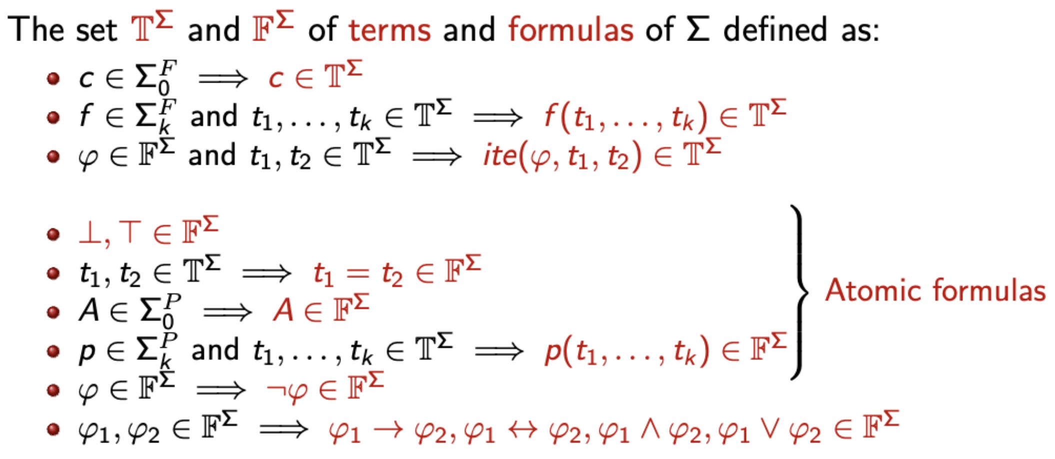
- Eager → translate eagerly into SAT and propagate using SAT. In general, it can be less efficient, e.g. for bit problems it can have a huge solve space, and it is also non-trivial.
- Lazy → the approach of chuffed and Or-tools for CP. They translate into SAT while searching, obtaining redundancies. CDCL Modulo Theory (extend DPLL with no-good learning and backjumping) and DPLL Modulo Theory (Unit Propagation and Backtracking)

SMT is more close to Constraint Logic Programming than CP (embed constraints into LP). So you can use LP Paradigms to solve constraints.

SMT Preliminaries

Assume that the logic has equality formulation.

Our signature is the set of all its function symbols and predicates denoted with their arities. 0-arity functions are constants, 0-arity predicates are propositional symbols, which can be True or False. Propositional Logic is a particular case where we don't have any functions and the only predicates we have are 0-arity. We consider only quantifier-free variables. For convenience, we treat the variables as constants.



The set $\mathbb{T}^\Sigma$ and $\mathbb{F}^\Sigma$ of terms and formulas of Σ defined as:

- $c \in \Sigma_0^F \implies c \in \mathbb{T}^\Sigma$
- $f \in \Sigma_k^F$ and $t_1, \dots, t_k \in \mathbb{T}^\Sigma \implies f(t_1, \dots, t_k) \in \mathbb{T}^\Sigma$
- $\varphi \in \mathbb{F}^\Sigma$ and $t_1, t_2 \in \mathbb{T}^\Sigma \implies ite(\varphi, t_1, t_2) \in \mathbb{T}^\Sigma$

- $\bot, \top \in \mathbb{F}^\Sigma$
- $t_1, t_2 \in \mathbb{T}^\Sigma \implies t_1 = t_2 \in \mathbb{F}^\Sigma$
- $A \in \Sigma_0^P \implies A \in \mathbb{F}^\Sigma$
- $p \in \Sigma_k^P$ and $t_1, \dots, t_k \in \mathbb{T}^\Sigma \implies p(t_1, \dots, t_k) \in \mathbb{F}^\Sigma$
- $\varphi \in \mathbb{F}^\Sigma \implies \neg\varphi \in \mathbb{F}^\Sigma$
- $\varphi_1, \varphi_2 \in \mathbb{F}^\Sigma \implies \varphi_1 \to \varphi_2, \varphi_1 \leftrightarrow \varphi_2, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2 \in \mathbb{F}^\Sigma$

Atomic formulas

A literal is either an atomic formula (atom) or the negation of one. Clause is a logical disjunction of literals. A formula is in Conjunctive Normal Form (CNF) if it is a conjunction of disjunctions.

The semantics of a formula denotes its meaning (i.e. a truth value in {true, false}, by means of a certain interpretation). Ultimate goal is see if the formula evaluates to T or F. A model for Σ is a pair $\mathcal{M} = \langle M, (.)^\mathcal{M} \rangle$ where set M is the universe of $\mathcal{M}$ and a mapping $(.)^\mathcal{M}$ such that:

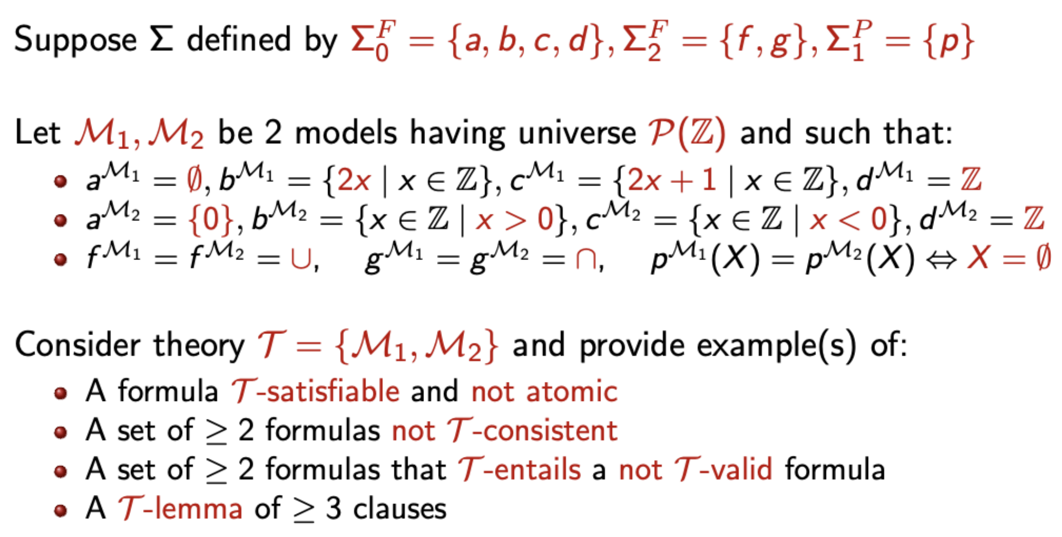
- $f^\mathcal{M} \in \{\varphi | \varphi : M^k \to M\}$ for each function $f \in \Sigma_k^F$, and in particular $c^\mathcal{M} \in M$ for each constant $c \in \Sigma_0^F$
- $p^\mathcal{M} \in \{\varphi | \varphi : M^k \to \{\text{true, false}\}\}$ for each predicate $f \in \Sigma_k^P$. In particular, $B^\mathcal{M} \in \{\text{True, False}\}$ for each proposition $B \in \Sigma_0^P$

The $(.)^\mathcal{M}$ extension to terms and formulas is called interpretation.

We say that $\mathcal{M}$ satisfies $\varphi \in \mathbb{F}^\Sigma$ if $\varphi^\mathcal{M} = \text{True}$. A Σ-Theory is a possibly infinite set $\mathcal{T}$ of Σ-models. $\varphi \in \mathbb{F}^\Sigma$ is $\mathcal{T}$-satisfiable if there exists a model $\mathcal{M} \in \mathcal{T}$ satisfying φ. $\{\varphi_1, ..., \varphi_k\} \subseteq \mathbb{F}^\Sigma$ is $\mathcal{T}$- consistent iff $\varphi_1 \wedge ... \wedge \varphi_k$ is $\mathcal{T}$- satisfiable.

