A verified SAT solver with watched literals using imperative HOL

Mathias Fleury, Jasmin Christian Blanchette, Peter Lammich

A Verified SAT Solver with Watched Literals Using Imperative HOL

Mathias Fleury
Max-Planck-Institut für Informatik
Saarbrücken, Germany
mathias.fleury@mpi-inf.mpg.de

Jasmin Christian Blanchette
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
j.c.blanchette@vu.nl
Max-Planck-Institut für Informatik
Saarbrücken, Germany
jasmin.blanchette@mpi-inf.mpg.de

Peter Lammich
Technische Universität München
Munich, Germany
lammich@in.tum.de
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.

CCS Concepts • **Computing methodologies → Theorem proving algorithms**; • **Software and its engineering → Software verification**; **Formal software verification**;

Keywords Isabelle/HOL, SAT solving, conflict-driven clause learning (CDCL), stepwise refinement

ACM Reference Format:
Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. 2018. A Verified SAT Solver with Watched Literals Using Imperative HOL. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3167080

1 Introduction

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 [36], 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 [7], following Nieuwenhuis, Oliveras, and Tinelli [29]. CDCL is the core of most modern SAT solvers. It generalizes the Davis–Putnam–Logemann–Loveland (DPLL) procedure [11] with clause learning and nonchronological backjumping. We also formalized a CDCL variant due to Weidenbach, described in a paper [37] and in an unpublished book draft, that explores first unique implication points [6, 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 [34] (Section 2).

A crucial optimization in modern SAT solvers is the two-watched-literal [28] 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 (Section 3). Weidenbach's book draft only presents the main invariant, without a precise description of the optimization. We enrich the invariant based on MiniSat's [13] 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 [19] (Section 4). 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 (Section 5). 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

CPP'18, January 8–9, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167080

clauses' truth value in the solver's current state. Watch lists are introduced in a separate refinement step (Section 6).

Next, we use the Sepref tool [20] 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 (Section 7). Finally, to obtain reasonably efficient code, we need to implement further optimizations and heuristics (Section 8). In particular, the literal selection heuristic is crucial. We use variable move to front [5] with phase saving [31].

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 [13]; the OCaml-based DPT;[1] and the most efficient verified solver we know of, `versat` [30] (Section 9). 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 [9, 10, 22]. 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 [14]. The theorems referenced in this paper are labeled with their Isabelle names for convenience. The contributions of this paper correspond to the theory files with names matching the patterns `Two_Watched_Literals_*.thy` and `IsaSAT*.thy`. They amount to about 31 000 lines of Isabelle text.

We have submitted an extended version of our earlier paper [7], on the formalization of CDCL calculi, to a journal. That article contains a section on the derivation of an imperative program. Here, we explain each refinement step in detail, with particular emphasis on the methodology and the proving technology used. We also show how to extend the calculus to minimize conflict clauses and include an empirical evaluation.

[1] http://dpt.sourceforge.net/

2 The CDCL Calculus

We define literals as a datatype *'v lit* with two constructors: Given an atom A of propositional logic, of type *'v* (variable), Pos A and Neg A are literals. The negation of a literal is defined by $-\,\text{Pos}\,A = \text{Neg}\,A$ and $-\,\text{Neg}\,A = \text{Pos}\,A$. As is customary in the literature [2, 37], a clause is a multiset of literals and has type *'v clause = 'v lit multiset*. Isabelle multisets are finite. We use logical symbols for clauses to ease reading when there is no risk of confusion, writing $\bot$, L, $C \vee D$ for $\emptyset$, $\{L\}$, and $C \uplus D$, respectively. A set of literals I *entails* a clause C ($I \models C$) if and only if I and C share a literal. I entails a (multi)set of clauses N if and only if I entails every clause in N.

In our previous paper [7] and in the corresponding journal submission, we presented two families of CDCL calculi and connected them through refinement steps. Here, we briefly describe the calculus that will serve as the starting point for further refinement. This calculus, which we call W, mostly follows Weidenbach's CDCL variant [37]. It operates on states (M, N, U, D), where M is the partial model under construction, or *trail*; N is the multiset of initial clauses; U is the multiset of learned clauses; and D is a conflict clause, or the special value $\top$ if no conflict has been detected. The multiset of initial clauses does not change during the execution of the calculus. In contrast, the multiset of learned clauses grows monotonically starting from $\emptyset$.

The trail M consists of a list of *annotated* literals. Each literal L in M can be a *decision*, written $L^\dagger$, or the result of a *unit propagation*, in which case it is annotated with the clause C that caused the propagation, written L^C. The *level* of a literal L in M is the number of decision literals to the right of the atom of L in M, or 0 if the atom is undefined. The level of a clause is the highest level of any of its literals, with 0 for $\bot$. The level of a state corresponds to the number of decision literals in the trail. We overload the infix operator $\cdot$ to denote both the Cons list constructor and list concatenation. In accordance with Isabelle conventions, the trail is extended on the left. In the following, we abuse notation, implicitly converting $\models$'s first operand from a list to a set and ignoring annotations on literals.

The calculus assumes that N contains no duplicate literals and never produces clauses containing duplicates. We write $S \Longrightarrow_{\text{W}} T$ if the calculus makes a transition from state S to state T. The following transitions are possible:

Propagate $(M, N, U, \top) \Longrightarrow_{\text{W}} (L^{C \vee L} \cdot M, N, U, \top)$
if $C \vee L \in N \uplus U$, $M \models \neg C$, and L is undefined in M (i.e., neither $L \in M$ nor $-L \in M$)

Decide $(M, N, U, \top) \Longrightarrow_{\text{W}} (L^\dagger \cdot M, N, U, \top)$
if L is undefined in M and occurs in N

Conflict $(M, N, U, \top) \Longrightarrow_{\mathsf{W}} (M, N, U, D)$
if $D \in N \uplus U$ and $M \vDash \neg D$

Resolve $(L^{C \vee L} \cdot M, N, U, D \vee -L) \Longrightarrow_{\mathsf{W}} (M, N, U, C \cup D)$
if D has the level of the current state

Skip $(L^{C} \cdot M, N, U, D) \Longrightarrow_{\mathsf{W}} (M, N, U, D)$
if $L \notin D$ and $D \notin \{\bot, \top\}$

Jump $(M' \cdot K^{\dagger} \cdot M, N, U, D \vee L) \Longrightarrow_{\mathsf{W}}$
$(L^{D' \vee L} \cdot M, N, U \uplus \{D' \vee L\}, \top)$
if L has the level of the current state, D has a lower level, $D' \subseteq D$, $N \uplus U \vDash D' \vee L$, and D' has the same level as K

The W calculus works as follows. We start with an initial state $(\epsilon, N, \emptyset, \top)$ and repeatedly apply the rules until we reach a normal form. A state (M, N, U, D) is *conclusive* if $D = \top$ and $M \vDash N$ or if $D = \bot$ and N is unsatisfiable. Given a conclusive state, we can extract a solution to the SAT problem. The calculus always terminates, but there are no guarantees that the final state is conclusive. If we want this property, we need a strategy to restrict rule applications. Weidenbach's *reasonable* strategy favors Propagate and Conflict over all other rules. We call the calculus restricted by this strategy W+stgy.

Compared with Weidenbach's formulation, which formed the basis of our earlier formalization, the Jump rule has been generalized to learn $D' \vee L$ instead of the potentially larger clause $D \vee L$. Like its predecessor, this calculus can be seen as a special case of the abstract CDCL by Nieuwenhuis et al. [29]. Our formalization includes a proof of this connection.

Given a relation $\Longrightarrow$, we write $S \Longrightarrow^{!} T$ if state T is a normal form (i.e., there exists no transition from T) reachable from state S.

Theorem 2.1 (Correctness [14, *full_cdcl$_{\mathrm{W}}$_stgy_final_state_conclusive_from_init_state*]). *If $(\epsilon, N, \emptyset, \top) \Longrightarrow^{!}_{\mathrm{W+stgy}} S'$ and N contains no clauses with duplicate literals, then S' is a conclusive state.*

3 Watched Literals

The two-watched-literal (2WL or TWL) scheme [28] is a data structure that makes it possible to efficiently identify candidate clauses for unit propagation and conflict. In each nonunit clause (i.e., a clause with at least two literals), we distinguish two *watched* literals; the remaining literals are *unwatched*. Initially, any of a nonunit clause's literals can be chosen to be watched. The solver maintains the following *2WL invariant* for each clause:

> Unless a conflict has been found, a watched literal may be false only if the other watched literal is true and all the unwatched literals are false.

