

Developing a Modern SAT Solver with Correctness Verified

Duckki Oe

Computer Science, The University of Iowa, USA

What is a SAT Solver?

An input formula in CNF

$$\begin{aligned} & (a \vee b) \\ \wedge & (\neg b \vee c) \\ \wedge & (\neg a \vee c \vee d) \\ \wedge & (\neg a \vee \neg d \vee b) \\ \wedge & (d \vee \neg c) \end{aligned}$$

Boolean variables: a, b, c, d, \dots

SAT solver decides the satisfiability of a formula.

- ▶ Model: an assignment to variables that makes the formula true.
- ▶ SAT if the formula has a model
- ▶ UNSAT if the formula has a contradiction (thus, no model)

Certificates

- ▶ SAT instance: a model that is found by the solver
- ▶ UNSAT instance: a refutational proof (deduction sequence to false)
- ▶ Certificates can be checked by a trusted verifier.

Overhead

- ▶ Generating models is now standard and cheap to check.
- ▶ Benchmarks for SAT competition can generate proofs of GBytes in size.
- ▶ SAT is not in co-NP. (Proofs can be exponentially big.)

Static Verification of Correctness

Focus on the Correctness of UNSAT Answers

- ▶ SAT certificates have very low overhead to implement and check.
- ▶ Why bother to statically verify the code for SAT?

Verify the solver's code to ensure that

- ▶ When it says UNSAT, it could generate a proof that will check.
- ▶ But, not actually generating it in run-time.
- ▶ The code should have information how to construct such a proof.

Clause Database and Detecting Contradiction

- 1: $a \vee b$ (by Assump)
- 2: $\neg b \vee c$ (by Assump)
- 3: $c \vee \neg a$ (by Assump)
- 4: $a \vee c$ (by Res 1, 2)
- 5: c (by Res 4, 3)
- \vdots
- n: \perp

$$\frac{C \vee I \quad D \vee \neg I}{C \vee D} \text{ Res}$$

- ▶ New clauses are deduced and added.
- ▶ Solver returns UNSAT when the empty clause is deduced.
- ▶ Each deduction has a reasoning behind it.
- ▶ Check the algorithm if the Res rule is applied correctly.
- ▶ Challenge: practical SAT solvers are highly optimized to apply a sequence of resolutions fast.

Most solvers in SAT competition are written in C/C++.

- ▶ Those SAT solvers are highly optimized for speed.
- ▶ They are not meant to be verified.

Essential Features: (to be "reasonably fast")

- ▶ Fast Boolean Constraint Propagation (using watch-lists)
- ▶ Fast Conflict clause analysis and learning
- ▶ Backjumping

The Guru Programming Language

Guru is a functional programming language with:

- ▶ Dependent type system (for verification)
- ▶ Resource type system (for efficient code generation)

Published Papers:

- ▶ PLPV(2010) Resource Typing in Guru. *Stump and Austin*
- ▶ PLPV(2009) Verified Programming in Guru. *Stump, et al.*
- ▶ PSTT(2009) Deciding Joinability Modulo Ground Equations in Operational Type Theory. *Petcher and Stump*

Dependently Typing a SAT Solver

The type of the `solve` function

```
Define clause := <list lit>.
Define formula := <list clause>.
Define solve : Fun(F:formula).<answer F> := ...
```

The answer type

```
Inductive answer : Fun(F:formula).type :=
  sat : Fun(spec F:formula).<answer F>
| unsat : Fun(spec F:formula)(spec p:<pf F (nil lit)>>).<answer F>
```

- ▶ The `<pf F C>` type enforces that $F \vdash C$.
- ▶ `spec` arguments are specificational, not actually constructed at run-time.

The `pf` type: derivation proofs

```
Inductive pf : Fun(F : formula)(C:clause).type :=
  pf_asm : Fun(F : formula)(C:clause)
            (u : { (member C F eq_clause) = tt }).    <pf F C>
| pf_res : Fun(F : formula)(C1 C2 Cr : clause)(l:lit)
            (d1 : <pf F C1>)
            (d2 : <pf F C2>)
            (u : { (is_resolvent Cr C1 C2 l) = tt }). <pf F Cr>
```

- ▶ `is_resolvent` is a logical definition (simple/slow).
- ▶ Need to prove it from the actual optimized code.
- ▶ It also requires several invariants across parts of solver.

Efficient Representation of Clauses

`aclause` type: wrapper for array-based clause and invariants

```
Inductive aclause : Fun(nv:word)(F:formula).type :=
  mk_aclause : Fun(spec nv:word)(spec F:formula)
    (spec n:word)(l:<array lit n>)
    (u1: (array_in_bounds nv l) = tt )
    (spec c:clause)(spec pf_c:<pf F c>)
    (u2: c = (to_cl l) )
  .<aclause nv F>
```

- ▶ `aclause` relates an **array-based clause** implementation and a **list-based clause** specification.

Efficient Representation of Clauses

`aclause` type: wrapper for array-based clause and invariants

```
Inductive aclause : Fun(nv:word)(F:formula).type :=
  mk_aclause : Fun(spec nv:word)(spec F:formula)
    (spec n:word)(l:<array lit n>)
    (u1: (array_in_bounds nv l) = tt )
    (spec c:clause)(spec pf_c:<pf F c>)
    (u2: c = (to_cl l) )
  .<aclause nv F>
```

- ▶ `to_cl` interprets a null-terminated array as a list. (Trusted)

Efficient Representation of Clauses

`aclause` type: wrapper for array-based clause and invariants

```
Inductive aclause : Fun(nv:word)(F:formula).type :=
  mk_aclause : Fun(spec nv:word)(spec F:formula)
    (spec n:word)(l:<array lit n>)
    (u1: (array_in_bounds nv l) = tt )
    (spec c:clause)(spec pf_c:<pf F c>)
    (u2: c = (to_cl l) )
    .<aclause nv F>
```

- ▶ Array bounds are statically checked.
- ▶ Implementation utilizes lookup tables indexed by variable number.
- ▶ `nv` is the maximum variable number.
- ▶ **Invariant**: each variable in the clause \leq `nv`.

Conclusion

`versat` is a SAT solver with modern features

- ▶ Goal: comparable performance
- ▶ Correct deduction is specified and enforced through types.
- ▶ Derivation of the empty clause is enforced for UNSAT answer.

Status

- ▶ Specification: 258 lines (Trusted).
- ▶ Code and Proofs: 2700 lines
- ▶ Backjumping implemented.
- ▶ CC analysis is implemented, but need to be optimized.
- ▶ Fast BCP is to be implemented