$\Gamma \subseteq \mathbb{F}^\Sigma$ $\mathcal{T}$- entails φ iff every $\mathcal{M} \in \mathcal{T}$ that satisfies Γ also satisfies φ → we write $\Gamma \models_\mathcal{T} \varphi$. Γ is $\mathcal{T}$-consistent iff $\Gamma \not\models_\mathcal{T} \bot$. $\varphi \in \mathbb{F}^\Sigma$ is $\mathcal{T}$-valid iff $\emptyset \models_\mathcal{T} \varphi$ i.e. every $\mathcal{M} \in \mathcal{T}$ satisfies φ. A $\mathcal{T}$-valid clause is called theory lemma. φ is $\mathcal{T}$-consistent $\iff$ $\neg\varphi$ is not $\mathcal{T}$-valid

Exercise



Suppose Σ defined by $\Sigma_0^F = \{a, b, c, d\}, \Sigma_2^F = \{f, g\}, \Sigma_1^P = \{p\}$

Let $\mathcal{M}_1, \mathcal{M}_2$ be 2 models having universe $\mathcal{P}(\mathbb{Z})$ and such that:

- $a^{\mathcal{M}_1} = \emptyset, b^{\mathcal{M}_1} = \{2x \mid x \in \mathbb{Z}\}, c^{\mathcal{M}_1} = \{2x+1 \mid x \in \mathbb{Z}\}, d^{\mathcal{M}_1} = \mathbb{Z}$
- $a^{\mathcal{M}_2} = \{0\}, b^{\mathcal{M}_2} = \{x \in \mathbb{Z} \mid x > 0\}, c^{\mathcal{M}_2} = \{x \in \mathbb{Z} \mid x < 0\}, d^{\mathcal{M}_2} = \mathbb{Z}$
- $f^{\mathcal{M}_1} = f^{\mathcal{M}_2} = \cup, \quad g^{\mathcal{M}_1} = g^{\mathcal{M}_2} = \cap, \quad p^{\mathcal{M}_1}(X) = p^{\mathcal{M}_2}(X) \Leftrightarrow X = \emptyset$

Consider theory $\mathcal{T} = \{\mathcal{M}_1, \mathcal{M}_2\}$ and provide example(s) of:

- A formula $\mathcal{T}$-satisfiable and not atomic
- A set of ≥ 2 formulas not $\mathcal{T}$-consistent
- A set of ≥ 2 formulas that $\mathcal{T}$-entails a not $\mathcal{T}$-valid formula
- A $\mathcal{T}$-lemma of ≥ 3 clauses

Power-Set $\rightarrow$ set of all the subsets of a set

1. $p(a) \vee$ ... (is empty on the empty set works on m1 but doesn't work on m2 - as itself is atomic)
2. take a formula and its contradiction $\{p(a), \neg p(a)\}$

…

Typically, one wants to check the $\mathcal{T}$-satisfiability of formulas containing a number of quantifier-free variables (aka a consistent assignment values $\rightarrow$ variables). We assume that we can expand to a number of variables that we want. We see them as "additional constants" not belonging to Σ_0^F. More generally, given signature Σ we can consider formulas with uninterpreted symbols.

$\mathcal{T}' = \{\mathcal{M}' | \mathcal{M}'$ is an expansion of a Σ-model $\mathcal{M}\}$.

The ground $\mathcal{T}$-satisfiability problem is determining, given Σ-Theory $\mathcal{T}$, the $\mathcal{T}$-satisfiability of ground formulas over a Σ-expansion $\mathcal{T}'$. Ground formulas are formulas with no variables. As in our case constants play the role of variables, our formulas are always ground.

Because φ is $\mathcal{T}$-satisfiable $\iff$ $\neg\varphi$ is not $\mathcal{T}$-valid, the ground $\mathcal{T}$-satisfiability problem has a dual validity problem.

A theory can be defined axiomatically. A minimal set of formulas $\wedge \subseteq \mathbb{F}^\Sigma$ called axioms is given, and the corresponding theory is the set of all models of $\wedge$. An example are Peano axioms. Given Σ with constant 0 and unary function S:

- $(\forall x)\neg(S(x) = 0)$
- $(\forall x)(\forall y)\ S(x) = S(y) \rightarrow x = y$

- $(\varphi(0) \land (\forall x)(\varphi(x) \to \varphi(S(x))) \to (\forall x)\varphi(x)$ for any $\varphi \in \mathbb{F}^{\Sigma}$

By adding $+$ and $\cdot$ Peano axioms are strong enough to prove many arithmetic theorems. But not all (Gödel Incompleteness Theorems).

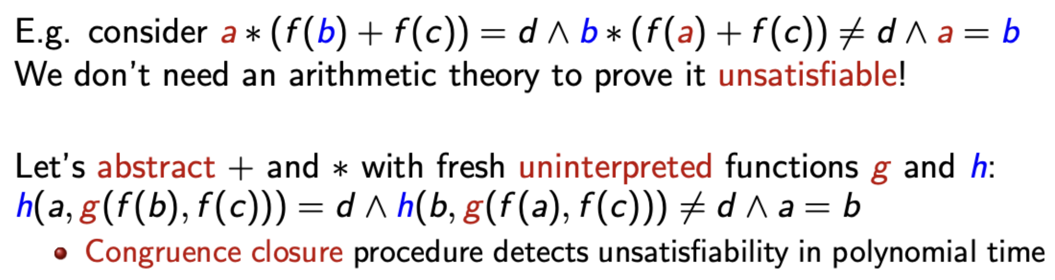
Most of SMT applications involve different data types or sorts. It may be convenient to formalize SMT problems with a many-sorted FOL having a set of sort symbols $\mathcal{S}$ (i.e. a set of types), a set of sorted variables uniquely associated with a sort $\sigma \in \mathcal{S}$, a sorted signature Σ including a set $\Sigma^{\mathcal{S}} \subseteq \mathcal{S}$ of sort symbols and corresponding semantics (sort $\equiv$ type without adding new formalism).

Some theories of interest

Uninterpreted Functions

An important baseline theory is EUF ($\mathcal{T}_{\text{EUF}}$). Equality with Uninterpreted Functions theory. There's no restrictions on how to interpret signatures. It is sometimes called Empty Theory, because its set of axioms is $\emptyset$. We use it to abstract complex or unsupported operations.

Example:


E.g. consider $a * (f(b) + f(c)) = d \land b * (f(a) + f(c)) \neq d \land a = b$
We don't need an arithmetic theory to prove it unsatisfiable!

Let's abstract $+$ and $*$ with fresh uninterpreted functions g and h:
$h(a, g(f(b), f(c))) = d \land h(b, g(f(a), f(c))) \neq d \land a = b$
- Congruence closure procedure detects unsatisfiability in polynomial time

Arithmetic

Theories over numbers are clearly very used and useful. Let $\Sigma \equiv (0, 1, +, -, \leq)$ and $\mathcal{T}_{\mathcal{Z}}$ interpreting Σ symbols in the usual way. $\mathcal{T}_{\mathcal{Z}}$ is called Presburger Arithmetic, which is decidable, both over $\mathcal{Z}$ and $\mathcal{R}$ (reals). There exist procedures to decide if a formula is true or false:

- For integers, it is $\mathbf{NP}$-Complete in general
- For reals, we have polynomial-time procedures (in the worst case, we have exponential methods, e.g. the simplex, that work fine in practice.