This is the invariant given by Weidenbach. It is inspired by MiniSat's code. A consequence of this invariant is that setting an unwatched literal will never yield a candidate for propagation or conflict. This can dramatically reduce the number of candidate clauses to consider.

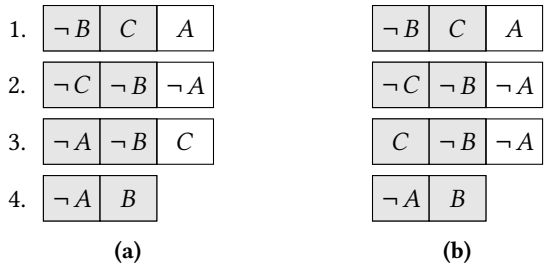


Figure 1. Evolution of the 2WL data structure on a simple example

For each literal L, the clauses that contain a watched L are chained together in a list, called a *watch list*. When a literal L becomes true, the solver needs to iterate only through the watch list for $-L$ to find candidates for propagation or conflict. For each candidate clause, there are four possibilities:

1. If the other watched literal is true, do nothing.
2. If one of the unwatched literals L' is not false, restore the invariant by *updating* the clause so that it watches L' instead of $-L$.
3. Otherwise, consider the other watched literal L' in the clause:
   3.1. If it is not set, propagate L'.
   3.2. Otherwise, L' is false, and we have found a conflict.

Propagation is performed eagerly. When a conflict is detected, the solver stops updating the data structure and processes the conflict.

To illustrate how the solver maintains the 2WL invariant, we consider the small problem shown in Figure 1. The clauses are numbered from 1 to 4. Gray cells identify the watched literals. Thus, clause 1 is $\neg B \vee C \vee A$, where $\neg B$ and C are watched. The following scenario is possible:

1. We start with an empty trail and the clauses shown in Figure 1a. We decide to make A true. The trail becomes $A^{\dagger}$. We need to consider every clause where $\neg A$ is watched, i.e., clauses 3 and 4, in any order.
2. We first consider clause 4 for $\neg A$. We propagate B from it. The trail becomes $B \cdot A^{\dagger}$. We still need to consider clause 3 for $\neg A$ and the clauses for $\neg B$.
3. We consider clause 3 for $\neg A$. Since C is unwatched and not false, we swap C and $\neg A$, resulting in the clauses shown in Figure 1b. We must still consider clauses 1, 2, and 3 for $\neg B$.
4. We consider clause 3 for $\neg B$: We propagate C. The trail becomes $C \cdot B \cdot A^{\dagger}$. We still need to update the clauses 1 and 2 for $\neg B$ and the clauses for $\neg C$.
5. We consider clause 2. All its literals are false—a conflict. Thanks to the invariant's precondition ("unless

a conflict has been found"), we do not need to update clause 1 or the clauses for $\neg C$.

Compatibility with the Jump rule is important for efficiency: When removing literals from the trail, the invariant is preserved without requiring any update.

To capture the 2WL data structure formally, we need a notion of state that takes into account pending updates. These can concern a specific clause or all the clauses associated with a literal. As in the example above, we first process the clause-specific updates; then we move to the next literal and start updating its associated clauses.

States have the form $(M, N, U, D, NP, UP, WS, Q)$, of type $'v\ state_{\mathsf{TWL}}$. The pending updates are stored in the last two components: the *work stack* WS is a multiset $\{(L, C_1), \ldots, (L, C_n)\}$, where L is a false literal and the clauses C_i watch L and may require an update. The other literals to update are stored in the *queue* Q. For example, at the end of step 4 above, WS is $\{(\neg B,\ \neg B \vee C \vee A),\ (\neg B,\ \neg C \vee \neg B \vee \neg A)\}$ and Q is $\{\neg C\}$.

Moreover, we store the unit clauses separately from the nonunit clauses. The unit clauses are put in the NP and UP components as singleton multisets. The nonunit clauses are put in N and U. Each nonunit clause is represented by a value $\mathsf{Clause}_{\mathsf{TWL}}\ W\ UW$, where W is the multiset of watched literals, of cardinality 2, and UW the multiset of unwatched literals.

The $\mathsf{state}_{\mathsf{W}}$_of function converts a TWL state to a W state:

definition $\mathsf{state}_{\mathsf{W}}$_of :: $'v\ state_{\mathsf{TWL}} \Rightarrow\ 'v\ state_{\mathsf{W}}$ **where**
$\mathsf{state}_{\mathsf{W}}$_of $(M, N, U, D, NP, UP, WS, Q)$ =
$(M,$ image $\mathsf{clause}_{\mathsf{W}}$_of $N \uplus NP,$
image $\mathsf{clause}_{\mathsf{W}}$_of $U \uplus UP, D)$

where $\mathsf{clause}_{\mathsf{W}}$_of $(\mathsf{Clause}_{\mathsf{TWL}}\ W\ UW) = W \uplus UW$ and image $f\ N$ applies the function f to each element of multiset N.

The first two TWL rules have direct counterparts in W:

Propagate $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}}$
$(L'^{C} \cdot M, N, U, \top, NP, UP, WS, \{-L'\} \uplus Q)$
if watched $C = \{L, L'\}$, L' is not set in M, and
$\forall K \in$ unwatched $C.\ -K \in M$

Conflict $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N, U, C, NP, UP, \emptyset, \emptyset)$
if watched $C = \{L, L'\}$, $-L' \in M$, and
$\forall K \in$ unwatched $C.\ -K \in M$

For both rules, C cannot be a unit clause. The condition $\forall K \in$ unwatched $C.\ -K \in M$ is necessary because the 2WL invariant trivially holds for C as long as an update on C is pending.

The next rules manipulate the state's 2WL-specific components, without affecting its semantics as seen through the function $\mathsf{state}_{\mathsf{W}}$_of:

Update $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N', U', \top, NP, UP, WS, Q)$
if $K \in$ unwatched C, $-K \notin M$, and N' and U' are obtained from N and U by replacing the clause $C = \mathsf{Clause}_{\mathsf{TWL}}\ W\ UW$ with $\mathsf{Clause}_{\mathsf{TWL}}\ (W - \{L\} \uplus \{K\})\ (UW - \{K\} \uplus \{L\})$

Ignore $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N, U, \top, NP, UP, WS, Q)$
if watched $C = \{L, L'\}$ and $L' \in M$

Next_Literal $(M, N, U, \top, NP, UP, \emptyset, \{L\} \uplus Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N, U, \top, NP, UP,$
$\{(L, C) \mid L \in$ watched $C \wedge C \in N \uplus U\}, Q)$

As in W+stgy, we postpone decisions. This is achieved by requiring that WS and Q are empty in the Decide rule. Skip and Resolve are as before, except that they also preserve the 2WL-specific components of the state. Due to the distinction between unit and nonunit clauses, we need two rules for nonchronological backjumping:

Decide $(M, N, U, \top, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{\dagger} \cdot M, N, U, \top, NP, UP, \emptyset, \{-L\})$
if L is not defined in M and appears in N

Jump_Nonunit $(M' \cdot K^{\dagger} \cdot M, N, U, D \vee L, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{D \vee L} \cdot M, N, U \uplus \{\mathsf{Clause}_{\mathsf{TWL}}\ \{L, L'\}\ (D' - \{L'\})\}, \top,$
$NP, UP, \emptyset, \{L\})$
if the conditions on Jump are satisfied by D, D', and L, $L' \in D$, and L' has the highest level among D''s literals

Jump_Unit $(M' \cdot K^{\dagger} \cdot M, N, U, D \vee L, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{L} \cdot M, N, U, \top, NP, UP \uplus \{L\}, \emptyset, \{L\})$
if the conditions on Jump are satisfied by D, $D' = \emptyset$, and L

In Jump_Nonunit, we need to choose a literal L' of D' with the highest level among D''s literals, or the next-highest level in $D' \vee L$ (since L has a higher level than L'). Jump_Nonunit is documented in MiniSat's code ("find the first literal assigned at the next-highest level"). Remarkably, this important property is mentioned neither in Weidenbach's book draft nor in the description of MiniSat [13].

