mSAT: a Sat/SMT/McSat library

Guillaume Bury

October 18, 2016

1 Introduction

The goal of the mSAT library is to provide a way to easily create automated theorem provers based on a Sat solver. More precisely, the library, written in OCaml, provides functors which, once instantiated, provide a Sat, SMT or McSat solver.

Given the current state of the art of SMT solvers, where most Sat solvers are written in C and heavily optimised[1], the mSAT library does not aim to provide solvers competitive with the existing implementations, but rather an easy way to create reasonably efficient solvers.

mSAT currently uses the following techniques:

- 2-watch literals scheme
- Activity based decisions
- Restarts

Additionally, mSAT has the following features:

- Local assumptions
- Proof/Model output
- Adding clauses during proof search

[1]Some solvers now have instructions to manage a processor's cache

Contents

2 Sat Solvers: principles and formalization

2.1 Idea

2.2 Inference rules

The SAT algorithm can be formalized as follows. During the search, the solver keeps a set of clauses, containing the problem hypotheses and the learnt clauses, and a trail, which is the current ordered list of assumptions and/or decisions made by the solver.

Each element in the trail (decision or propagation) has a level, which is the number of decision appearing in the trail up to (and including) it. So for instance, propagations made before any decisions have level 0, and the first decision has level 1. Propagations are written $a \leadsto_C \top$, with C the clause that caused the propagation, and decisions $a \mapsto_n \top$, with n the level of the decision. Trails are read chronologically from left to right.

In the following, given a trail t and an atomic formula a, we will use the following notation: $a \in t$ if $a \mapsto_n \top$ or $a \leadsto_C \top$ is in t, i.e. $a \in t$ is a is true in the trail t. In this context, the negation $\neg$ is supposed to be involutive (i.e. $\neg\neg a = a$), so that, if $a \in t$ then $\neg\neg a = a \in t$.

There exists two main Sat algorithms: DPLL and CDCL. In both, there exists two states: first, the starting state Solve, where propagations and decisions are made, until a conflict is detected, at which point the algorithm enters in the Analyse state, where it analyzes the conflict, backtracks, and re-enter the Solve state. The difference between DPLL and CDCL is the treatment of conflicts during the Analyze phase: DPLL will use the conflict only to known where to backtrack/backjump, while in CDCL the result of the conflict analysis will be added to the set of hypotheses, so that the same conflict does not occur again. The Solve state take as argument the set of hypotheses and the trail, while the Analyze state also take as argument the current conflict clause.

We can now formalize the CDCL algorithm using the inference rules in Figure 1. In order to completely recover the Sat algorithm, one must apply the rules with the following precedence and termination conditions, depending on the current state:

- If the empty clause is in $\mathbb{S}$, then the problem is unsat. If there is no more rule to apply, the problem is sat.
- If we are in Solve mode:
  1. First is the rule Conflict;
  2. Then the try and use Propagate;
  3. Finally, is there is nothing left to be propagated, the Decide rule is used.
- If we are in Analyze mode, we have a choice concerning the order of application. First we can observe that the rules Analyze-Propagate, Analyze-Decision and Analyze-Resolution can not apply simultaneously, and we

will thus group them in a super-rule Analyze. We now have the choice of when to apply the Backjump rule compared to the Analyze rule: using Backjump eagerly will result in the first UIP strategy, while delaying it until later will yield other strategies, both of which are valid.

2.3 Invariants, correctness and completeness

The following invariants are maintained by the inference rules in Figure 1:

Trail Soundness In Solve$(\mathbb{S}, t)$, if $a \in t$ then $\neg a \notin t$

Conflict Analysis In Analyze$(\mathbb{S}, t, C)$, C is a clause implied by the clauses in $\mathbb{S}$, and $\forall a \in C . \neg a \in t$ (i.e. C is entailed by the hypotheses, yet false in the partial model formed by the trail t).

Equivalence In any rule $\frac{s_1}{s_2}$, the set of hypotheses (usually written $\mathbb{S}$) in s_1 is equivalent to that of s_2.