$\mathcal{T}_{\mathcal{Z}}$ fragments have more efficient decision procedures:

- Difference logic: x, y variables, k integer. Every atom must be $x - y \bowtie k$ with $\bowtie \in \{=, \leq\}$

- UTPVI (“unit two variable per inequality”): every atom $x \pm y \bowtie k$

Things get much harder when we introduce multiplication, as the integer case becomes undecidable and the real case doubly-exponential. Another non trivial case is the floating-point arithmetic, where $+$ and $\cdot$ are commutative but not necessarily associative nor distributive

Arrays

Arrays are homogeneous and indexed collections of elements. Let $\Sigma_{\mathcal{A}}$ be a signature with two interpreted functions read and write, as:

- $read(a, i)$ returns the value of $a[i]$
- $write(a, i, v)$ returns the array obtained by replacing $a[i]$ with v

The theory of array $\mathcal{T}_{\mathcal{A}}$ is the set of all models of these axioms:

- $(\forall a)(\forall i)(\forall v)\ \text{read(write(a,i,v),i)} = v$
- $(\forall a)(\forall i)(\forall j)(\forall v)\ i \neq j \rightarrow \text{read(write(a,i,v),j)=read(a,j)}$
- $(\forall a)(\forall a')((\forall i)\text{read(a,i)=read(a',i))} \rightarrow a = a'$ (extensionality)

The full theory is undecidable, but we have decidable fragments. It is a useful theory for SW/HW verification. In particular, arrays are often used to abstract memory locations. The main advantage is that the abstraction depends on the number of accesses to the memory rather than its size.

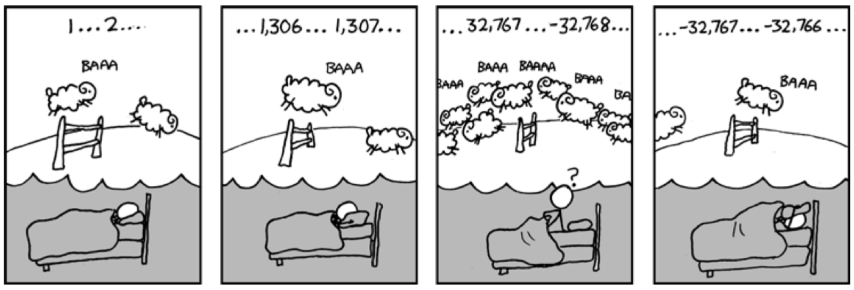
Bit-Vectors

The theory of bit-vectors $\mathcal{T}_{\mathcal{BV}}$ naturally handles verification of programs and circuits. Constants of $\mathcal{T}_{\mathcal{BV}}$ are typically vectors of bits with fixed bit-width. Typical bit-vector operations are:

- string-like operations (selections, slicing, concatenation, …)
- logical operations
- arithmetic operations

It has a straightforward reduction to SAT, whose satisfiability is NP-complete (bit-blasting).

Bit-vectors are better than integers to model machine operations. If you consider $x = 200 \land y = x + 100 \land y > x$ where x, y are unsigned 8-bit integers, this formula is valid under classical arithmetic theories, but 8-bit unsigned integer machines operate under $[0, 2^8 - 1]$ so the result would be out of range. In this case we talk of “wraparound”.



Strings

Before, string solving was typically handled with automata (which limit the expressiveness and may be inefficient) or bit-vectors (which impose a limit on length). Theory of strings can handle complex operations on unbounded-length strings natively, often in conjunction with other theories; over bounded-length strings we can also have some CP proposals.

The theory of word equations is fundamental for string solving. Fixed an alphabet $\mathcal{S}$, a word equation has form $L = R$, with L, R concatenations of (uninterpreted) constants → they concatenate string variables and strings of $\mathcal{S}^*$

The general theory is equivalent to arithmetic theory and therefore undecidable, while the quantifier-free version is decidable

In Practice

Theories in practice are not isolated, and the goal is to efficiently combine decision procedures for each theory, as efficient procedures already exist for many theories of interest.

We have decidability issues: in particular, decidability of word equations with linear length constraints is still an open problem.

SMT Solving: Eager vs Lazy Approaches

@April 21, 2023

How do we prove a given formula? Two main approaches: lazy and eager (which is similar to what we've seen on CP, when you use SAT encodings on some solvers for CP). By doing everything in advance, you have an explosion of SAT clauses and this is not always desirable. Solvers like Chuffed work this way: when you generate a failure, you add a new constraint, which means you introduce redundancy. At this point you can use heuristics to get rid of the explanations to clean up space.

SMT strongly coupled to SAT (which is not the case for CP - e.g. Gecode, which is a finite-domain solver), because it is an extension, it is built on top of SAT. So you can choose to dive into SAT eagerly or lazy.

The ultimate goal is to find if a model exists in a given theory

Eager Approaches

Given an SMT formula, you translate upfront to equisatisfiable formula in SAT. φ, φ' are equisatisfiable iff φ has a model $\mathcal{M} \iff \varphi'$ has a model $\mathcal{M}'$. All theory information is used from the beginning.

The encoding is naturally theory specific. Pros: you don't need SMT solving (you just map everything to SAT and that's the problem, you don't need any decision theory and you just then use SAT solvers and you can use the best available). The main disadvantage is that resulting SAT formulas can be huge and it is not easy to map a formula to SAT.

Now, consider EUF formula, with no restriction on the possible interpretation of a function. The first step is to remove each function/predicate and reduce each atom to equalities between constants:

- Ackermann method → replace terms with fresh constants and then add additional constraints to maintain coherence. You just have equalities, not functions anymore.
- Bryant method → replace one term and then go by If-Then-Else for the rest of the terms left.

The second step is to remove the equalities to reduce φ to Propositional Logic. Small-Domain Encoding: If the formula has n distinct uninterpreted constants , we don't care about what these values are, but just that they respect the equality constraints. Else, you can use the Direct Encoding by introducing propositional symbols and enforce transitivity.

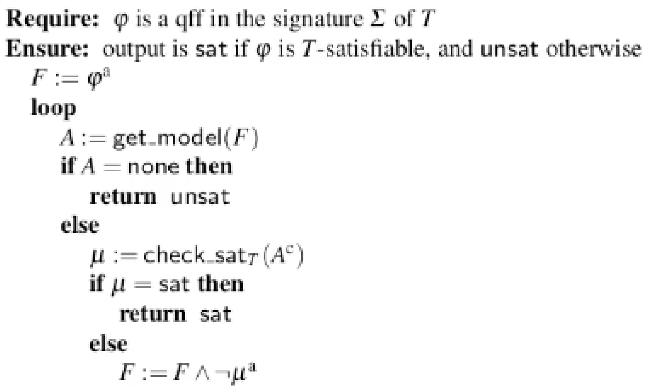
One could have more complex theories. The encoding to use always depends on the problem structure (it is an AS problem - Algorithm Selection). Direct encoding may generate larger problems solved quickly, also depending on the underlying SAT solver(s).

Lazy Approaches and CDCL($\mathcal{T}$)

Instead of compiling SMT problems to SAT, we integrate SAT solvers with theory-specific decision procedures (SMT-solving process). Check consistency using theory solvers on the boolean abstractions for the formula. SMT solvers are nowadays lazy and combine SAT solvers with theory specific solvers.

Search is not driven by theory solver but by SAT solvers (can modify by introducing theory information).

Consider the EUF formula φ. First, φ is abstracted into a SAT formula expressed in CNF. At this point I can invoke a SAT solver to find a solution for this abstraction. The model we obtain is for a boolean abstraction, therefore it might not be really true for the EUF formula (doesn't necessarily map back), so the model I obtained is sent to a $\mathcal{T}_{\mathcal{E}}$-solver to check (verify). If the model is $\mathcal{T}$-inconsistent, the process is repeated to find another model, until we find something that works, or the SAT-Solver deems the formula UNSAT.



```
Require: $\varphi$ is a qff in the signature $\Sigma$ of $T$
Ensure: output is sat if $\varphi$ is $T$-satisfiable, and unsat otherwise
  $F := \varphi^a$
  loop
    $A :=$ get_model($F$)
    if $A =$ none then
      return unsat
    else
      $\mu :=$ check_sat$_T(A^c)$
      if $\mu =$ sat then
        return sat
      else
        $F := F \wedge \neg\mu^a$
```

General approach of the SMT solver

Model can be optimized:

- Theory consistency can be dynamically checked on partial assignments, instead of always considering the full propositional model.
- Given $\mathcal{T}$-inconsistent assignment μ identify a smaller, $\mathcal{T}$-inconsistent $\eta \subseteq \mu$ and return $\neg\eta$, instead of always returning $\neg\mu$.
- Backjump to previous consistent points instead of systematic chronological backtracking

Advantages:

- everyone does what is good at → SAT care about boolean information and theory solvers only solve the conjunctions of literals.
- Modular → SAT/SMT solvers communicate via simple APIs, adding a new theory means simply add a new solver

Essentially, CDCL($\mathcal{T}$) (CDCL Modulo Theory) is the extension of the CDCL approach to SAT solving to enumerate truth values whose $\mathcal{T}$-satisfiability is checked by a $\mathcal{T}$-solver. This solver checks the consistency of conjunctions of literals, doesn't implement branching over the clauses. It can perform deduction and produce explanation of inconsistencies. It should be incremental (something that CP solvers are not) and backtrackable.

