



# mSAT:An OCaml SAT Solver

Guillaume Bury

► **To cite this version:**

Guillaume Bury. mSAT:An OCaml SAT Solver. OCaml Users and Developers Workshop, Sep 2017, Oxford, United Kingdom. <hal-01670765>

**HAL Id: hal-01670765**

**<https://hal.inria.fr/hal-01670765>**

Submitted on 21 Dec 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

## Introduction

mSAT : a SAT solving library in OCaml. It solves the **satisfiability** of propositional clauses. It is **Modular** : the user provides the theory. And it **produces formal proofs**.

## Conflict Driven Clause Learning

**Propagation** If there exists a clause  $C = C' \vee a$ , where  $C'$  is false in the partial model, then add  $a \mapsto \top$  to the partial model, and record  $C$  as the reason for  $a$ .

**Decision** Take an atom  $a$  that is not yet in the partial model, and add  $a \mapsto \top$  to the model.

**Conflict** A conflict is a clause  $C$  that is false in the current partial model.

**Analyze** Perform resolution between the analyzed clause and the reason behind the propagation of its most recently assigned literal, until the analyzed clause is suitable for backjumping.

**Backjump** A clause is suitable for backjumping if its most recently assigned literal  $a$  is a decision. We can then backtrack to before the decision, and add the analyzed clause to the solver, which will then enable to propagate  $a \mapsto \perp$ .

**SMT** Formulas using first-order theories can be handled using a theory. Each formula propagated or decided is sent to the theory, which then has the duty to check whether the conjunction of all formulas seen so far is satisfiable, if not, it should return a theory tautology (as a clause), that is not satisfied in the current partial model.

## Implementation

### ► Imperative design

- ✓ 2-watch literal
- ✓ Backtrackable theories (less demanding than immutable theories)

### ► Features

- ✓ Functorized design, using generative functors
- ✓ Local assumptions
- ✓ Model output and proof output (Coq, dot)

## Solver Interface

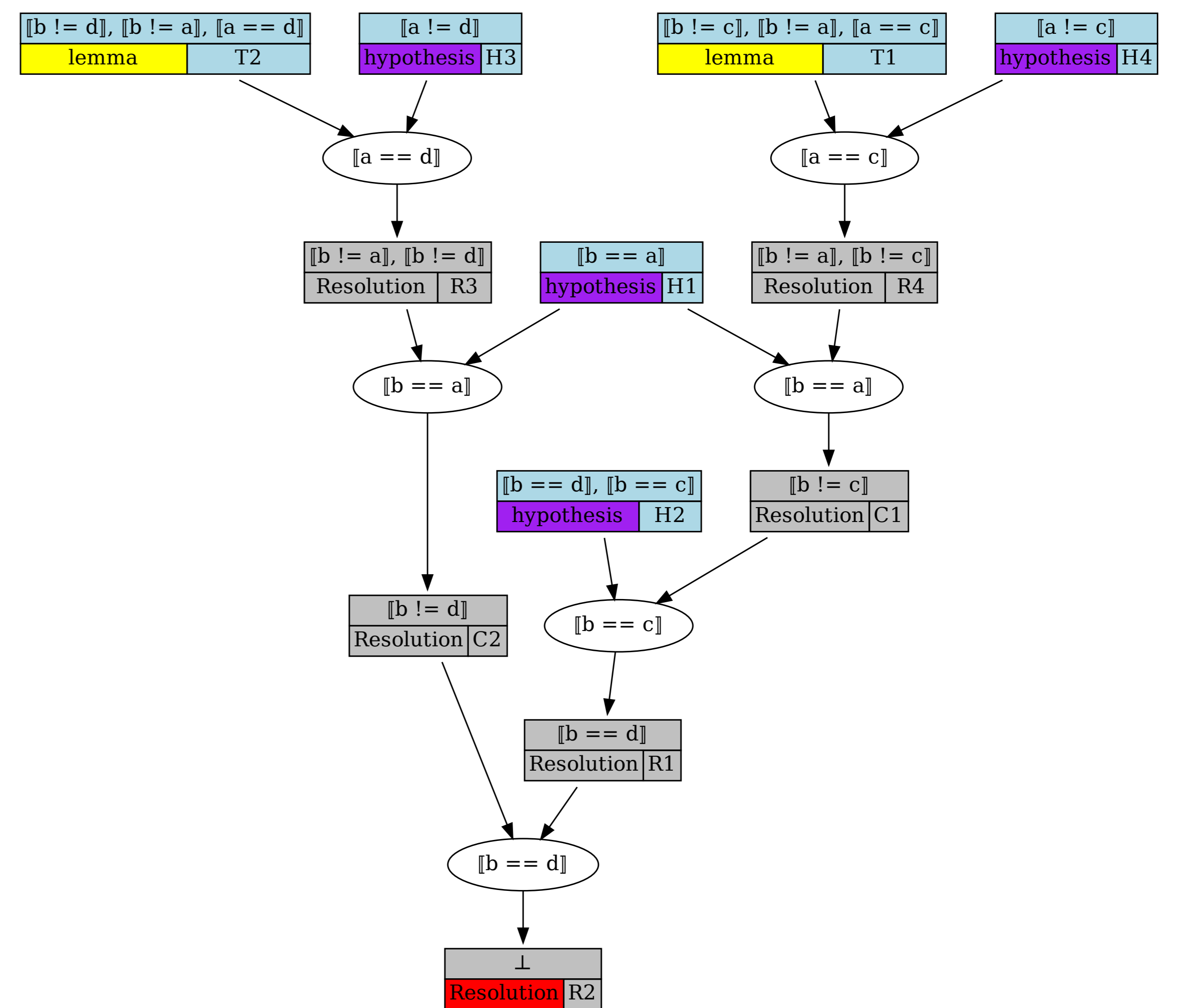
```
module Make(Th: Theory_intf.S)() : sig
  type 'f sat_state = { eval : 'f -> bool; ... }
  type ('c, 'p) unsat_state =
    { conflict: unit -> 'c; proof : unit -> 'p }
  type res = Sat of formula sat_state
    | Unsat of (clause, proof) unsat_state
  val assume : ?tag:int -> atom list list -> unit
  val solve : ?assumptions:atom list -> unit -> res
end
```

