

Certificates and satisfiability

Laurent Théry, Benjamin Grégoire, Michaël Armand

INRIA Sophia

September 10, 2009

Initial project

We want to link SMT with Coq

SMT (**DPLL(T)**) are made of :

- SAT Solver (**DPLL**)
- consistence checker (for theory **T**)

First step : linking SAT with Coq

First step

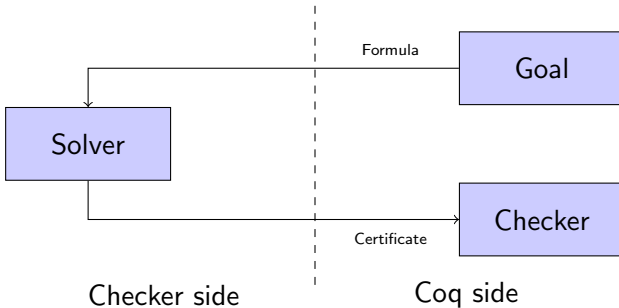


Table of contents

- 1 DPLL Algorithm
 - Standard Algorithm
 - Usual optimizations
- 2 Checking certificate
 - Certificates and Coq
 - Resolution verification
 - Replaying DPPL
 - Efficient programming in Coq
- 3 Proving Coq checker
- 4 Conclusion and future work : From SAT to SMT

DPLL Algorithm

DPLL is the algorithm used by most SAT solver (including Minisat)

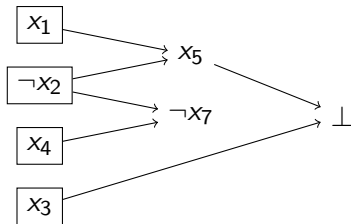
- choose a literal , which leads to partial assignment
: $\{x_1, \neg x_2, x_3, x_4, \neg x_7\}$
- **propagate unit information** : $\neg x_1 \vee x_2 \vee x_5$
- if a conflict is raised, **learn** a clause and **backjump** :
 $\neg x_3 \vee \neg x_5$
- else, choose another literal

We need to check each clause at each step!

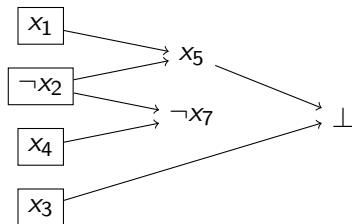
Conflict analysis : choosing and proving the learned clause

Learning clauses

We learn a clause C representing the conflict, called conflict clause. Both learning and backjump use C



Conflict analysis : choosing and proving the learned clause



$$\frac{\neg x_5 \vee \neg x_3 \quad \neg x_1 \vee x_2 \vee x_5}{\neg x_1 \vee x_2 \vee \neg x_3}$$

Given a clause $\neg x \vee C$ and a reason $x \vee C'$, we must prove $C \vee C'$:

$$\frac{\neg x \vee C \quad x \vee C'}{C \vee C'}$$

Proof of unsatisfiability

- We stop if \perp is learned (or if an assignment is found)
- We can prove :
 - $\vdash F \rightarrow F_0$ (resolution chain : $\vdash F \rightarrow C_0$)
 - ...
 - $\vdash F_n \rightarrow \perp$ (resolution chain : $\vdash F_n \rightarrow \perp$)
- So, we can prove the initial goal :

$$\frac{F \vdash \perp}{\vdash \neg F}$$

This is usual certificate from SAT solvers (including Minisat)

Optimizing DPLL

DPLL Algorithm needs :

- **unit propagation**
- conflict analysis (learning and backtrack)

An efficient unit propagation should :

- detect each possible propagation
- check only a few clauses

Optimizing unit propagation

We need a correct criterion to decide if a clause need to be checked when a literal l is chosen

Optimizing unit propagation

Watcher system

- for each unresolved clause C , we choose two literals of C : l and l' , both unresolved. l and l' are watchers for C
- as long as l and l' keep unresolved, C keep unresolved
- we check C only if $\neg l$ or $\neg l'$ is assigned

Implementation of watcher system

- first version :

$$C = l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5$$

$$\begin{array}{cccccc} & \uparrow & & \uparrow & & \\ & & & & & \end{array}$$

- second version :

$$C = l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5$$

$$\begin{array}{cccccc} & \uparrow & & & & \uparrow \\ & & & & & \end{array}$$

"assign $\neg l_1$ "

Implementation of watcher system

- first version :

$$C = l_3 \vee l_2 \vee l_1 \vee l_4 \vee l_5$$

$$\quad \uparrow \quad \uparrow$$

- second version :

$$C = l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5$$

$$\quad \quad \uparrow \quad \quad \uparrow$$

"backtrack $\neg l_1$ "

Implementation of watcher system

- first version :

$$C = l_3 \vee l_2 \vee l_1 \vee l_4 \vee l_5$$

$$\quad \uparrow \quad \quad \uparrow$$

- second version :

$$C = l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5$$

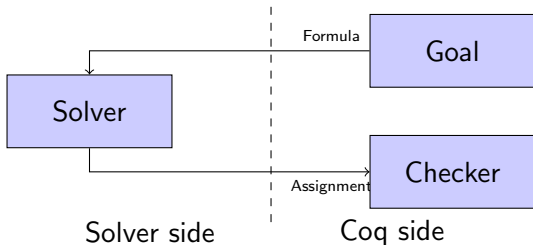
$$\quad \uparrow \quad \quad \quad \quad \quad \uparrow$$

External solver and internal checker

Useful observations...