These invariants are relatively easy to prove, and provide an easy proof of correctness for the CDCL algorithm. Termination can be proved by observing that the same trail appears at most twice during proof search (once during propagation, and eventually a second time right after backjumping[2]). Correctness and termination implies completeness of the Sat algorithm.

3 SMT solver architecture

3.1 Idea

In a SMT solver, after each propagation and decision, the solver sends the newly assigned literals to the theory. The theory then has the possibility to declare the current set of literals incoherent, and give the solver a tautology in which all literals are currently assigned to $\bot$[3], thus prompting the solver to backtrack. We can represent a simplified version of the information flow (not taking into account backtracking) of usual SMT Solvers, using the graph in fig 2.

Contrary to what the Figure 2 could suggest, it is not impossible for the theory to propagate information back to the Sat solver. Indeed, some SMT solvers already allow the theory to propagate entailed literals back to the Sat solver. However, these propagations are in practice limited by the complexity of deciding entailment. Moreover, procedures in a SMT solver should be incremental in order to get decent performances, and deciding entailment in an incremental manner is not easy (TODO : ref nÃľcessaire). Doing efficient, incremental entailment is exactly what McSat allows (see Section 4).

[2]This could be avoided by making the Backjump rule directly propagate the relevant literal of the conflict clause, but it needlessly complicates the rule.

[3]or rather for each literal, its negation is assigned to $\top$

Sat

$$\textsc{Propagate}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Sove}(\mathbb{S}, t :: a \rightsquigarrow_C \top)} \qquad \begin{array}{l} a \in C, C \in \mathbb{S}, \neg a \notin t \\ \forall b \in C. b \neq a \to \neg b \in t \end{array}$$

$$\textsc{Decide}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S}, t :: a \mapsto_n \top)} \qquad \begin{array}{l} a \notin t, \neg a \notin t, a \in \mathbb{S} \\ n = \text{max\_level}(t) + 1 \end{array}$$

$$\textsc{Conflict}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Analyze}(\mathbb{S}, t, C)} \qquad C \in \mathbb{S}, \forall a \in C. \neg a \in t$$

$$\textsc{Analyze-propagation}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \rightsquigarrow_C \top, D)}{\text{Analyze}(\mathbb{S}, t, D)} \qquad \neg a \notin D$$

$$\textsc{Analyze-decision}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \mapsto_n \top, D)}{\text{Analyze}(\mathbb{S}, t, D)} \qquad \neg a \notin D$$

$$\textsc{Analyze-Resolution}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \rightsquigarrow_C \top, D)}{\text{Analyze}(\mathbb{S}, t, (C - \{a\}) \cup (D - \{\neg a\}))} \qquad \neg a \in D$$

$$\textsc{Backjump}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \mapsto_d \top :: t', C)}{\text{Solve}(\mathbb{S} \cup \{C\}, t)} \qquad \begin{array}{l} \text{is\_uip}(C) \\ d \leq \text{uip\_level}(C) \end{array}$$

SMT

$$\textsc{Conflict-Theory}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Analyze}(\mathbb{S}, t, C)} \qquad \begin{array}{l} \mathcal{T} \vdash C \\ \forall a \in C. \neg a \in t \end{array}$$

McSat

$$\frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S}, t :: a \mapsto_n v)} \qquad a \notin t, a \in \mathbb{S}, n = \text{max\_level}(t) + 1$$

$$\frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S}, t :: a \rightsquigarrow_n \top)}$$