Theorem 3.1 (Invariant [14, *cdcl_twl_stgy_twl_struct_invs*]). *If the state S satisfies the 2WL invariant and $S \Longrightarrow_{\mathsf{TWL}} T$, then T satisfies the 2WL invariant.*

Theorem 3.2 (Refinement [14, *full_cdcl_twl_stgy_cdcl$_{\mathsf{W}}$_stgy*]). *Let S be a state that satisfies the 2WL invariant. If $S \Longrightarrow^{!}_{\mathsf{TWL}} T$, then* $\mathsf{state}_{\mathsf{W}}$_of $S \Longrightarrow^{!}_{\mathsf{W}}$ $\mathsf{state}_{\mathsf{W}}$_of T.

TWL refines W+stgy's end-to-end behavior and produces final states that are also final states for W. We can apply Theorem 2.1 to establish partial correctness. Termination of TWL is a direct consequence of the termination of W.

4 Refining the Calculus to an Algorithm

We want to obtain an executable SAT solver from TWL. We do this by refining the calculus in multiple consecutive steps until we reach an implementation.

The Isabelle Refinement Framework [19, 20, 23] provides a tool chain for program development via stepwise refinement.

It is based on the *nondeterminism monad* over the datatype *'a nres* = FAIL | RES (*'a set*). If the program has an execution that diverges or raises an error, its result is FAIL; otherwise, the result is RES X, where X is the set of possible return values. The function RETURN x, which abbreviates RES $\{x\}$, returns the value x; bind m f nondeterministically chooses a return value from m and applies f to it. Based on these constructs and Isabelle's standard 'if–then–else' and 'case' expressions, the Refinement Framework defines higher-level constructs such as 'while' and 'for each' loops. The Haskell-style 'do' monadic notation is also supported: do $\{a \leftarrow m; f\,a\}$ is syntactic sugar for bind m f.

The first step in the refinement chain is to implement the calculus as a program in the nondeterminism monad. The program operates on states of type *'v* $state_{\mathsf{TWL}}$, as in the TWL calculus, but it reduces some of the calculus's nondeterminism. The program consists of a few functions that implement mutually disjoint sets of rules. We focus on the function that applies Propagate, Conflict, Update, or Ignore, assuming that its first argument, the pair $LC = (L, C)$, has already been removed from the *WS* component of S:

```
definition PCUI_algo ::
  'v lit × 'v clause ⇒ 'v state_TWL ⇒ 'v state_TWL
where
  PCUI_algo LC S = do {
    let (M, N, U, D, NP, UP, WS, Q) = S;
    let (L, C) = LC;
    L' ← RES {L' | L' ∈ watched C − {L}};
    if L' ∈ M then                                  (* Ignore *)
      RETURN S
    else
      if ∀L ∈ unwatched C. −L ∈ M
        if −L' ∈ M then                             (* Conflict *)
          RETURN (M, N, U, C, NP, UP, ∅, ∅)
        else                                        (* Propagate *)
          RETURN (L'^C · M, N, U, D, NP, UP, WS,
              {−L'} ⊎ Q)
      else do {                                     (* Update *)
        K ← RES {K | K ∈ unwatched C ∧ −K ∉ M};
        (N', U') ← RES {(N', U') |
            update_clss (N, U) C L K (N', U')};
        RETURN (M, N', U', D, NP, UP, WS, Q)
      }
  }
```

The predicate update_clss (N, U) C L K (N', U') updates the clause C by exchanging the watched literal L and the unwatched literal K in C. The clause is updated in N and U, yielding N' and U'. Since propagations are performed eagerly, *WS* never refers to unit clauses.

The $\mathsf{PCUI}_{\mathsf{algo}}$ algorithm still contains abstract, nondeterministic parts. For example, in the Update part, we leave the choice of the new watched literal K underspecified.

To allow us to specify the connection between two programs, the Refinement Framework defines a partial order $\leq$ on *'a nres*, with FAIL as the top element: RES $X \leq$ RES Y if and only if $X \subseteq Y$, and $r \leq$ FAIL for all r. The bottom element RES {} is an unrefinable program. We also use this partial order to state program correctness: The statement $P\,x \Longrightarrow f\,x \leq \mathsf{RES}\,\{y \mid Q\,y\}$ expresses the total correctness of program f with precondition P and postcondition Q. For $\mathsf{PCUI}_{\mathsf{algo}}$, we have the following refinement theorem:

Lemma 4.1 (Refinement [14, *unit_propagation_inner_loop_body_add*]). *If the 2WL invariant holds for all clauses occurring in the N and U components of S, then*

$$\mathsf{PCUI}_{\mathsf{algo}}(L, C)S \leq \mathsf{RES}\,\{T \mid \mathsf{add\_to\_WS}\,(L, C)\,S \Longrightarrow_{\mathsf{PCUI}} T\}$$

The PCUI subscript on the transition arrow refers to the fragment of TWL consisting of the four rules Propagate, Conflict, Update, and Ignore, whereas add_to_WS (L, C) S returns the state obtained by adding (L, C) to S's *WS* component. For the entire SAT solver, we have the following theorem:

Theorem 4.2 (Refinement [14, *cdcl_twl_stgy_prog_spec*]). *If the 2WL invariant holds for all clauses occurring in the N and U components of S, then*

$$\mathsf{TWL}_{\mathsf{algo}}\,S \leq \mathsf{RES}\,\{T \mid S \Longrightarrow^{!}_{\mathsf{TWL}} T\}$$

The state returned by the program is a final state for TWL. From Theorem 3.2, we deduce that it is also a final state for W+stgy. Hence, the program $\mathsf{TWL}_{\mathsf{algo}}$ is a SAT solver by Theorem 2.1.

5 Representing Clauses as Lists

The nondeterministic program $\mathsf{TWL}_{\mathsf{algo}}$ presented in Section 4 relies on the same state type as the TWL calculus. This changes with the next refinement step: We now store the initial and learned clauses together in a list, and we use indices to refer to the clauses. States are now tuples $(M, NU, u, D, NP, UP, WS', Q)$:

- NU is the list of all nonunit clauses. It simultaneously refines N and U. The initial clauses occupy indices 1 to $u - 1$, and the learned clauses start at index u. The list's first element is left unused to keep index 0 as a null clause reference.
- M is the trail, where the annotations are replaced by numeric indices. For nonunit clauses, L^i is used instead of L^C if $NU\,!\,i = C$, where the ! operator denotes 0-based list access. When annotating literals with unit clauses (which are not present in NU), we use the special index 0—i.e., we put L^0 on the trail to mean L^L.
- In WS', we implement a pair (L, C) by the index of clause C. The literal L, which is the same for all pairs in *WS*, is stored locally in the refined unit propagation algorithm.

Abusing notation, we will use the letter C to refer to clause indices and will not distinguish between a clause and its index.

In addition to the modifications to the state, we also transform the representation of clauses, from a pair of multisets holding the watched and unwatched literals to a list of literals such that its first two elements are watched. Given a nonunit clause (index) C, its watched literals are available as $(NU\ !\ C)\ !\ 0$ and $(NU\ !\ C)\ !\ 1$. Furthermore, we set the stage for future refinements by replacing the test $L \in M$ by a call to a function, polarity, that returns Some True if $L \in M$, Some False if $-L \in M$, and None otherwise.

The refined version of the $\text{PCUI}_{\text{algo}}$ algorithm follows:

```
definition PCUI_list ::
  'v lit ⇒ 'v clause_idx ⇒ 'v state_list ⇒ 'v state_list
where
  PCUI_list L C S = do {
    let (M, NU, u, D, NP, UP, WS, Q) = S;
    let i = if (NU ! C) ! 0 = L then 0 else 1;
    let L' = (NU ! C) ! (1 − i);
    let pol' = polarity M L';
    if pol' = Some True then                    (* Ignore *)
      RETURN (M, NU, u, D, NP, UP, WS, Q)
    else
      case find_unwatched M (NU ! C) of
        None ⇒
        if pol' = Some False then               (* Conflict *)
          RETURN (M, NU, u, NU ! C, NP, UP, ∅, ∅)
        else                                    (* Propagate *)
          RETURN (L'^C · M, NU, u, D, NP, UP, WS,
              {−L'} ⊎ Q)
      | Some j ⇒ do {                           (* Update *)
          let NU' = list_update NU C
              (list_swap (NU ! C) i j);
          RETURN (M, NU', u, D, NP, UP, WS, Q)
        }
  }
```

Refinement between a concrete program g and an abstract program f is expressed by a statement $\forall x\ y.\ (y, x) \in R \Longrightarrow g\ y \le \Downarrow_S f\ x$, where R is a relation between concrete and abstract arguments, and S is a relation between concrete and abstract results. The function $\Downarrow_S$ maps an abstract result to the largest concrete result whose value is related, by S, to the abstract value. Two edge cases are $\Downarrow_S \text{FAIL} = \text{FAIL}$ and $\Downarrow_{\text{Id}} r = r$. The former implies that a failing assertion is refinable by any program, so we can assume in the refinement proof that assertions on the abstract side never fail. The latter makes correctness a special case of refinement, the abstract program being of the form $\text{RES}\ \{x \mid Q\ x\}$.