## Other Solvers

| name                 | theory | bindings    | features      |
|----------------------|--------|-------------|---------------|
| regstab              | SAT    | binary only | only pure SAT |
| <b>minisat</b>       |        |             |               |
| <b>sattools</b>      | SAT    | C bindings  | only pure SAT |
| ocaml-sat-solvers    |        |             |               |
| Alt-ergo             | SMT    | binary only | Fixed theory  |
| <b>Alt-ergo-zero</b> | SMT    | OCaml lib   | Fixed theory  |
| ocamlyices           |        |             |               |
| yices2               | SMT    | C bindings  | Fixed theory  |

## Problem Example

Are the following hypotheses satisfiable?  
 $H1 : a = b$      $H2 : b = c \vee b = d$   
 $H3 : a \neq d$      $H4 : a \neq c$



## Theory Interface

```
type ('f, 'p) res = Sat | Unsat of 'f list * 'p
type 'f slice = { start:int; length:int; get:int -> 'f }
module type S = sig
  val backtrack : level -> unit
  val current_level : unit -> level
  val assume : formula slice -> (formula, proof) res
end
```

## Proof Generation

- ✓ Each clause records its "history" which is the clauses used during analyzing
- ✓ Minimal impact on proof search (already done to compute unsat-core)
- ✓ Sufficient to rebuild the whole resolution tree
- ✓ A proof is a clause and proof nodes are expanded on demand → no memory issue
- ✓ Enables various proof outputs :
  - Dot/Graphviz (see example above)
  - Coq (and soon Dedukti) formal proofs

## Performances

| solver (package)  | Alt-ergo-zero (aez) | mSAT (msat) | minisat (minisat/sattools)            | cryptominisat (sattools) |
|-------------------|---------------------|-------------|---------------------------------------|--------------------------|
| uuf100 (1000 pbs) | 0.125               | 0.012       | 0.004                                 | 0.006                    |
| uuf125 (100 pbs)  | 2.217               | 0.030       | 0.006                                 | 0.013                    |
| uuf150 (100 pbs)  | 67.563              | 0.087       | 0.017                                 | 0.045                    |
| pigeon/hole6      | 0.120               | 0.018       | 0.006                                 | 0.006                    |
| pigeon/hole7      | 4.257               | 0.213       | 0.015                                 | 0.073                    |
| pigeon/hole8      | 31.450              | 0.941       | 0.096                                 | 2.488                    |
| pigeon/hole9      | timeout (600)       | 8.886       | 0.634                                 | 4.075                    |
| pigeon/hole10     | timeout (600)       | 161.478     | 9.579 (minisat)<br>160.376 (sattools) | 72.050                   |