A Verified SAT Solver with Watched Literals Using Imperative HOL (Extended Abstract)

Mathias Fleury[1] and Jasmin Christian Blanchette[2,1] and Peter Lammich[3,4]

[1] Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
{mathias.fleury,jasmin.blanchette}@mpi-inf.mpg.de
[2] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
j.c.blanchette@vu.nl
[3] Technische Universität München, Munich, Germany
lammich@in.tum.de
[4] Virginia Tech, Blacksburg, Virginia, USA
lpeter1@vt.edu

Abstract. Based on our earlier formalization of conflict-driven clause learning (CDCL) in Isabelle/HOL, we refine the CDCL calculus to add a crucial optimization: two watched literals. We formalize the data structure and the invariants. Then we refine the calculus to obtain an executable SAT solver. Through a chain of refinements carried out using the Isabelle Refinement Framework, we target Imperative HOL and extract imperative Standard ML code. Although our solver is not competitive with the state of the art, it offers acceptable performance for some applications, and heuristics can be added to improve it further.

SAT solvers are programs that decide the Boolean satisfiability problem. Other NP-complete problems are often reduced to SAT, to exploit efficient SAT solvers. For example, the CeTA (non)termination and (non)confluence checker [19], which is formalized in Isabelle/HOL, includes a naive SAT solver based on a disjunctive normal form transformation. Using a verified SAT solver that is reasonably efficient could greatly improve the checker's performance.

We recently formalized an abstract calculus for conflict-driven clause learning (CDCL) in Isabelle/HOL [4], following Nieuwenhuis, Oliveras, and Tinelli [15]. CDCL is the core of most modern SAT solvers. It generalizes the Davis–Putnam–Logemann–Loveland (DPLL) procedure [7] with clause learning and nonchronological backjumping. We also formalized a CDCL variant due to Weidenbach, described in a paper [20] and in an unpublished book draft, that explores first unique implication points [3, Chapter 3] to learn clauses.

In this paper, we connect the formalized metatheory of CDCL with the code of an imperative SAT solver that implements several key optimizations found in modern CDCL-based solvers. We start by extending Weidenbach's backjumping rule to minimize conflict clauses [18].

A crucial optimization in modern SAT solvers is the two-watched-literal [14] data structure. It allows for efficient unit propagation and conflict detection—the core CDCL operations. We introduce an abstract transition system, called

TWL, that captures the essence of a SAT solver with this optimization as a nondeterministic transition system. Weidenbach's book draft only presents the main invariant, without a precise description of the optimization. We enrich the invariant based on MiniSat's [8] source code and prove that it is maintained by all transitions.

To get an executable program that can be incorporated into CeTA, we refine the TWL calculus in several correctness-preserving steps. The stepwise refinement methodology enables us to inherit invariants, correctness, and termination from previous refinement steps. The first refinement step implements the rules of the calculus in a more algorithmic fashion, using the nondeterministic programming language provided by the Isabelle Refinement Framework [11]. The next step refines the data structure: Multisets are replaced by lists, and clauses justifying propagations are represented by indices into a list of clauses. A key ingredient for an efficient implementation of watched literals is a data structure called *watch lists*. These index the clauses by their two watched literals—literals that can influence their clauses' truth value in the solver's current state. Watch lists are introduced in a separate refinement step.

Next, we use the Sepref tool [12] to synthesize imperative code for a functional program, together with a refinement proof. Sepref replaces the abstract functional data structures by concrete imperative implementations, while leaving the algorithmic structure of the program unchanged. Isabelle's code generator can then be used to extract a self-contained SAT solver in imperative Standard ML. Finally, to obtain reasonably efficient code, we need to implement further optimizations and heuristics. In particular, the literal selection heuristic is crucial. We use variable move to front [2] with phase saving [17].

To measure the gap between our solver, *IsaSAT*, and the state of the art, we compare IsaSAT's performance with four other solvers: the leading solver Glucose [1]; the well-known MiniSat [8]; and the most efficient verified solver we know of, `versat` [16]. Although our solver is competitive with `versat`, the results are sobering. They confirm the view that the generation of and checking of unsatisfiability certificates is the superior approach to combine efficiency and trustworthiness [5, 6, 13]. Compared with other verified SAT solvers, the hallmark of our solver is its modularity. New heuristics can be incorporated, and further refinement steps can be performed if desired. Furthermore, our solver is guaranteed to terminate.

Much of the scientific value of formalization is that it constitutes a case study in the use of a proof assistant. We depend heavily on Isabelle's Refinement Framework. We especially benefit from its ability to align program steps, allowing us to focus on the changes between subsequent programs in the refinement chain. The Sepref tool simplifies the last refinement step by generating imperative code and a corresponding refinement theorem. It makes it easy to change data structures. The refinement approach encourages a clean separation of concerns; for example, termination can be proved at the abstract, calculus level, and optimizations can be considered in isolation. Although IsaSAT is not as efficient as the state of the art, our work suggests that refinement could be applied further to derive competitive SAT solvers.

Our formalization is available online as part of the Isabelle Formalization of Logic (IsaFoL) repository [9]. The contributions of this paper correspond to the theory files with names matching the patterns `Watched_Literals_*.thy` and `IsaSAT*.thy`. They amount to about 31 000 lines of Isabelle text.

This extended abstract is based on our CPP paper [10] which is available online.[5]

References

[1] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009. pp. 399–404. ijcai.org (2009)

[2] Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer (2015)

[3] Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)

[4] Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 25–44. Springer (2016)

[5] Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 220–236. Springer (2017)

[6] Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 118–135. Springer (2017)

[7] Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)

[8] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer (2003)

[9] Fleury, M., Blanchette, J.C.: Formalization of Weidenbach's *Automated Reasoning—The Art of Generic Problem Solving* (2017), `https://bitbucket.org/isafol/isafol/src/master/Weidenbach_Book/README.md`, Formal proof development

[10] Fleury, M., Blanchette, J.C., Lammich, P.: A verified SAT solver with watched literals using imperative HOL. In: CPP. pp. 158–171. ACM (2018)

[11] Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer (2013)

[12] Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer (2015)

[13] Lammich, P.: The GRAT tool chain—efficient (UN)SAT certificate checking with formal correctness guarantees. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 457–463. Springer (2017)

[14] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC 2001. pp. 530–535. ACM (2001)

[15] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)

[5] `http://people.mpi-inf.mpg.de/~mfleury/sat_twl.pdf`

[16] Oe, D., Stump, A., Oliver, C., Clancy, K.: `versat`: A verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012, LNCS, vol. 7148, pp. 363–378. Springer (2012)

[17] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer (2007)

[18] Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 9340, pp. 237–243. Springer (2009)

[19] Thiemann, R., Sternagel, C.: Certification of termination proofs using ceta. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer (2009)

[20] Weidenbach, C.: Automated reasoning building blocks. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday. LNCS, vol. 9360, pp. 172–188. Springer (2015)