The Refinement Framework includes a verification condition generator. Hoare-logic-style rules are used to prove correctness goals. For other refinement goals, a heuristic tries to align the concrete and abstract program structure and generates refinement goals for corresponding statements of the two programs. In the refinement proof of the PCUI algorithm, the goal for the definition of the other watched literal L' is $(NU\ !\ C)\ !\ (1 - i) \in \text{watched}\ C - \{L\}$. Intuitively, this holds because for $i \in \{0, 1\}$, the expression $1 - i$ returns the index of the other watched literal.

The generated goals are often easy to discharge with standard Isabelle tactics, but they may also point to missing lemmas or invariants. The primary technical challenge during proof development is to handle cases where the verification condition generator fails to properly align the programs and generates nonsensical, and usually unprovable, proof obligations. In some cases, the tool generates error messages, but these are often cryptic. Another hurdle is that refinement proof goals can be very large, and the Isabelle/jEdit graphical interface is painfully slow at displaying them. We suspect that this is mostly due to type annotations and other metainformation available as tooltips.

6 Storing Clauses Watched by a Literal: Watch Lists

In the Next_Literal rule of the TWL calculus, the set of clauses that watch a given literal is calculated. A refinement step eliminates this gratuitous inefficiency: Instead of iterating over all clauses, we maintain a map from literals to the clauses that contain them as watched literals. States now have the form $(M, NU, u, D, NP, UP, Q, W)$, where $W :: 'v\ \textit{lit} \Rightarrow \textit{clause\_idx list}$ maps each literal to its watch list.

The abstract state stores all the clauses that watch the current literal L and still require processing in its WS component. In the concrete algorithm, we use a local variable w to traverse the watch list. After processing a clause, there are two cases. If the clause still watches L (rules Propagate, Conflict, and Ignore), we increment w to move to the next clause. Otherwise, the clause no longer watches L (rule Update). We exchange the element at index w with the watch list's last element and shorten the list by one (function delete_idx_and_swap). Since the traversal order is irrelevant, this is an efficient way to delete an element in constant time based on arrays. This technique is implemented in many solvers.

The refined PCUI algorithm is presented below, where the syntax $f(x := y)$ denotes the function that maps x to y and otherwise coincides with f:

```
definition PCUI_wlist ::
  'v lit ⇒ nat ⇒ 'v state_wlist ⇒ nat × 'v state_wlist
where
  PCUI_wlist L w S = do {
    let (M, NU, u, D, NP, UP, Q, W) = S;
    let C = W L ! w;
    let i = if C ! 0 = L then 0 else 1;
    let L' = (NU ! C) ! (1 − i);
    let pol' = polarity M L';
    if pol' = Some True then                    (* Ignore *)
```

```
      RETURN (w + 1, (M, NU, u, D, NP, UP, Q, W))
    else
      case find_unwatched M (NU ! C) of
        None ⇒
        if pol' = Some False then          (* Conflict *)
          RETURN (w + 1, (M, NU, u, NU ! C, NP, UP,
              ∅, W))
        else                               (* Propagate *)
          RETURN (w + 1, (L'^C · M, NU, u, D, NP, UP,
              {−L'} ⊎ Q, W))
      | Some j ⇒ do {                      (* Update *)
          let K = (NU ! C) ! j;
          let NU' = list_update NU C
              (list_swap (NU ! C) i j);
          let W' =
            W(L := delete_idx_and_swap (W L) w)
              (K := W K · C);
          RETURN (w, (M, NU', u, D, NP, UP, Q, W'))
        }
  }
```

When performing a chain of refinements, we often want to reuse information from earlier refinement steps. Assume that we have previously shown the refinement relation

$$g\ y \le \Downarrow_{\{(t,s)\in R \mid I_1\ t \wedge I_2\ t\ s\}} f\ x, \tag{1}$$

where R relates concrete and abstract states and I_1 and I_2 are invariants. Now suppose we want to refine g by the function h with relation S and invariant J. The invariant J typically consists of a genuinely new part J_{new} and a part inherited from higher abstraction levels. We first prove the new part:

$$h\ z \le \Downarrow_{\{(u,t)\in S \mid J_{\text{new}}\ u\ t\}} g\ y \tag{2}$$

Then we can combine it with equation (1), using the invariant I_1 that does not depend on a state s, yielding

$$h\ z \le \Downarrow_{\{(u,t)\in S \mid J_{\text{new}}\ u\ t \wedge I_1\ t\}} g\ y \tag{3}$$

Finally, we can prove the desired refinement relation $h\ z \le \Downarrow_{\{(u,t)\in S \mid J\ u\ t\}} g\ y$, by showing the inclusion

$$\{(u,t) \in R \mid J_{\text{new}}\ u\ t \wedge I_1\ t\} \subseteq \{(u,t) \in R \mid J\ u\ t\} \tag{4}$$

Because we frequently needed to combine large invariants to derive refinement lemmas such as (3), we developed a specialized tactic in the Eisbach language [27]. It takes as input the relations (1) and (2). It separates I_1 and I_2, based on their syntactic dependencies, and derives the relation (3). Another Eisbach tactic takes (3) and the desired refinement goal as arguments and leaves (4) as the goal. Eisbach is very useful for such tedious but straightforward manipulations, especially for goals containing large formulas.

7 Generating Code

In the last refinement steps, we introduce mutable data structures. To represent code with side effects, we use Imperative HOL [8], a library for Isabelle/HOL that is based on the *heap monad* over the datatype *'a Heap* = Heap (*heap* ⇒ (*'a* × *heap*) *option*). A program (of type *'a Heap*) takes a memory state (of type *heap*) as input and returns, on success, a value (of type *'a*) and a new memory state.

For technical reasons, we need an intermediate refinement step between the introduction of watch lists (Section 6) and the change of data structures. This step amounts to adding assertions in the watch list algorithms stating that all literals belong to a fixed, finite domain. Given the set of all literals $\mathcal{L}_{\text{in}}$ that appear in the clause set N, we need to consider only the literals that appear in $\mathcal{L}_{\text{in}}$ or whose negation appear in $\mathcal{L}_{\text{in}}$. We call this set $\mathcal{L}_{\text{all}}$. The intermediate refinement step involves stating and discharging assertions of the form $L \in \mathcal{L}_{\text{all}}$. This sets the stage for many subsequent optimizations, by allowing us to allocate arrays that are large enough to represent mappings from atoms or literals. Arrays are used for watch lists (which map literals to clauses), polarity caching (which map atoms to their polarities, corresponding to the polarity function), and other optimizations.

To generate imperative code, the Sepref tool [20] bridges the gap between the Refinement Framework and Imperative HOL. It can be used to automatically replace the operations of the abstract algorithm in the nondeterminism monad by their concrete deterministic counterparts in the heap monad. The outcome is a deterministic imperative program and a refinement theorem linking it to the original abstract program.

Some of the data structures we need are already available in the Imperative Collections Framework [21], while others must be developed specifically for this project. Since the code in Imperative HOL is deterministic, we must commit to a strategy for applying the calculus rules. The precise heuristics and other optimizations are described in Section 8.

The solver state is enriched with information necessary for optimizations and heuristics, and its components are implemented by efficient data structures. For example, literals are refined to 32-bit unsigned integers, representing a positive literal Pos i by $2 \cdot i$ and a negative literal Neg i by $2 \cdot i + 1$. All required operations, such as atom extraction and negation, can be efficiently implemented on this representation. The use of 32-bit numbers restricts our implementation to at most 2^{31} atoms (which seems to be a common restriction for SAT solvers).