- To check a solution is usually easier than to solve a problem
- C++ is faster than Coq

Example, certificate to prove satisfiability of a CNF Formula :



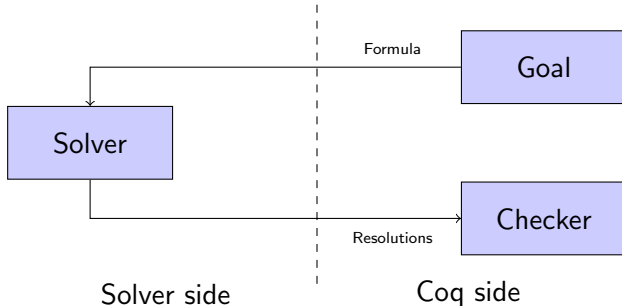
Generalities about certificates

- Coq must typecheck the certificate
- Coq must check the proof

N.B.

A good certificate is a compromise between size of certificate and information given to the checker

Resolution chains as certificates



- large information given to the checker
- large certificate
- large proof objects

Efficient resolution checker

- resolving c and c' consists in merging two clauses while looking for resolution var
- this operation is costly with list because we need to allocate a new list for each resolution
- with imperative structures (array), we can work on place and record the result in c

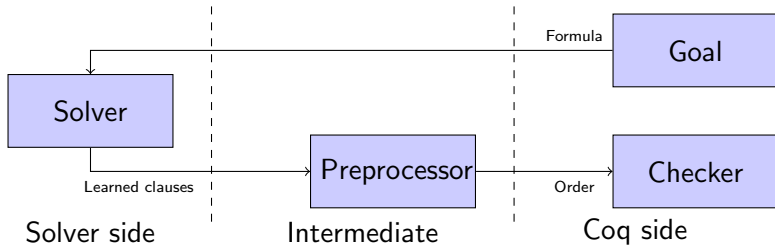
What does matter?

Other idea

Replay solver's algorithm, with more information

A good solver needs :

- efficient implementation for clause analysis
- a good heuristic to choose variable assignment order (*certificate*)
- a good heuristic to define conflict clauses (1UIP or using only decision variable)
- a good heuristic to select learned clauses (*certificate*)



Using learned clause to build a good decision heuristic

- Preprocessor runs DPLL Algorithm
- Preprocessor looks for the first unit clause under assignment in learned clauses from ext. solver
- Preprocessor records variable assignment order and set a flag for each learned clause that is used
- Checker runs DPLL Algorithm again with (possibly better) variable assignment order, and learning only useful clauses

This heuristic :

- is complete (we do not need to choose more variable assignments than the ones produced by learned clauses)
- is globally efficient since each chosen variable has been recorded as “implicit reason” of conflict

DPLL implementation and certification

Both Preprocessor and Checker need a DPLL implementation, with :

- imperative objects (array), as much as possible
- optimized unit propagation
- certification (Coq Checker)

Both approach need to handle efficiently large clauses. Using imperative objects such as arrays is needed. Note that :

- Coq is purely functional, we use *pseudo-imperative* programming
- this *pseudo-imperative* programming also makes the proof more complicated

Pseudo-imperative programming in Coq

```
...  
get : forall T : Type, array T -> int31 -> T ->  
array T  
set : forall T : Type, array T -> int31 -> T  
...
```

Functional interface, but imperative implementation

Experimental results

Example	Minisat	P.P.	1st Unit	Incr. Order	Resolutions
ex_aim50	1 ms	1 ms	1 + 4 ms	7 ms	18 ms
dubois50	8 ms	12 ms	11 + 80 ms	92 ms	23.07 s
barrel4	5 ms	13 ms	6 + 9 ms	17 ms	13.39 s
barrel5	392 ms	1.85 s	4.41 + 8.22 s	18.11 s	1287 s
barrel6	2.49 s	20.66 s	10.51 + 65.57 s	114.15 s	-
barrel7	8.10 s	106.45 s	97.42 + 57.23 s	511.68 s	-
longmult	15.66 s	453.25 s	117 + 1296 s	2583 s	-
hole8	210 ms	29.76 s	23.63 + 116.91 s	985 ms	522 s

We can handle cases with 3 500 variables and 14 000 clauses (like barrel7). Resolution version values should be time for typechecking.

Experimental result for resolution checker

zChaff	0.065
Coq res. checking	0.122
.v file size for resolution	1 634 461
Coq replaying SAT	8.223
.v file size for COQ SAT	193 748

Well-formedness invariants

...

$\text{wf} : \text{state} \rightarrow \text{prop} :=$

 |wf0 : wf_order s.order s.reason_of \rightarrow

 wf_watches s.clauses s.watches \rightarrow

$l \in \text{s.order} \leftrightarrow \text{s.assign}(\text{var_of } l) = \text{sign}(l) \rightarrow$

 wf s

...

Algorithm correction invariant

$\text{invS } F \phi \text{ s} :=$
 $\llbracket F \rrbracket_{\phi} \rightarrow$
 $(\llbracket \text{s.clauses} \rrbracket_{\phi} \wedge$
 $(\forall x \in \text{Var}, \text{s.reason_of } \text{var} = (0, _) \rightarrow$
 $(\text{s.assign } x = \llbracket x \rrbracket_{\phi} \vee \text{s.assign } x = \perp)))$

...

Current state of the project

- flexible and efficient external solver, with Caml parser for output format
- efficient (and almost proved) Coq resolution Checker
- efficient Caml preprocessor and (unproved) Coq checker for the replaying version

Second step

- SAT solver is a part of SMT solver
- second step : building a complete SMT tactic in Coq using SAT tactic

Our idea :

- run external SMT solver,
- record theory lemmas built by the SMT
- prove theory lemmas in Coq with a specific tactic for each theory
- solve the SAT problem with SAT tactic

Proof of concept

Demo!

That's all folks!

Any questions?