The approach can also be defined via an abstract framework based on state transitions of the form $\mu||\varphi \implies \mu'||\varphi'$. φ, φ' are $\mathcal{T}$-formulas, μ, μ' are partial boolean assignments to atoms, $\mu||\varphi$ is a state. A sequence of transitions is called derivation. If from the initial state

$\emptyset \| \varphi$ we soundly derive $\mu \| \varphi$ where μ is a complete assignment of φ, then $\mu \models_{\mathcal{T}} \varphi$ ($\mathcal{T}$-consistent)

Theory Propagation ($\mathcal{T}$-propagation) = guide the search via deduction of unassigned literals and makes the lazy approach “less lazy”. Sometimes redudancy can be good to reduce the size of the search tree.

General rule:

$$\text{if } \mu \models_{\mathcal{T}} \ell \text{ and } \ell \text{ or } \neg\ell \text{ occurs in } \varphi, \text{ and neither } \ell \text{ nor } \neg\ell \text{ occur in } \mu, \text{ then:}$$
$$\mu \| \varphi \implies \mu \cup \{\ell\} \| \varphi$$

```
1:  function $\mathcal{T}$-CDCL($\varphi$ : $\mathcal{T}$-formula, $\mu$ : $\mathcal{T}$-assignment)
2:      if preProcess($\varphi, \mu$) = Conflict then return $\bot$          ▷ Pre-processing
3:      $\varphi^p \leftarrow \mathcal{T}2\mathcal{B}(\varphi)$;   $\mu^p \leftarrow \mathcal{T}2\mathcal{B}(\mu)$          ▷ Boolean abstractions
4:      while true do
5:          $\ell \leftarrow$ decideNextLit($\varphi^p, \mu^p$)          ▷ Next literal to split on
6:          while true do
7:              status $\leftarrow$ propagate($\varphi^p, \mu^p, \ell$)          ▷ Unit/$\mathcal{T}$-propagation
8:              if status = SAT then return $\mathcal{B}2\mathcal{T}(\mu^p)$          ▷ $\varphi$ satisfiable
9:              else if status = UNSAT then
10:                 level $\leftarrow$ analyzeConflict($\varphi^p, \mu^p$)          ▷ Conflict analysis
11:                 if level = 0 then return $\bot$          ▷ $\varphi$ unsatisfiable
12:                 backjump(level, $\varphi^p, \mu^p$)
13:             else break
14:         end while
15:     end while
```

- `preProcess` → to see if you can easily find a conflict. Simplifies/updates φ and μ. It is a boolean pre-processing + $\mathcal{T}$-specific rewriting.
- $\mathcal{T}2\mathcal{B}$ maps a $\mathcal{T}$-formula to its Boolean abstraction
- `decideNextLit` → select the next literal to split on according to given heuristics as in standard DPLL (but might exploit $\mathcal{T}$-information)
- `propagate` → iteratively applies unit propagation, $\mathcal{T}$-consistency checks and $\mathcal{T}$-propagation. It updates φ^P, μ^P and returns either: SAT, UNSAT, UNKNOWN if no more literals can be deduced
- `analyzeConflict` → performs conflict analysis if UNSAT is returned

- If conflict detected by Boolean propagation, a Boolean conflict set η^P is produced (CDCL); if conflict detected by $\mathcal{T}$-propagation, a theory conflict set η is produced and abstracted to η^P
- Then φ^Pupdated with $\neg\eta^P \wedge \varphi^P$ and a decision level is returned (important!): if 0, φ is deemed UNSAT. Otherwise, backjump to a specific level:
    - Standard DPLL does chronological backtracking, back to last level
    - CDCL($\mathcal{T}$) typically jumps to the highers point in the stack where one $\ell^P \in \eta^P$ is not assigned, and propagates $\neg\ell^P \wedge \neg\eta^P$.

In CDCL, we traverse backward the implication graph from the conflict until a standard condition is met. We use these information to return a conflict set. This condition might be the 1UIP (closest Unique Implication Point to the conflict, where UIP is the node traversed by all paths from the current decision node to the conflict).

Summarizing, the main extensions of CDCL($\mathcal{T}$) w.r.t. CDCL are:

- $\mathcal{T}$-propagation in addition to Boolean propagation
- $\mathcal{T}$-conflicts - conflict propagation - (also mixed) in addition to Boolean conflicts
- Cheap operations are computed first: we can delay the $\mathcal{T}$-solver.

Theory Solvers, Combinations and Extensions

@April 27, 2023

Theory Solvers

In its simplest form, a $\mathcal{T}$-solver (Theory Solver) takes as input a conjunction of $\mathcal{T}$-literals μ and decides whether μ is $\mathcal{T}$-satisfiable. We can see the SMT solver as a “collection” of Theory solvers.

The crucial features for state-of-the-art $\mathcal{T}$-solvers are:

- Incrementality → If there is new information you don’t need to restart from scratch (as it happens in CP solvers). When restarting the CP solver, we have both advantages and disadvantages: e.g. it can get us out of a soon-to-be-failing situation, it’s a recovery mechanism from bad solutions; by restarting we might be able to improve the process. For problems where the search space is huge, restarting can be worse (note also that restarting on Gecode is different than restarting from Chuffed - Chuffed uses SAT explanations to explain failures, so when we stop and restart we use these explanations to guide the search. Gecode instead does not record any explanations, you lose all the information you learned during the run). In SMT we can assert new formulas without restarting the search. The $\mathcal{T}$-solver incrementally checks new literals with a cost proportional to the size of the addition; it avoids restarting the computation from scratch.

- Backtrackability → any solver must be able to recover from a failure; the solver can undo steps and return to a previous state efficiently
- Literal Deduction → (e.g. in EUF Theory we can do deduction about equalities, same for LRA) the solver can perform deductions of literals not yet assigned in the input formula ($\mathcal{T}$-propagation) - CP solver propagation removes inconsistent values (so there's a form of propagation)
- Explanation Generation → when a conflict involving the literal ℓ is found, it is necessary to have a (possibly short) explanation $\ell_1 \wedge \ell_2 \wedge ... \wedge \ell_n \rightarrow \ell$ (in form of disjunction of clauses - CNF form) for performing conflict analysis and determine how far to backtrack (e.g. by traversing the Implication Graph). This is natural on SMT solvers as they're based on SAT (key feature), not in all CP solvers.

Different theories often share common features. There are general frameworks to solve different theories(e.g. Shostak's method, case splitting or layered solvers). As one could expect, $\mathcal{T}$-solvers are specialized according to an underlying theory $\mathcal{T}$.