The encoding of literals as unsigned integers can be used to represent a map from literals by an array, indexed by the literal representation. In this way, we implement the W function that maps literals to its watch lists by an array of arrays. The outer array's size is determined by the actual number of atoms in the problem, while we use a dynamic resizing strategy for the inner arrays that hold the watch lists. Using the same literal encoding, clauses are represented by arrays of 32-bit integers. In contrast, the indices used as annotations in the trail and in the *WS* component are unbounded integers.

Internally, the refinement of the state is done in two steps: The first step handles the addition of the data for optimizations and heuristics, and the second step uses Sepref to refine the functional representations of the state's components to efficient mutable data structures.

To obtain a complete SAT solver, we must provide code to initialize the data structure with the 2WL invariant using the list of the atoms in the problems. Initialization works as follows: We first go through the clauses and extract all the atoms by taking the literal where it occurs first. Then for each clause, either it contains at least two literals, in which case the first two are watched, or it is a unit clause, in which case the literal is propagated (or a conflict is marked) and the clause is added to *NP*. If there is a conflict, there is no need to analyze it—the clauses are unsatisfiable.

Once we have refined TWL into an imperative program and combined it with a function initializing the data structure from a list of clauses, we define the complete imperative SAT solver as a function $\text{IsaSAT}_{\text{code}}$ in Imperative HOL. The abstract specification of the solver is given by

$$\begin{aligned}&\text{model\_if\_satisfiable} = \\ &\text{RES } \{M \mid \text{if satisfiable } CS \text{ then } M \neq \text{None} \wedge \text{the } M \models CS \\ &\qquad\qquad \text{else } M = \text{None}\}\end{aligned}$$

where the (Some x) = x. This abstract program returns None if the input clauses are unsatisfiable; otherwise, it returns Some M, where M is a model of the clauses. By combining the refinement theorems for all refinement steps, we obtain end-to-end correctness for the entire solver.

Theorem 7.1 (End-to-End Correctness [14, *IsaSAT_code_full_correctness*])**.** *The imperative SAT solver returns a model if its input is satisfiable:*

$$\begin{aligned}&(\text{IsaSAT}_{\text{code}}, \text{model\_if\_satisfiable}) \\ &\in [\text{no\_duplicate\_no\_false}]\ \text{clauses\_assn}^{\text{k}} \to \\ &\text{option\_assn (list\_assn lit\_assn)}\end{aligned}$$

The refinement predicate $(p', p) \in [precond]\ arg \to res$ means that if the precondition *precond* holds on the arguments and if the arguments are refined by the relation *arg*, then the two programs p' and p return a result refined by the relation *res*. The clauses_assn relation refines a multiset of multisets of literals to a list of lists of 32-bit literals, and option_assn (list_assn lit_assn) refines an optional list of literals to an optional list of 32-bit literals.

Finally, we invoke Isabelle's code generator [16] to extract Standard ML code from the Imperative HOL program. The result is a self-contained program consisting of about 2700 lines of code. It is extended with a simple unverified parser for SAT problems in conjunctive normal form. To give a flavor of the program, we show its main loop below (slightly reformatted for readability):

```
fun IsaSAT_code initial_state () =
  let val (_, final_state) =
    heap_WHILET
      (fn (done, _) => fn () => not done)
      (fn (_, T) =>
         analyze_or_decide_code
           (PCUI_and_Next_Literal T ()) ())
      (false, initial_state) ()
  in final_state end
```

8 Optimizations and Heuristics

Our imperative SAT solver relies on a few optimizations that deserve to be explained in more detail: an efficient decision heuristic, a representation of conflicts as a lookup table, conflict clause minimization, and the elimination of redundant components from the state.

8.1 Variable Move to Front

The variable-move-to-front (VMTF) heuristic [5], based on the move-to-front algorithm [32], selects which atom to decide next. It offers similar performance to the better-known variable-state-independent-decaying-sum (VSIDS) scheme [28]. VMTF's main advantage, from a formalization point of view, is that it does not require floating-point arithmetic.

VMTF works on a list of atoms *As*, which must contain all atoms from $\mathcal{L}_{\text{in}}$ in some order. Two operations access or modify this list: When a decision is needed, VMTF traverses *As* to *find* the first unset atom with respect to the trail. When an atom is heuristically determined to be important to the problem, it is moved to the front of *As* so that it is found next—an operation called *rescoring*.

To speed up these operations, we implement some of the optimizations described by Biere and Fröhlich [5]:

- To efficiently remove atoms from *As*, we represent it by a doubly linked list. Moreover, we store it in an array *ns* indexed by the atoms, enabling fast accesses to the associated nodes. Each entry in *ns* has the form (*st*, *prev*, *next*), where *st* is the timestamp indicating when the atom was rescored, and *prev* and *next* are the linked-list "pointers," or rather indices in *As* (with None representing a null pointer).
- We extend the data structure with a *next_search* component that stores an atom. If $As = As_0 \cdot \ldots \cdot As_{|As|-1}$, with $next\_search = As_j$, all atoms $As_0, \ldots, As_{j-1}$ are set in the trail. When searching for an undefined atom, we can start at index *j*.
- Timestamps enable us to efficiently unset a literal (e.g., when jumping). Since atoms are sorted in reverse timestamp order in *As*, we need to update *next_search* only if the unset atom has a higher timestamp than the current *next_search* atom.
- We batch the rescoring of atoms. Atoms are not removed from *As*, until the end of the next Jump when rescoring takes place: We sort the atoms to rescore by their timestamps and prepend them to *As*.

The VMTF data structure is captured as a tuple $vmtf = ((ns, st, fst, next\_search), to\_rescore)$. The *ns* component corresponds the doubly linked list described above; *st* is the maximum timestamp; *fst* gives the first atom in *As*; and *to_rescore* is the batch of atoms that are awaiting rescoring.

In Isabelle, we define the inductive predicate vmtf *As st ns* that checks whether *ns* stores a doubly linked list corresponding to *As* and the timestamps are bounded by *st*. It is defined by the following introduction rules:

Empty list vmtf ϵ *st ns*, where ϵ denotes the empty list;

Singleton list vmtf *i st ns*
if $i < |ns|$ and $ns\,!\,i = (st, \text{None}, \text{None})$;

List of length 2 or more vmtf $(i \cdot j \cdot As)$ $(st + 1)$ *ns*
if vmtf $(j \cdot As)$ *st ns'*, $i \neq j$, $i < |ns|$, and *ns* is *ns'* where $ns\,!\,i = (st+1, \text{None}, \text{Some } j)$ and the *prev* component of $ns'\,!\,j$ has been updated to Some *i*.

In the function that finds the next unset literal, we iterate over the doubly linked list stored in *ns*:

```
find_next_undef ((ns, st, fst, next_search), _) M = do {
  WHILE_T (λnext_search. next_search ≠ None
      ∧ defined_atm M (the next_search))
    (λnext_search.
      RETURN (get_next (A ! the next_search)))
    next_search
}
```

The defined_atm predicate tests whether an atom is set in the trail. The get_next *i* function returns the *next* component of the node associated with atom *i*—i.e., the atom following atom *i* in *As* (or None if *i* is the last element in *As*).

To prove this program correct, we must show the termination of the while loop, which amounts to the well-foundedness of the relation

$$\{(\text{get\_next } (\text{ns} \,!\, \text{the } next\_search), next\_search) \mid next\_search \neq \text{None}\}$$

This, in turn, amounts to showing that the chain of get_next calls contains no loops. We achieve this by showing that the chain is a traversal of the list *As*, which is finite.

When implementing a heuristic such as VMTF, we must prove that it does not fail (e.g., because of an out-of-bound array access) and that it returns a correct result. We do not need to prove that our implementation is actually a "VMTF" as defined by Biere and Fröhlich [5]. For example, there are no formal guarantees that the sorting function we use to rescore the batched atoms by their timestamps is correct; it is sufficient to show that sorting introduces no new atoms.

VMTF gives only the next atom to decide (if one exists). We also need to choose the literal's polarity. We use the *phase saving* heuristic [31]. It is a mapping φ from an atom to a polarity, implemented as an array of Booleans. Initially, all atoms are mapped to a negative polarity. Then for each conflict, the mapping is updated: Every atom involved in the conflict will be mapped to the polarity it has in the trail.