Figure 1: Inference rules

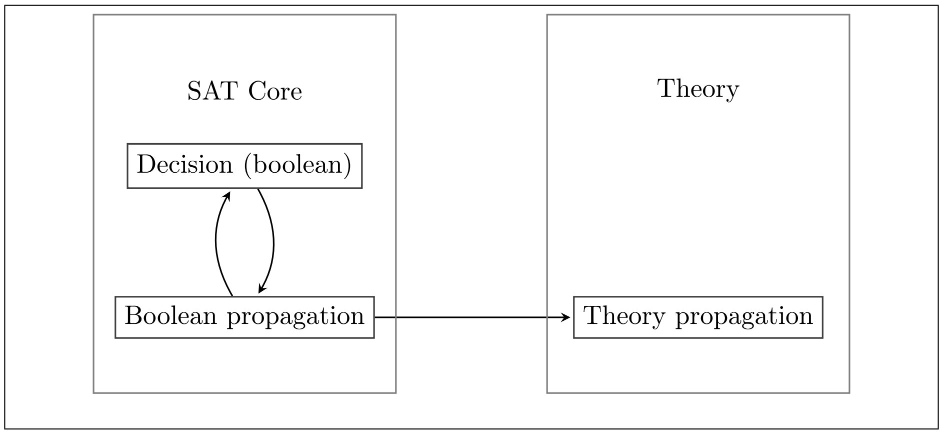

Figure 2: Simplified SMT Solver architecture

3.2 Formalization and Theory requirements

An SMT solver is the combination of a Sat solver, and a theory $\mathcal{T}$. The role of the theory $\mathcal{T}$ is to stop the proof search as soon as the trail of the Sat solver is inconsistent. A trail is inconsistent iff there exists a clause C, which is a tautology of $\mathcal{T}$ (thus $\mathcal{T} \vdash C$), but is not satisfied in the current trail (each of its literals has either been decided or propagated to false). Thus, we can add the Conflict-Theory rule (see Figure 1) to the CDCL inference rules in order to get a SMT solver. We give the Conflict-Theory rule a slightly lower precedence than the Conflict rule for performance reason (detecting boolean conflict is faster than theory specific conflicts).

So, what is the minimum that a theory must implement in practice to be used in a SMT solver ? The theory has to ensure that the current trail is consistent (when seen as a conjunction of literals), that is to say, given a trail t, it must ensure that there exists a model $\mathcal{M}$ of $\mathcal{T}$so that $\forall a \in t. \mathcal{M} \vDash a$, or if it is impossible (i.e. the trail is inconsistent) produce a conflict.

4 McSat: An extension of SMT Solvers

4.1 Motivation and idea

McSat is an extension of usual SMT solvers, introduced in [2] and [1]. In usual SMT Solvers, interaction between the core SAT Solver and the Theory is pretty limited : the SAT Solver make boolean decisions and propagations, and sends them to the theory, whose role is in return to stop the SAT Solver as soon as the current set of assumptions is incoherent. This means that the information that theories can give the SAT Solver is pretty limited, and furthermore it heavily restricts the ability of theories to guide the proof search (see Section 3.1).

While it appears to leave a reasonably simple job to the theory, since it completely hides the propositional structure of the problem, this simple interaction

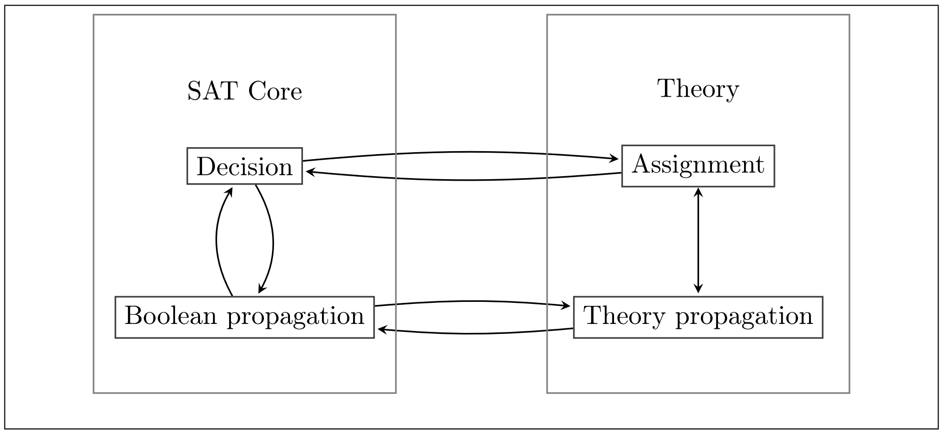

Figure 3: Simplified McSat Solver architecture

between the SAT Solver and the theory makes it harder to combine multiple theories into one. Usual techniques for combining theories in SMT solvers typically require to keep track of equivalence classes (with respect to the congruence closure) and for instance in the Nelson-Oppen method for combining theories (TODO : ref nÃľcessaire) require of theories to propagate any equality implied by the current assertions.