EUF Theory

Remember: Equality with Uninterpreted Functions. This is the simplest theory: all Σ-models (possible models) of a given signature Σ, also called Empty Theory as we have no constraint on how to represent the models. We can use the properties and the congruence to declare satisfiability. The conjunctions of literals $\mathcal{T}_\varepsilon$ are decided in polynomial time with congruence closure (must be preserved) procedures:

- Add fresh c and replace each $p(t_1, ..., t_k)$ with $f_p(t_1, ..., t_k) = c$
- Partition - split in a number of subsets in which the union of the subsets gives the original set and the pairwise intersection is empty - the set of input literals into the subsets of equalities (E) and disequalities (D)
- Let E^* be the congruence closure of E, i.e. the smallest equivalence relation (relation which is reflexive, symmetric and transitive - any equivalence relation defines a partition: equivalence class, i.e. the set of all items equivalent) $\equiv_E$ over the terms of E such that:
    - $t_1 = t_2 \in E \rightarrow t_1 \equiv_E t_2$
    - For each $f(s_1, ..., s_k), f(t_1, ..., t_k)$ occurring in E, if $s_i \equiv_E t_i$ for each $i \in \{1, ..., k\}$, then $f(s_1, ..., s_k) \equiv_E f(t_1, ..., t_k)$ (congruence property)

- Then, Φ satisfiable $\iff$ for each $t_1 \neq t_2 \in D, t_1 \not\equiv_E t_2$

Standard algorithms use a Directed Acyclic Graph to represent terms, and union-find data structures (like merge-find or disjoint-set) for the classes of $\equiv_E$ (union-find is used to represent sets of disjoint sets).

LRA Theory

Consider the Linear Real Arithmetic theory, whose signature is $\Sigma_{LRA} = (\mathbb{Q}, +, -, *, \leq)$ and supports linear multiplication (first-order) only.

We can use different approaches to decide LRA-literals, like for example the Fourier-Motzkin elimination:

- Replace $t_1 \neq t_2$ with $t_1 \vee t_2$ and $t_1 \leq t_2$ with $t_1 < t_2 \vee t_1 = t_2$ (case splitting)
- Eliminate equalities and apply Fourier-Motzkin elimination to all variables to determine its satisfiablity

This is though not practical for large sets of constraints, where Simplex method is preferable.

LIA Theory

Consider the Linear Integer Arithmetic theory (harder than the previous), whose signature is $\Sigma_{LIA} = (\mathbb{Z}, +, -, *, \leq)\Sigma_{LRA} = (\mathbb{Q}, +, -, *, \leq)$ and supports linear multiplication only.

- if it is non-linear, it is undecidable (Peano Arithmetic)
- if it is fully quantified, Presburger Arithmetic
- if it is quantifier-free, we have different decision procedures we can use

We can indeed apply methods like Fourier-Motzkin, but in general, Simplex+branch&bound (case in which the solution we found is not an integer)/cut (of unfeasible parts) works generally better.

We have also methods for LIRA (Integer + Real Arithmetic) and NLA (non-linear arithmetic)

Difference Logic

Difference Logic theory has atomic formulas of the form $x - y \leq k$ (at most two variables and a constant), with x, y variables and k constant. We can rewrite constraints of the type $x - y \bowtie k$ with $\bowtie \in \{=, \neq, <, \geq, >\}$. Unary constraints $x \leq k$ can be rewritten into $x + z_0 \leq k$ by enforcing $z_0 = 0$ in any satisfying assignment.

If we allow $\neq$ and the domain is $\mathbb{Z}$, deciding satisfiability of DL formulas is $\mathbf{NP}$-hard, e.g. it is as hard as k-coloring problem, which states:

If we have k (≥ 3 to make the problem $\mathbf{NP}$-hard) colors available, can we color a graph s.t. adjacent nodes have different colors?

Formally, given a graph (V, E) and $k \in \mathbb{N}$, does it exist a function $c : V \to \{1, ..., k\}$ s.t. for each $(i, j) \in E$ we have $c(i) \neq c(j)$?

Any k-coloring instance can be mapped to a DL formula with $|V|$ variables, $|E|$ disequalities $x_i \neq x_j$ for each $(i, j) \in E$ and $2|V|$ disequalities $1 \leq x_i \leq k$. If we can decide the DL formula in polynomial time, we can solve any problem in $\mathbf{NP}$ in polynomial time.

From DL literals in Φ we can get a directed weighted graph $\mathcal{G}_\Phi$ with a node for each variable occurring in Φ and a weighted edge $x \xrightarrow{k} y$ for each $x - y \leq k \in \Phi$.

<ins>**Theorem**</ins>

Φ is inconsistent $\iff$ $\mathcal{G}_\Phi$ has a negative cycle

Negative loops can be detected with Bellman-Ford in $O(|V||E|)$ by adding to V a source vertex x_0 and an edge $x_0 \xrightarrow{0} x$ for each $x \in E$. Note that other more efficient variants exist.

In general, negative loops denote inconsistency explanations, which are not minimal in general.

Theory propagations are computed from consistent graphs: if there is a path between x and y with total weight k, we can deduce $x - y \leq k$:

> If $x \xrightarrow{k_1} x_1 \xrightarrow{k_2} ... \xrightarrow{k_n} y$ the total weight is $k = \sum_{i=1}^{n} k_i$ hence we can say $x - x_1 \leq k_1, ..., x_n - y \leq k_n$, so $(x - x_1) + ... + (x_n - y) \leq \sum_{i=1}^{n} k_i = k$, thus because many of the sums cancel themselves we get $x - y \leq k$

Other theories

For each theory I can define a decision procedure, that may be not complete or solve just a fragment.

- Bit-vectors → formulas are first simplified, then encoded into SAT formulas (bit-blasting)
- Arrays → theory axioms instantiations + congruence closure and optimizations
- Multi-sets

- Strings
- Floating points
- …

Combining Theories

Often, SMT formulas contain atoms from disparate theories. In particular, software verifications applications can generate constraints over several data types. So the question is: given $\mathcal{T}_i$-solvers for theories $\mathcal{T}_1, ..., \mathcal{T}_n$, can we combine them to get a solver for $\bigcup_i \mathcal{T}_i$ (take advantage of what we already have)?

We can define a new decision procedure for the combination of theories, or we could use a different procedure to combine them, which involves:

1. Purification: each literal must belong to only one theory; therefore, we will need to use fresh constants. To merge the models, solvers must agree on equalities between shared constants, a.k.a. interface equalities, which can be done by entailing and exchanging interface equalities
2. Satisfiability check and equalities exchange (to check the coherence of the values of the shared constants, as just checking the individual theories is not enough)

Nelson-Oppen Procedure (Convex Case)

Let Σ_1, Σ_2 be signatures and $\mathcal{T}_1, \mathcal{T}_2$ their theories. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are (preconditions):

- signatures are disjoint → $\Sigma_1 \cap \Sigma_2 = \emptyset$
- stably-infinite → a Σ-theory $\mathcal{T}$ of sort σ is stably infinite if every $\mathcal{T}$-satisfiable Σ-formula has a model interpreting σ as an infinite set.
- convex → for each set of $\mathcal{T}_i$-literals S we have that $\mathcal{S} \models_{\mathcal{T}_i} a_1 = b_1 \vee ... \vee a_n = b_n$ implies that $S \models a_k = b_k$ for some $k \in \{1, ..., n\}$ → if from $\mathcal{S}$ we can deduce a disjunction of equalities, then one of these equalities is for sure satisfiable. It is non-trivial.