8.2 Conflict Clause as a Lookup Table

In the TWL calculus and the refinements shown so far, the conflict clause is either $\top$ (None) or an actual clause (Some *C*). Four operations access or modify the conflict clause:

- The Conflict rule replaces $\top$ by a conflict clause.
- The Resolve rule merges the conflict clause with another clause, removing duplicates.
- The choice between the Resolve and Skip rules depends on whether the trail's head appears in the conflict clause.
- The choice between Resolve and Jump requires an iteration through the clause to evaluate the maximum level of the clause minus one literal.

Initially, we tried representing the conflict as an optional resizable array that is converted to a nonresizable array when the clause is learned (rule Jump_Nonunit). However, this led to many memory allocations and to inefficient code for resolution (rule Resolve).

Inspired by MiniSat, we moved to an encoding of the conflict clause as a lookup table. We use an array *ps* such that the entry at position *i* indicates the polarity of atom *i* in the conflict clause—i.e, whether the literal *i* occurs positively, negatively, or not at all in the clause. More precisely, a conflict clause is represented by a triple (b, n, ps), where *b* indicates whether the conflict is $\top$ and *n* stores the size of the conflict clause. The *n* component is useful to quickly test whether the conflict clause is empty, or whether it has size one.

There are two main differences between the lookup table and the original version. First, duplicate literals and tautologies cannot be represented. We know from our invariants that this is not an issue. Second, the clause can only contain atoms that are smaller than the length of the array.

To give a sense of what this involves, we describe the refinement of a small program fragment from the abstract level, where a conflict is an optional multiset, to the concrete level, where a conflict is a lookup table. At the end of Jump_Nonunit, we need to convert the conflict clause *C* to a list that we can add to our list of clauses such that two given literals $L, L' \in C$ are watched (i.e., are at positions 0 and 1). This conversion is specified abstractly as

$$\text{RES } \{(D, \text{None}) \mid D\,!\,0 = L \wedge D\,!\,1 = L' \wedge \text{mset } D = C \wedge |D| \geq 2\}$$

The condition $|D| \geq 2$ ensures that the accesses to positions 0 and 1 are well-defined. In the refined code, we convert the lookup table to an array (*D* in the specification) and empty the lookup table (instead of reallocating a new one later; this is the None in the specification).

The refinement is done in two steps. We first refine the specification to an intermediate function that describes the implementation on the level of the abstract data structures (leftmost column of Figure 2). In a second step, the abstract data structures and operations are refined to concrete data

Intermediate code	Refinement relation	Imperative HOL code
	(b', n', ps'_C) refines the clause C as a lookup table; L' and K' refine the literals L and K	
let n = size (the C);		$n' \leftarrow$ size_conflict_code (b', n', ps'_C)
	The 32-bit unsigned integer n' is equal to the natural number n	
let D = replicate n K;		$D' \leftarrow$ Array.new n' K'
	The array D' has the same length and same content as the list D	
let $D = D[1 := L]$;		$D' \leftarrow$ Array.upd 1 D' L'
	The array D' refines the updated list D; both contain K at position 1	
let C' = Some (the $C - \{K, L\}$);		$(b', n', ps'_{C'}) \leftarrow$ remove_from K' L' (b', n', ps'_C)
	$(b', n' - 2, ps'_C)$ refines the clause C'	
RES $\{(E,$ None$) \mid E\,!\,0 = K$ $\wedge\ E\,!\,1 = L \wedge |E| \geq 2$ $\wedge$ mset (drop 2 E) $= C'\}$		$(E', (b', n', ps'_\top)) \leftarrow$ $(\lambda\_\ \_.$ conflict_with_cls) K' L' D' $(b', n', ps'_{C'})$
	The array E' refines the clause C, and $(b', n', ps'_\top)$ refines $\top$	

Figure 2. Conversion from the lookup table to a clause, assuming $C \neq$ None

structures and operations (rightmost column of Figure 2). The middle column gives the refinement relation that connects the notions of states used on either side, before and after every statement. Each statement from the intermediate code is mapped to a concrete function, such that the refinement relation of the result is also the refinement relation of the arguments of the next statement. Since intermediate and concrete functions must have the same number of arguments, some arguments are ignored on the concrete side (indicated by _ in the λ-abstractions).

8.3 Conflict Clause Minimization

Conflict clause minimization consists of removing some literals from the conflict clause while ensuring that the clause is still entailed by the other clauses. In the Jump rule, this corresponds to the conditions $D' \subseteq D$ and $N \uplus U \models D' \vee L$ on D'. Shorter conflict clauses need less memory and can allow propagations to take place earlier.

We follow a minimization scheme due to Sörensson and Biere [34]. If the conflict is $E \vee K$, where E contains the literal L that is always kept in rule Jump, and we can show that $N \uplus U \models E \vee -K$, then by resolution we have $N \uplus U \models E$ and the conflict can be reduced to E. More precisely, minimization is a recursive procedure that considers each literal K of the conflict distinct from L in turn:

1. If K appears in E, then $E \vee K$ can be reduced to E.
2. If $-K$ is set at level 0 in the trail, then $-K$ is entailed by $N \uplus U$ and $E \vee K$ can be reduced to E.
3. If $(-K)^{-K \vee C}$ appears in the trail and for each literal K' of C, we have that $E \vee K'$ be recursively reduced to E, then $E \vee K$ can be reduced to E.
4. Otherwise (e.g., if K was decided), the literal K is kept.

The minimization procedures terminates because the literals K' have been set earlier than K. To optimize the procedure, we cache the clause's minimization status: "can be minimized", "cannot be minimized", or "not determined yet." This turns out to be the trickiest part of the proof. After exploring many dead ends, we found that we can define "can be minimized" as $N \uplus U \models E^{\succ_M K} \vee -K$, where $E^{\succ_M K}$ denotes the subclause of E consisting only of literals that appear to the right of K in the trail M.

Minimization is specified abstractly in terms of multisets and refined to an efficient implementation using the lookup-table representation. To simplify the code, when propagating a literal we ensure it appears at the first position in the clause, as in MiniSat. Similarly to VMTF, we prove correctness but no notion of optimality.

8.4 State Representation

The states we are considering before generating code in Imperative HOL are eight-tuples $(M, NU, u, D, NP, UP, Q, W)$. However, two components are redundant and can be eliminated: Unit clauses are added to NP and UP but never accessed afterwards.

Initially, we wrote code as we have shown in Section 6: All function bodies started with let $(M, NU, u, D, NP, UP, WS, Q) = S$. This made it convenient to refer to the components individually, or to refine them. We could also add information to the components during refinement. For example, since the VMTF heuristic depends on the trail, its *vmtf* tuple could be added to the refined trail component.

However, this approach works only if the additional information depends on a single component. Moreover, it offers no means of eliminating redundant components such as *NP* and *UP*.

After gathering some experience with the Refinement Framework, we decided to move to a different scheme, following which all state manipulation is mediated by accessor functions. We can then refine each of these functions individually. For example, when refining $(M, NU, u, D, NP, UP, WS, Q)$ to the intermediate representation $(M, NU, u, D, WS, Q, vmtf, \varphi)$ with heuristics (where *vmtf* is the VMTF data structure and φ is the mapping used for phase saving), the get_queue function that selects the eighth tuple component is mapped to a function that selects the sixth tuple component.

There is, however, a difficulty with this scheme. In an imperative implementation, a getter that returns a component of a state that is stored on the heap must either copy the component or return a pointer into the state. The first option can be very inefficient, and the alternative is not supported by the Sepref tool, which does not permit pointer aliases. Our solution is to provide ad hoc getters to extract the relevant information from the state, without exposing parts of the state simultaneously to the whole state (which would require aliasing). Similarly, we provide setter functions to update components of the state.

For example, after reducing a conflict (rules Resolve and Skip), we must distinguish between either jumping (rules Jump_Unit and Jump_Nonunit) or stopping the solver by testing whether the conflict was reduced to $\bot$:

$$\text{the } (\text{get\_conflict}_{\text{wlist}}\ S) = \emptyset$$

(The result is unspecified if the conflict is $\top$, i.e., None.)