McSat extends the SAT paradigm by allowing more exchange of information between the theory and the SAT Solver. This is achieved by allowing the solver to not only decide on the truth value of atomic propositions, but also to decide assignments for terms that appear in the problem. For instance, if the SAT Solver assigns a variable x to 0, an arithmetic theory could propagate to the SAT Solver that the formula $x < 1$ must also hold, instead of waiting for the SAT Solver to guess the truth value of $x < 1$ and then inform the SAT Solver that the conjunction : $x = 0 \wedge \neg x < 1$ is incoherent.

This exchange of information between the SAT Solver and the theories results in the construction of a model throughout the proof search (which explains the name Model Constructing SAT).

The main addition of McSat is that when the solver makes a decision, instead of being restricted to making boolean assignment of formulas, the solver now can decide to assign a value to a term belonging to one of the literals. In order to do so, the solver first chooses a term that has not yet been assigned, and then asks the theory for a possible assignment. Like in usual SMT Solvers, a McSat solver only exchange information with one theory, but, as we will see, combination of theories into one becomes easier in this framework, because assignments are actually a very good way to exchange information.

Using the assignments on terms, the theory can then very easily do efficient propagation of formulas implied by the current assignments: it is enough to evaluate formulas using the current partial assignment. The information flow then looks like fig 3.

4.2 Decisions and propagations

In McSat, semantic propagations are a bit different from the propagations used in traditional SMT Solvers. In the case of McSat (or at least the version presented here), semantic propagations strictly correspond to the evaluation of formulas in the current assignment. Moreover, in order to be able to correctly handle these semantic propagations during backtrack, they are assigned a level: each decision is given a level (using the same process as in a Sat solvers: a decision level is the number of decision previous to it, plus one), and a formula is propagated at the maximum level of the decisions used to evaluate it.

We can thus extend the notations introduced in Section 2.2: $t \mapsto_n v$ is a semantic decision which assign t to a concrete value v, $a \leadsto_n \top$ is a semantic propagation at level n.

For instance, if the current trail is $\{x \mapsto_1 0, x + y + z = 0 \mapsto_2 \top, y \mapsto_3 0\}$, then $x + y = 0$ can be propagated at level 3, but $z = 0$ can not be propagated, because z is not assigned yet, even if there is only one remaining valid value for z. The problem with assignments as propagations is that it is not clear what to do with them during the Analyze phase of the solver, see later.

4.3 First order terms & Models

A model traditionally is a triplet which comprises a domain, a signature and an interpretation function. Since most problems define their signature using type definitions at the beginning, and built-in theories such as arithmetic usually have canonic domain and signature, we will consider in the following that the domain $\mathbb{D}$ and signature are given (and constant). Additionally, we consider a fixed interpretation of the theory-defined symbols (which usually does along with the domain).

In the following, we use the following notations:

- $\mathbb{V}$ is an infinite set of variables;
- $\mathbb{C}$ is the possibly infinite set of constants defined by the input problem's theories[4];
- $\mathbb{S}$ is the finite set of constants defined by the input problem's type definitions;
- $\mathbb{T}$ is the (infinite) set of first-order terms over $\mathbb{V}$, $\mathbb{C}$ and $\mathbb{S}$ (for instance $a, f(0), x + y, \ldots$);
- $\mathbb{F}$ is the (infinite) set of first order quantified formulas over the terms in $\mathbb{T}$.