Then we can check the $(T_1 \cup T_2)$-satisfiability with the deterministic Nelson-Oppen algorithm

> Let S be a $(\mathcal{T}_1 \cup \mathcal{T}_2)$-formula (union of two theories) and E the set of interface equalities between S_1 and S_2. Steps:
>
> 1. Purify $\mathcal{S}$ and split (specifically, we make a partition) it into $\mathcal{S}_1$ and $\mathcal{S}_2$, where $\mathcal{S}_i$ contain $\mathcal{T}_i$ literals only

2. If $\mathcal{S}_1 \models_{\mathcal{T}_1} \perp$ then return UNSAT
3. If $\mathcal{S}_2 \models_{\mathcal{T}_2} \perp$ then return UNSAT
4. If $\mathcal{S}_1 \models_{\mathcal{T}_1} x = y$ with $x = y \in E - \mathcal{S}_2$, then $\mathcal{S}_2 \leftarrow \mathcal{S}_2 \cup \{x = y\}$ and go to 3 (iterative step)
5. If $\mathcal{S}_2 \models_{\mathcal{T}_2} x = y$ with $x = y \in E - \mathcal{S}_1$, then $\mathcal{S}_1 \leftarrow \mathcal{S}_1 \cup \{x = y\}$ and go to 2 (iterative step)
6. return SAT

Nelson-Oppen Procedure (Non-Convex Case)

Deterministic Nelson-Oppen procedure doesn't work if $\mathcal{T}_i$ is not convex. However, there is a non-deterministic Nelson-Oppen procedure that also works on non-convex theories (still need disjoint and stably-infinite theories). It works through arrangements of shared constants, basically doing case splitting $x = y \vee x \neq y$ between pairs of shared constants. The complexity is exponential, worst-case

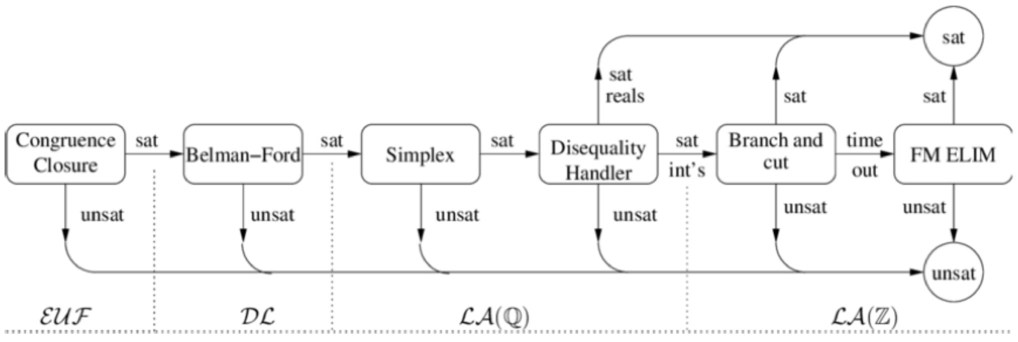
SMT Extensions

@April 28, 2023

There are several extensions and enhancements to the SMT framework seen so far.

- Quantified formulas
- Layered Solvers

  We follow a hierarchical approach: the problem is stratified in layers $L_1, L_2, ...$ of increasing complexity; if the solver finds conflict at L_i, use conflict to prune the search space, otherwise activate the next, more expensive solver. If a partial assignment is found, it is passed to L_{i+1}.



- Case splitting

Sometimes, consistency checking might require case reasoning: a complete $\mathcal{T}_{\mathcal{A}}$-solver can “internally” detect inconsistencies via case splitting and backtracking, or, as an alternative, lift (send) this case reasoning from $\mathcal{T}$-solver to SAT solver.

- On-Demand Solvers

  $\mathcal{T}$-solver encodes the splits as clauses and sends them to SAT engine. As a pro: the $\mathcal{T}$-solvers are not necessarily complete, case reasoning coordinated by SAT solver, can be generalized to combinations of theories. As a con: we have potential termination issues, specific criteria must be met to ensure soundness/completeness, performance issues.

- Optimization Modulo Theory

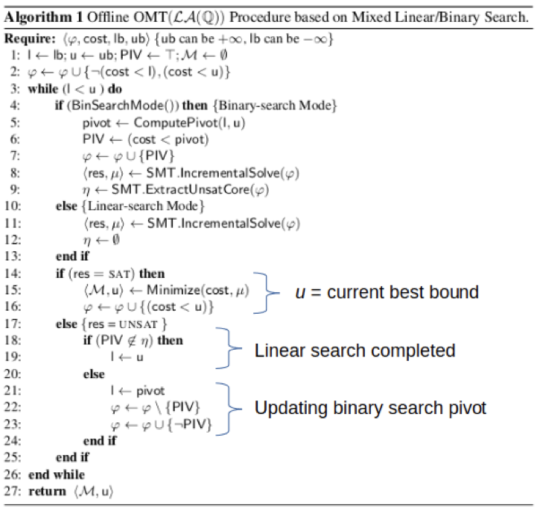
  OMT is an extension of SMT where we need to find a model for an input formula φ which is optimal w.r.t. an objective function f_{obj}. φ refers to a theory $\mathcal{T}_{\preceq} \cup \mathcal{T}_1 \cup ... \cup \mathcal{T}_n$ where $\mathcal{T}_{\preceq}$ contains a predicate $\preceq$ representing a total order (we need to have a means to compare solutions), and $\bigcup_{i=1}^{n} \mathcal{T}_i$ might be empty. The goal is to find a model $\mathcal{M}$ s.t. $\varphi^{\mathcal{M}} =$ true and $f_{obj}^{\mathcal{M}}$ is minimal according to $\preceq$ (typically the ≤ predicate over integers and reals).

  A total order is a partial order for which, if you take two generic elements, call them x and y, either $x \leq y$ or $y \leq x$. Each couple of elements is always comparable. A partial order is reflexive and antisymmetric, but not all couples of elements are comparable (e.g. the $\leq$ relation between integers is a total order, the subset relation is reflexive, antisymmetric, transitive, so you can impose an ordering, but it is not total)

  OMT is younger than SMT (2006, Nieuwenhuis&Oliveras). Some state-of-the-art SMT solvers natively provide OMT capabilities (e.g. Z3)

  Linear search (simplest approach to optimize SAT problem) repeatedly narrows the cost domain $[l_i, u_i)$ by adding cost < c_i if a model with cost c_i is found at the i-th iteration. If no model is found, c_i is the minimum cost

  Binary search picks a pivot $p_i \in [l_i, u_i)$ and adds cost $< p_i$, we assume $p_i \simeq (l_i + u_i)/2$. If no model is found, look into $[p_i, u_i)$. On average is more efficient, though we must know the cost bounds.


Algorithm 1 Offline OMT($\mathcal{LA}(\mathbb{Q})$) Procedure based on Mixed Linear/Binary Search.

```
Require: $\langle \varphi, cost, lb, ub \rangle$ {ub can be $+\infty$, lb can be $-\infty$}
$l \leftarrow lb$; $u \leftarrow ub$; $PIV \leftarrow \top$; $\mathcal{M} \leftarrow \emptyset$
$\varphi \leftarrow \varphi \cup \{\neg(cost < l), (cost < u)\}$
while ($l < u$) do
    if (BinSearchMode()) then {Binary-search Mode}
        pivot $\leftarrow$ ComputePivot(l, u)
        PIV $\leftarrow$ (cost < pivot)
        $\varphi \leftarrow \varphi \cup \{PIV\}$
        $\langle res, \mu \rangle \leftarrow$ SMT.IncrementalSolve($\varphi$)
        $\eta \leftarrow$ SMT.ExtractUnsatCore($\varphi$)
    else {Linear-search Mode}
        $\langle res, \mu \rangle \leftarrow$ SMT.IncrementalSolve($\varphi$)
        $\eta \leftarrow \emptyset$
    end if
    if (res = SAT) then
        $\langle \mathcal{M}, u \rangle \leftarrow$ Minimize(cost, $\mu$)
        $\varphi \leftarrow \varphi \cup \{(cost < u)\}$
    else {res = UNSAT }
        if (PIV $\notin \eta$) then
            $l \leftarrow u$
        else
            $l \leftarrow$ pivot
            $\varphi \leftarrow \varphi \setminus \{PIV\}$
            $\varphi \leftarrow \varphi \cup \{\neg PIV\}$
        end if
    end if
end while
return $\langle \mathcal{M}, u \rangle$
```

This approach is called offline because the SMT solvers used to find the models are black boxes, we simply call it. We don't need to change their internals

In binary search, if we don't find the solution in the left interval, it is necessarily on the right side, we update the pivots. On linear search, if we find an UNSAT, we've found a previous optimal solution by adding the further constraint on the minimum cost.