Since all we need is the emptiness check and not the conflict clause itself, we can define a specialized getter:

$$\text{conflict\_is\_empty}_{\text{wlist}}\ S \longleftrightarrow \text{the } (\text{get\_conflict}_{\text{wlist}}\ S) = \emptyset$$

Then we refine it to the intermediate state with heuristics:

$$\text{conflict\_is\_empty}_{\text{heuristic}}\ (M, NU, u, D, WS, Q, vmtf, \varphi) \longleftrightarrow \text{conflict\_is\_empty}\ D$$

with the following auxiliary function that operates only on the D component:

$$\text{conflict\_is\_empty}\ D \longleftrightarrow \text{the } D = \emptyset$$

Next, we refine the auxiliary function to use the lookup-table representation:

$$\text{conflict\_is\_empty}_{\text{lookup}}\ (b, n, ps) \longleftrightarrow n = 0$$

Finally, this function is given to Sepref, which generates Imperative HOL code.

The representation of states changes between refinement layers. It can also change within a layer, to store temporary information. Consider the number of literals of maximum level in the conflict clause. When it reaches 1, the Resolve rule no longer applies. Keeping this number around, in a locally enriched state tuple, can be much more efficient than iterating over the conflict clause to evaluate the maximum level. With our initial concrete notion of state as an eight-tuple, adding this information would have required a new layer of refinement, since the level depends simultaneously on two state components (the trail and the conflict clause).

9 Evaluation

We compare the performance of our solver, IsaSAT, with Glucose 4.1 [1], MiniSat 2.2 [13], DPT 2.0, and `versat` [30].

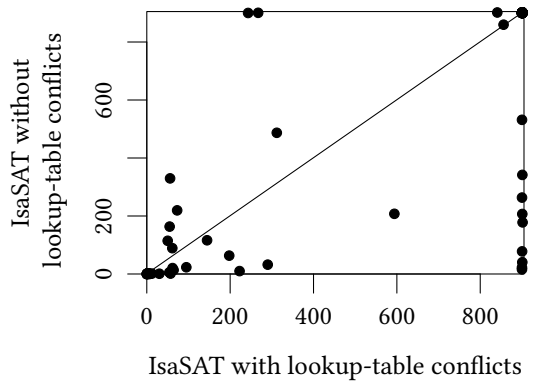
`versat`, by Oe et al. [30], is specified and verified using the Guru proof assistant [35], which can generate C code. `versat` consists of 15 000 lines of C code. Optimized data structures are used, including for watched literals and conflict analysis (but not for conflict minimization), and the VSIDS heuristic is in charge of decisions. However, termination is not guaranteed, and model soundness is proved trivially by means of a run-time check of the models; if this check fails, the solver's outcome is "unknown."

We ran all five solvers on the 150 problems classified easy or medium from the SAT Competition 2009, with a time limit of 900 s. Glucose solves 147 problems, spending 51 s on average per problem it solves. MiniSat solves 143 problems in 98 s. DPT solves 70 problems in 206 s on average. `versat` solves 53 problems in 235 s on average.

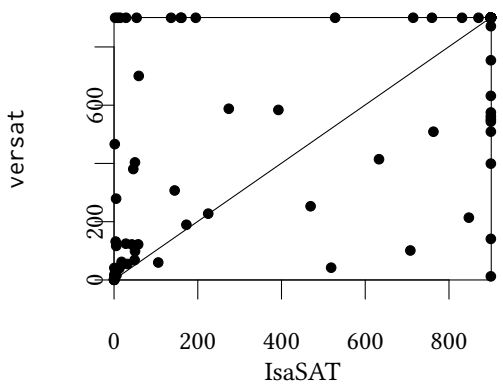
To evaluate the lookup-table-conflict representation, we ran IsaSAT without caching of the number of literals of maximum level. IsaSAT without the lookup table solves 43 problems in 126 s on average, while the version with lookup table solves only 36 problems in 127 s on average (including four problems that the version without lookup table could not solve). IsaSAT with every optimization solves 56 problems in 183 s on average. As an indication of how far we have come, the functional solver implementing the W calculus [7, Section 5] and our first imperative unoptimized version with watched literals do not solve *any* of the problems. The solvers were run on a Xeon E5-2680 with 256 GB of memory, with Intel Turbo Boost deactivated. Globally, the experiments show that Glucose and MiniSat are much faster than the other solvers and that DPT solves substantially more instances than IsaSAT and `versat`, which are roughly comparable.

A more precise comparison of performance of our solver with and without the lookup table is shown in Figure 3a. A point at coordinates (x, y) indicates that the version with the lookup table took x seconds, whereas the version without the table took y seconds. Points located above the main diagonal correspond to problems for which the table pays off. Figure 3b compares `versat` and the optimized IsaSAT: It shows that either solver solves some problems on which the other solver times out. This is to be expected given that the two solvers implement different decision heuristics.

There are several reasons explaining why our solver is much slower than the state of the art. First, it lacks restarts



(a) IsaSAT with and without the lookup-table conflict representation (Section 8.2), without caching the number of literals of highest level in the conflict



(b) IsaSAT and `versat`

Figure 3. Comparison of performance on the problems classified easy or medium from the SAT Competition 2009

and forgetting. Restarts enable the solver to explore another part of the search space, whereas forgetting removes some learned clauses. Glucose and MiniSat also use preprocessing techniques to simplify the initial set of clauses. Other SAT solvers, such as Lingeling [4], also use inprocessing techniques to simplify initial and learned clauses after restarts.

Another difference is that Isabelle/HOL can only generate code in impure functional languages, whereas most unverified SAT solvers are developed in C or C++. Although we proved that all array accesses are within bounds, functional languages nonetheless check array bounds at run-time. Moreover, other features, such as the arbitrary precision arithmetic (which we use for clause indices), tend to be less efficient than their C++ counterparts.

To reduce these effects, we implemented literals by 32-bit unsigned integers (which required some extra work to prove absence of overflows). This increased the speed of our solver by a factor between two and four. In a slight extension of the trusted base of the code generation, we convert literals directly to machine-size integers (32- or 64-bit), instead of taking the detour through unbounded integers. This simple change improved performance by another factor of two.

10 Discussion and Related Work

We found formalizing the two watched literals challenging. In the literature, only variants of the invariant from Section 3 are presented. However, there are several other key properties that are necessary to prove that no work is needed when backjumping. For example, the invariant states that "a watched literal may be false only if the other watched literal is true," but this is not the whole story. It would be more precise to state that "a watched literal may be false only if the other watched literal is true *and this false literal's level is greater than or equal to the true literal's level*." This version of the invariant explains why no update is required after Jump: Either both watched literals are now unset in the trail, or only the true literal remains.

One difficulty we faced when adding optimizations is that the "edit, compile, run" cycle is much longer when code is developed through the Isabelle Refinement Framework instead of directly in a programming language such as C++. For example, the change to the conflict-clause representation took two weeks to prove and implement, before we found out that the overall solver gets slower. We have yet to find a good methodology for carrying out quick experiments.

The distinguishing feature of our work is the systematic application of refinement to connect abstract calculi with generated code. The Refinement Framework allows us to generate imperative code while keeping programs underspecified for as long as possible. It makes it straightforward to change the implementation or to derive multiple implementations from the same abstract specification. Its support for assertions makes it possible to reuse properties proved on an abstract level to reason about more concrete levels.

The Refinement Framework's lack of support for pointer aliasing impacted our solver in two main ways. First, we had to use array indices instead of pointers to clauses. This moved the dependency between the array and the clause from the code level to the abstract specification level. Second, array access $NU\,!\,C$ must take a copy of the array at position C. We avoided this issue by consistently using two-dimensional indexing, $(NU\,!\,C)\,!\,i$, which yields an unsigned 32-bit integer representing a literal.

The formalization work described in this paper took about 1.5 person-years. The longest part was the refinement from the abstract algorithm to the first executable version. To improve performance, we studied the generated code and looked for bottlenecks. This was tedious: The code is hardly readable, with generated variable names. But at least, at every step we knew that the code was correct.

Given that we had formalized CDCL in Isabelle/HOL, it was natural to use the Isabelle Refinement Framework and Sepref. For Coq, the Fiat tool is available [12]. Like Sepref, it applies automatic data refinement to obtain efficient implementations. However, it is limited to purely functional implementations and does not support recursive programs. Nor does it support assertions, which are an important mechanism to move facts down the refinement chain instead of reproving them at each level.

Gries and Volpano [15] describe a data refinement approach that, like Sepref, automatically transforms abstract to concrete data structures, by replacing abstract with concrete operations. It refines imperative to imperative programs, whereas Sepref connects functional to imperative programs. To our knowledge, their approach has not been implemented in a theorem prover.