We are therefore interested in finding an interpretation of a problem, and more specifically an interpretation of the symbols in $\mathbb{S}$ not defined by the theory,

[4]For instance, the theory of arithmetic defines the usual operators $+, -, *, /$ as well as $0, -5, \frac{1}{2}, -2.3, \ldots$

i.e. non-interpreted functions. An interpretation $\mathcal{I}$ can easily be extended to a function from ground terms and formulas to model value by recursively applying it:

$$\mathcal{I}(f(e_1, \ldots, e_n)) = \mathcal{I}_f(\mathcal{I}(e_1), \ldots, \mathcal{I}(e_n))$$

Partial Interpretation The goal of McSat is to construct a first-order model of the input formulas. To do so, it has to build what we will call partial interpretations: intuitively, a partial interpretation is a partial function from the constants symbols to the model values. It is, however, not so simple: during proof search, the McSat algorithm maintains a partial mapping from expressions to model values (and not from constant symbols to model values). The intention of this mapping is to represent a set of constraints on the partial interpretation that the algorithm is building. For instance, given a function symbol f of type (int $\rightarrow$ int) and an integer constant a, one such constraint that we would like to be able to express in our mapping is the following: $f(a) \mapsto 0$, regardless of the values that f takes on other argument, and also regardless of the value mapped to a. To that end we introduce the notion of abstract partial interpretation.

An abstract partial interpretation σ is a mapping from ground expressions to model values. To each abstract partial interpretation correspond a set of complete models that realize it. More precisely, any mapping σ can be completed in various ways, leading to a set of potential interpretations:

$$\text{Complete}(\sigma) = \{\mathcal{I} \mid \forall t \mapsto v \in \sigma, \mathcal{I}(t) = v\}$$

Coherence An abstract partial interpretation σ is said to be coherent iff there exists at least one model that completes it, i.e. Complete$(\sigma) \neq \emptyset$. One example of incoherent abstract partial interpretation is the following mapping:

$$\sigma = \begin{cases} a \mapsto 0 \\ b \mapsto 0 \\ f(a) \mapsto 0 \\ f(b) \mapsto 1 \end{cases}$$

Compatibility In order to do semantic propagations, we want to have some kind of notion of evaluation for abstract partial interpretations, and we thus define the partial interpretation function in the following way:

$$\forall t \in \mathbb{T} \cup \mathbb{F}. \forall v \in \mathbb{D}. \, (\forall \mathcal{I} \in \text{Complete}(\sigma). \mathcal{I}(t) = v) \rightarrow \sigma(t) = v$$

The partial interpretation function is the intersection of the interpretation functions of all the completions of σ, i.e. it is the interpretation where all completions agree. We can now say that a mapping σ is compatible with a trail t iff σ is coherent, and $\forall a \in t. \not\exists(\sigma(a) = \bot)$, or in other words, for every literal a true in the trail, there exists at least one model that completes σ and where a is satisfied.

Completeness We need one last property on abstract partial interpretations, which is to specify the relation between the terms in a mapping, and the terms it can evaluate, according to its partial interpretation function defined above. Indeed, at the end of proof search, we want all terms (and sub-terms) of the initial problem to be evaluated using the final mapping returned by the algorithm when it finds that the problem is satisfiable: that is what we will call completeness of a mapping. To that end we will here enumerate some sufficient conditions on the mapping domain $\mathrm{Dom}(\sigma)$ compared to the finite set of terms (and sub-terms) T that appear in the problem.

A first way, is to have $\mathrm{Dom}(\sigma) = T$, i.e. all terms (and sub-terms) of the initial problem are present in the mapping. While this is the simplest way to ensure that the mapping is complete, it might be a bit heavy: for instance, we might have to assign both x and $2x$, which is redundant. The problem in that case is that we try and assign a term for which we could actually compute the value from its arguments: indeed, since the multiplication is interpreted, we do not need to interpret it in our mapping. This leads to another way to have a complete mapping: if $\mathrm{Dom}(\sigma)$ is the set of terms of T whose head symbol is uninterpreted (i.e. not defined by the theory), it is enough to ensure that the mapping is complete, because the theory will automatically constrain the value of terms whose head symbol is interpreted.

References

[1] Devan Jovanovic, Clark Barrett, and Leonardo de Moura. The design and implementation of the model constructing satisfiability calculus. 2013.

[2] Devan Jovanovic and Leonardo de Moura. A model-constructing satisfiability calculus. 2013.