The minimal cost is computed by a minimizer over linear rational inequalities. The offline approach can be improved by an inline schema, which is more efficient, but requires modifying the internals of SMT solver. This basically integrates the optimization procedure into the SMT solver.

SMT Technology

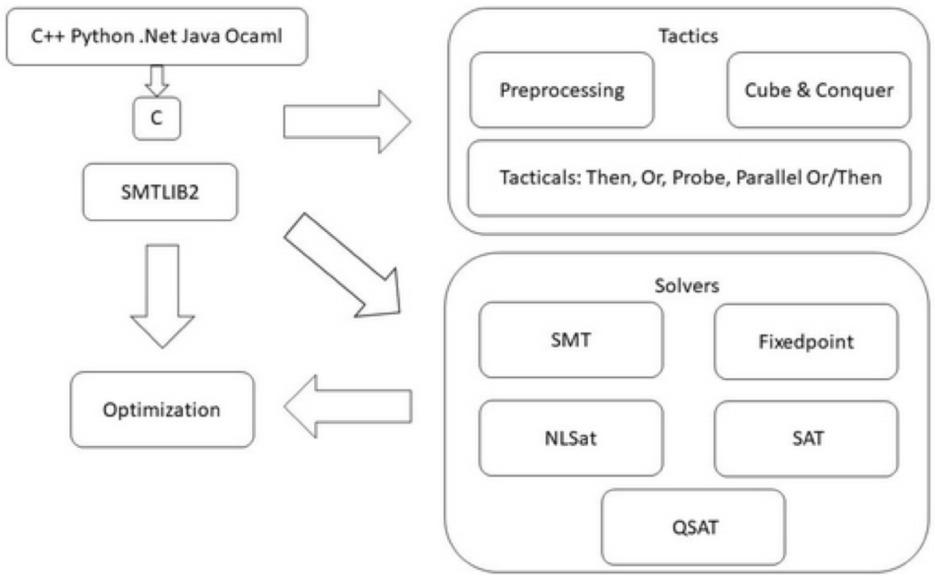
Given a theory $\mathcal{T}$, a $\mathcal{T}$-solver is a procedure for deciding whether a conjunction of $\mathcal{T}$-literals is satisfiable (SAT or UNSAT). We can define a SMT solver as a collection of $\mathcal{T}_i$-solvers for different theories $\mathcal{T}_i$ (or combinations of those). SMT solvers can handle formulas involving variables of different sort (type). The user interacts with SMT solvers through queries, to check the satisfiability of a formula or add new formulas.

Nowadays plenty of SMT solvers are available, especially for software analysis applications. Some of them are "special-purpose", others are more "general-purpose": they can handle different theories.

Z3

Well known SMT solver with specialized algorithms for efficiently tackling several theories (2008). It is open source and provides APIs for common programming languages (e.g. Python - Z3Py, Java, C++)

Z3 handles different sorts apart from built-in `Bool` , like for example `Int`, `Real`, `BitVec`, `Array`, `String` → Formulas are terms of Bool sort, that may include (un-)interpreted functions and constants.



The baseline theory is the EUF theory, which is computed with a standard congruence closure procedure; it also supports several well-known theories. Arithmetical constraints are clearly fundamentals. Z3 has different procedures according to which fragment of arithmetic is used:

Logic	Description	Solver	Example
LRA	Linear Real Arithmetic	Dual Simplex [28]	$x + \frac{1}{2}y \leq 3$
LIA	Linear Integer Arithmetic	Cuts + Branch	$a + 3b \leq 3$
LIRA	Mixed Real/Integer	[7, 12, 14, 26, 28]	$x + a \geq 4$
IDL	Integer Difference Logic	Floyd-Warshall	$a - b \leq 4$
RDL	Real Difference Logic	Bellman-Ford	$x - y \leq 4$
UTVPI	Unit two-variable / inequality	Bellman-Ford	$x + y \leq 4$
NRA	Polynomial Real Arithmetic	Model based CAD [42]	$x^2 + y^2 < 1$
NIA	Non-linear Integer Arithmetic	CAD + Branch [41]	$a^2 = 2$
		Linearization [15]	

CAD = Cylindrical Algebraic Decomposition

If we need to precisely model finite precision arithmetic, then using fixed width bit-vectors is probably a better choice. Z3 handles bit-vectors with eager SAT encoding (bit-blasting)

The other theories offered are Arrays, via reduction to EUF, Floating Points, via reduction to Bit-Vectors, Algebraic Datatypes, which captures the theory of finite trees, String and Sequences, which encode the theory of free monoids, with further specific operations (length, replace,…)

Z3 allows Incremental Solving: we can dynamically add and retract some formulas, which is not possible in any state-of-the-art CP Solvers. How this is performed? Via push and pop operations (similar to stack data structures); this creates local scopes: assertions added within a “push” are retracted on the matching “pop” (and this is something CP Solvers cannot do, you need to restart from scratch).

Z3 enables Optimization Modulo Theory via the `Optimize` module in 2 ways:

- by specifying (classic way) an objective function to minimize or maximize
- via soft constraints

The objective function is either a linear arithmetical term or a bit-vector term. Soft constraints are assertions that the solver can ignore if needed (e.g. timetabling problems), if considering them makes the problem unsatisfiable. The goal is to maximize the satisfied soft constraints (MaxSMT). Soft constraints might have an optional weight (a priority), which make the goal minimizing the sum of the weights of unsatisfied constraints.

CVC

Stands for *Cooperating Validity Checker* (2002, Stanford University). Currently we are at version 5, which is a major improvement from 4, and introduces new APIs, theories, solvers and procedures. It provides strong performance on industrial use cases. It is open source and can be programmed via APIs (C++, Java, Python) or executed in interactive mode. The Optimization is not supported, which is not surprising as optimization is not an active feature of SMT families of solvers (SAT stands for Satisfiability). Even though there's no native support, we can implement it through the use of a correct search algorithm (linear, binary,…)

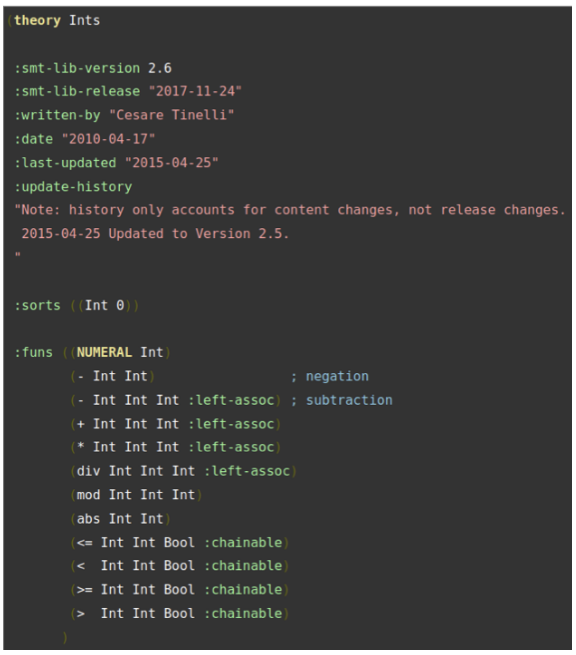
SMT-LIB

Each state-of-the-art solver has its strengths and weaknesses, so the selection is not trivial (Algorithm Selection problem - for a given problem, one solver might be better than the other). The idea, anyway, is to "*Model once, solve everywhere*", so there's a need for standardization, which is answered by SMT-LIB (2003). Also CP and SAT have their own standards (and CP's is not MiniZinc, which is just a De Facto).