The closest formalizations to ours are `versat` by Oe et al. [30] and Marić's [25, 26]. Marić verified a CDCL-based SAT solver in Isabelle/HOL, including watched literals, as a purely functional program. His methodology is quite different from ours, as it does not include refinement. While he was able to generate code with an earlier version of Isabelle, the code export does not work anymore. Since he uses lists instead of arrays, we expect the performance to be substantially worse than ours. Beyond Oe et al. and Marić, there are several formalizations of CDCL that do not include watched literals, including Lescuyer's in Coq [24] and Shankar and Vaucher's in PVS [33], and also some formalizations of DPLL, including Berger et al. [3], whose Haskell solver outperforms `versat` on large pigeon-hole problems. (CDCL is not faster than DPLL on such problems, because the learned clauses are useless at pruning the search space.)

Instead of verifying a SAT solver, another way to obtain trustworthy results is to have the solver produce a certificate, which can be processed by a checker. While certificates for satisfiable formulas are simply a valuation of the variables and can easily be generated and checked, certificates for unsatisfiable formulas are more complicated. The de facto standard format is DRAT (deletion resolution asymmetric tautology) [17]. The standard DRAT certificate checker [38] is, however, an unverified C program. Recent research [9, 10, 18], including by Lammich [22], shows that it is now possible to have efficient verified checkers.

11 Conclusion

We have extended our Isabelle/HOL framework for CDCL with a verified imperative SAT solver. Refinement-based development is flexible and makes later changes in the development easier. We expect that many heuristics and optimizations can be added with only comparatively small modular changes to the existing proofs. For example, an initial version of our SAT solver did not include the lookup-table representation of conflicts (Section 8); adding it required changes only to the last refinement step.

The refinement steps are an interesting case study of the Refinement Framework, Imperative HOL, and the code generator. In future work, we plan to add (fast) restarts and forget and to extend the CeTA checker to use our verified SAT solver. We would also like to formalize CDCL(T), the metatheory behind satisfiability-modulo-theories (SMT) solvers, and other extensions such as MaxSAT.

Acknowledgments

Max Haslbeck, Anders Schlichtkrull, Mark Summerfield, and the anonymous reviewers suggested many textual improvements. Blanchette has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka).

References

[1] Gilles Audemard and Laurent Simon. 2009. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI 2009*, Craig Boutilier (Ed.). ijcai.org, 399–404.

[2] Leo Bachmair and Harald Ganzinger. 2001. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Vol. I. Elsevier, 19–99.

[3] Ulrich Berger, Andrew Lawrence, Fredrik Nordvall Forsberg, and Monika Seisenberger. 2015. Extracting Verified Decision Procedures: DPLL and Resolution. *Logical Methods in Computer Science* 11, 1 (2015).

[4] Armin Biere. 2016. Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In *SAT Competition 2016—Solver and Benchmark Descriptions (Department of Computer Science Series of Publications B)*, Tomáš Balyo, Marijn Heule, and Matti Järvisalo (Eds.), Vol. B-2016-1. University of Helsinki, 44–45.

[5] Armin Biere and Andreas Fröhlich. 2015. Evaluating CDCL Variable Scoring Schemes. In *SAT 2015 (LNCS)*, Marijn Heule and Sean Weaver (Eds.), Vol. 9340. Springer, 405–422.

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.

[7] Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. 2016. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. In *IJCAR 2016 (LNCS)*, Nicola Olivetti and Ashish Tiwari (Eds.), Vol. 9706. Springer, 25–44.

[8] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. 2008. Imperative Functional Programming with Isabelle/HOL. In *TPHOLs 2008 (LNCS)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.), Vol. 5170. Springer, 134–149.

[9] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. 2017. Efficient Certified RAT Verification. In *CADE-26 (LNCS)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, 220–236.

[10] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. 2017. Efficient Certified Resolution Proof Checking. In *TACAS 2017 (LNCS)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10205. Springer, 118–135.

[11] Martin Davis, George Logemann, and Donald W. Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5, 7 (1962), 394–397.

[12] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a

Proof Assistant. In *POPL 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 689–700.

[13] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-Solver. In *SAT 2003 (LNCS)*, Enrico Giunchiglia and Armando Tacchella (Eds.), Vol. 2919. Springer, 502–518.

[14] Mathias Fleury and Jasmin Christian Blanchette. 2017. 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.

[15] David Gries and Dennis M. Volpano. 1990. The Transform—A New Language Construct. *Structured Programming* 11, 1 (1990), 1–10.

[16] Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *FLOPS 2010 (LNCS)*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.), Vol. 6009. Springer, 103–117.

[17] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. 2014. Bridging the Gap between Easy Generation and Efficient Verification of Unsatisfiability Proofs. *Softw. Test. Verif. Reliab.* 24, 8 (2014), 593–607.

[18] Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. 2017. Efficient, Verified Checking of Propositional Proofs. In *ITP 2017 (LNCS)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.), Vol. 10499. Springer, 269–284.

[19] Peter Lammich. 2013. Automatic Data Refinement. In *ITP 2013 (LNCS)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.), Vol. 7998. Springer, 84–99.

[20] Peter Lammich. 2015. Refinement to Imperative/HOL. In *ITP 2015 (LNCS)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 253–269.

[21] Peter Lammich. 2016. Refinement Based Verification of Imperative Data Structures. In *CPP 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 27–36.

[22] Peter Lammich. 2017. The GRAT Tool Chain—Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees. In *SAT 2017 (LNCS)*, Serge Gaspers and Toby Walsh (Eds.), Vol. 10491. Springer, 457–463.

[23] Peter Lammich and Thomas Tuerk. 2012. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *ITP 2012 (LNCS)*, Lennart Beringer and Amy P. Felty (Eds.), Vol. 7406. Springer, 166–182.

[24] Stephane Lescuyer. 2011. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq.* Ph.D. Dissertation. Université Paris-Sud.

[25] Filip Marić. 2008. Formal Verification of Modern SAT Solvers. *Archive of Formal Proofs* (2008). http://isa-afp.org/entries/SATSolverVerification.shtml, Formal proof development.

[26] Filip Marić. 2010. Formal Verification of a Modern SAT Solver by Shallow Embedding into Isabelle/HOL. *Theor. Comput. Sci.* 411, 50 (2010), 4333–4356.

[27] Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. 2016. Eisbach: A Proof Method Language for Isabelle. *J. Autom. Reasoning* 56, 3 (2016), 261–282.

[28] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *DAC 2001*. ACM, 530–535.

[29] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM* 53, 6 (2006), 937–977.

[30] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. 2012. `versat`: A Verified Modern SAT Solver. In *VMCAI 2012*, Viktor Kuncak and Andrey Rybalchenko (Eds.). LNCS, Vol. 7148. Springer, 363–378.

[31] Knot Pipatsrisawat and Adnan Darwiche. 2007. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *SAT 2007 (LNCS)*, João Marques-Silva and Karem A. Sakallah (Eds.), Vol. 4501. Springer, 294–299.

[32] Lawrence Ryan. 2004. *Efficient Algorithms for Clause-Learning SAT Solvers.* Ph.D. Dissertation. Simon Fraser University.

[33] Natarajan Shankar and Marc Vaucher. 2011. The Mechanical Verification of a DPLL-Based Satisfiability Solver. *Electr. Notes Theor. Comput. Sci.* 269 (2011), 3–17.

[34] Niklas Sörensson and Armin Biere. 2009. Minimizing Learned Clauses. In *SAT 2009 (LNCS)*, Oliver Kullmann (Ed.), Vol. 9340. Springer, 237–243.

[35] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy W. Simpson. 2009. Verified Programming in Guru. In *PLPV 2009*, Thorsten Altenkirch and Todd D. Millstein (Eds.). ACM, 49–58.

[36] René Thiemann and Christian Sternagel. 2009. Certification of Termination Proofs Using CeTA. In *TPHOLs 2009 (LNCS)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 452–468.

[37] Christoph Weidenbach. 2015. Automated Reasoning Building Blocks. In *Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday (LNCS)*, Roland Meyer, André Platzer, and Heike Wehrheim (Eds.), Vol. 9360. Springer, 172–188.

[38] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *SAT 2014 (LNCS)*, Carsten Sinz and Uwe Egly (Eds.), Vol. 8561. Springer, 422–429.