SMT-LIB's objective is to:

- Provide rigorous descriptions of SMT theories
- Develop and promote common languages for SMT solvers
- Connect developers, researchers and users of the SMT community → standardization promotes the formation of a community
- Establish and make available benchmarks for SMT solvers
- Collect and promote software tools useful to the SMT community

It uses a parenthesized prefix notation, which is similar to functional languages (LISP). It is designed to be machine-readable rather than human-readable. It has three main components: theory declarations, logic declarations (basically theory declarations with restrictions) and scripts. SMT-LIB theories are defined by sorts (types) and functions: predicates $\equiv$ `Bool`-valued functions. SMT-LIB logics consist of Theory declarations + Restrictions on formulas. What is interesting for us is to define a script.

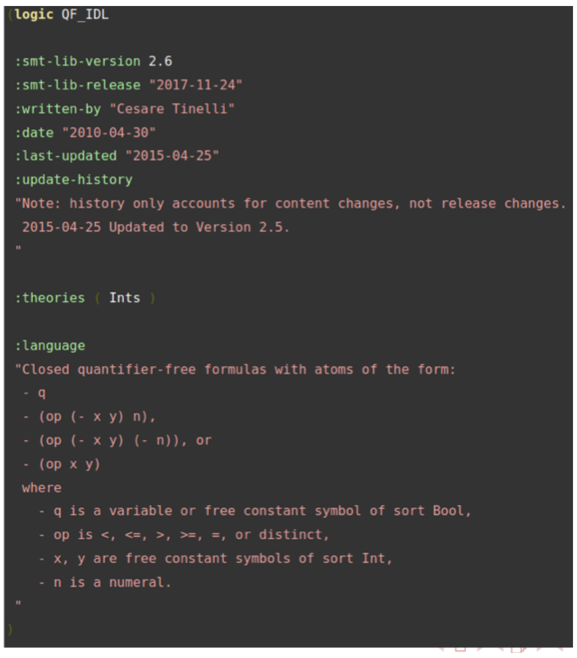


```
(theory Ints

 :smt-lib-version 2.6
 :smt-lib-release "2017-11-24"
 :written-by "Cesare Tinelli"
 :date "2010-04-17"
 :last-updated "2015-04-25"
 :update-history
 "Note: history only accounts for content changes, not release changes.
  2015-04-25 Updated to Version 2.5.
 "

 :sorts ((Int 0))

 :funs ((NUMERAL Int)
        (- Int Int)                 ; negation
        (- Int Int Int :left-assoc) ; subtraction
        (+ Int Int Int :left-assoc)
        (* Int Int Int :left-assoc)
        (div Int Int Int :left-assoc)
        (mod Int Int Int)
        (abs Int Int)
        (<= Int Int Bool :chainable)
        (<  Int Int Bool :chainable)
        (>= Int Int Bool :chainable)
        (>  Int Int Bool :chainable)
       )
```



```
(logic QF_IDL

 :smt-lib-version 2.6
 :smt-lib-release "2017-11-24"
 :written-by "Cesare Tinelli"
 :date "2010-04-30"
 :last-updated "2015-04-25"
 :update-history
 "Note: history only accounts for content changes, not release changes.
  2015-04-25 Updated to Version 2.5.
 "

 :theories ( Ints )

 :language
 "Closed quantifier-free formulas with atoms of the form:
  - q
  - (op (- x y) n),
  - (op (- x y) (- n)), or
  - (op x y)
  where
    - q is a variable or free constant symbol of sort Bool,
    - op is <, <=, >, >=, =, or distinct,
    - x, y are free constant symbols of sort Int,
    - n is a numeral.
 "
)
```

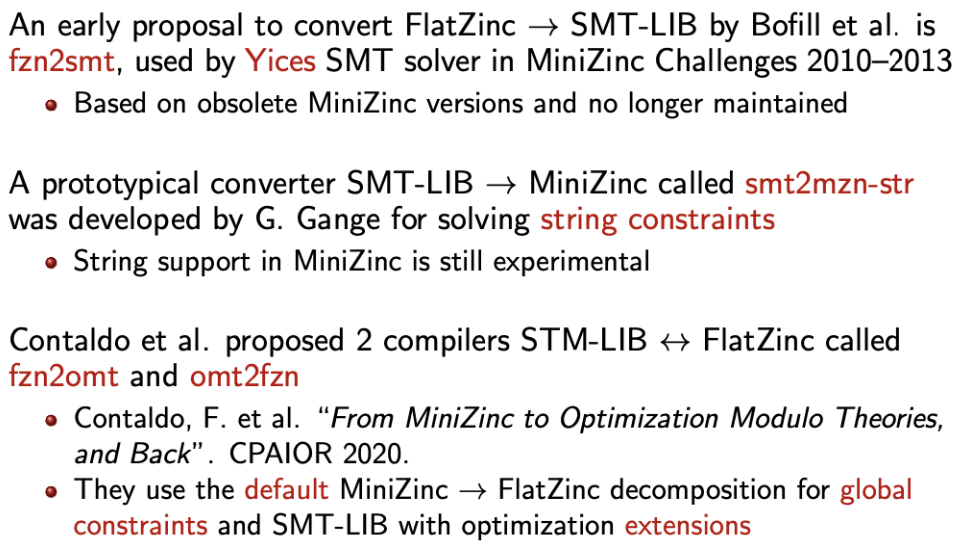
- `set-logic` specifies the logic
- `declare-fun` introduces a new function symbol, so it can be used to declare variables too (variables $\equiv$ uninterpreted constants). We have also `declare-const` for constants
- `assert` specifies formulas and check-sat checks the satisfiability of all the specified formulas

SMT solvers react to commands by modifying an assertion stack. Each stack element is called level and consists of a set of assertions (formulas + declarations/definitions of sorts and functions). By default, a new assertion always belongs to the current level. Levels can be added and removed with push and pop commands, noting that pop removes all level assertions, including declarations/definitions.

Standard SMT-LIB does not have an explicit support to optimization. One possible workaround would be to implement an offline OMT approach (SMT solvers as black-boxes - either a binary or a linear search to repeatedly check satisfiability). SMT solvers should be in incremental mode avoiding to restart each time a new bound is found. In binary search mode, one should use push/pop primitives.

SMT-LIB $\Longleftrightarrow$ MiniZinc

One may think of SMT-LIB as the equivalent (SMT) of MiniZinc language for CP problems. SMT-LIB is lower-level, more similar to FlatZinc (MiniZinc models, which are higher-level, together with optional data and solver-specific redefinitions, are compiled into FlatZinc instances). Translation is quite straightforward, except that MiniZinc does not support all the standard SMT-LIB theories (e.g. does not support Theories of Arrays and Strings, and Global Constraints are likely lost). One can translate SMT-LIB to FlatZinc, if the target solver is known or simply ignored.



An early proposal to convert FlatZinc $\rightarrow$ SMT-LIB by Bofill et al. is fzn2smt, used by Yices SMT solver in MiniZinc Challenges 2010–2013

- Based on obsolete MiniZinc versions and no longer maintained

A prototypical converter SMT-LIB $\rightarrow$ MiniZinc called smt2mzn-str was developed by G. Gange for solving string constraints

- String support in MiniZinc is still experimental

Contaldo et al. proposed 2 compilers STM-LIB $\leftrightarrow$ FlatZinc called fzn2omt and omt2fzn

- Contaldo, F. et al. "*From MiniZinc to Optimization Modulo Theories, and Back*". CPAIOR 2020.
- They use the default MiniZinc $\rightarrow$ FlatZinc decomposition for global constraints and SMT-LIB with optimization extensions

The standard has decided to prefer the ease of parsing over human readability. SMT-LIB is created to be readable by solvers. One could write the instance manually, or define an ad hoc script (bash or python) → very specific, but for the project might work. One could also define the instance with Z3 and then use Z3's API to generate the corresponding instance (though this translation might introduce additional variables).