Efficient Verified (UN)SAT Certificate Checking

Peter Lammich

Technische Universität München, lammich@in.tum.de

Abstract. We present an efficient formally verified checker for satisfiability and unsatisfiability certificates for Boolean formulas in conjunctive normal form. It utilizes a two phase approach: Starting from a DRAT certificate, the unverified generator computes an enriched certificate, which is checked against the original formula by the verified checker.

Using the Isabelle/HOL Refinement Framework, we verify the actual implementation of the checker, specifying the semantics of the formula down to the integer sequence that represents it.

On a realistic benchmark suite drawn from the 2016 SAT competition, our approach is more than two times faster than the unverified standard tool drat-trim. Additionally, we implemented a multi-threaded version of the generator, which further reduces the runtime.

1 Introduction

Modern SAT solvers are highly optimized and use complex algorithms and heuristics. This makes them prone to bugs. Given that SAT solvers are used in software and hardware verification, a single bug in a SAT solver may invalidate the verification of many systems.

One measure to increase the trust in SAT solvers is to make them output a certificate, which is used to check the result of the solver by a simpler algorithm. Most SAT solvers support the output of a satisfying valuation of the variables as an easily checkable certificate for satisfiability. Certificates for unsatisfiability are more complicated, and different formats have been proposed (e. g. [39, 41, 42]). Since 2013, the SAT competition [35] requires solvers to output unsat certificates. Since 2014, only certificates in the DRAT format [42] are accepted [36].

The standard tool to check DRAT certificates is drat-trim [10, 42]. It is a highly optimized C program with many features, including forward and backward checking mode, a satisfiability certificate checking mode, and a feature to output reduced (trimmed) certificates. However, the high degree of optimization and the wealth of features come at the price of code complexity, increasing the likelihood of bugs. And indeed, during our formalization of the RAT property, we realized that drat-trim was missing a crucial check, thus accepting (maliciously engineered) unsat certificates for satisfiable formulas. This bug has been confirmed by the authors, and is now fixed. Moreover, we discovered several numeric and buffer overflow issues in the parser [11], which could lead to misinterpretation of the formula. Thus, although being less complex than SAT solvers, efficient DRAT checkers are still complex enough to easily overlook bugs.[1]

One method to eliminate bugs from software is to conduct a machine-checked correctness proof. A common approach is to prove correct a specification in the logic of an interactive theorem prover, and then generate executable code from the specification. Here, code generation is merely a syntax transformation from the executable fragment of the theorem prover's logic to the target language. Following the LCF approach [14], modern theorem provers like Isabelle [34] and Coq [3] are explicitly designed to maximize their trustworthiness. Unfortunately, the algorithms and low-level optimizations required for *efficient* unsat certificate checking are hard to verify and existing approaches (e. g. [8, 41]) do not scale to large problems.

While working on the verification of an efficient DRAT checker, the author learned about GRIT, proposed by Cruz-Filipe et al. [7]: They use a modified version of drat-trim to generate an enriched certificate from the original DRAT certificate. The crucial idea is to record the required unit propagations, such that the checker of the enriched certificate only needs to implement a check whether a clause is unit, instead of a fully fledged unit propagation algorithm.

Cruz-Filipe et al. formalize a checker for their enriched certificates in the Coq theorem prover [3], and generate OCaml code from the formalization. However, their current approach still has some deficits: GRIT only supports the less powerful DRUP fragment [41] of DRAT, making it unsuitable for recent SAT solvers that output full DRAT (e. g. CryptoMiniSat, Riss6 [37]). Also, their checker does not consider the original formula, but assumes that the certificate correctly mirrors the formula. Moreover, they use unverified code to parse the certificate into the internal data structures of the checker. Finally, their verified checker is quite slow: Checking a certificate requires roughly the same time as generating it, which effectively doubles the verification time. However, an unverified implementation of their checker in C is two orders of magnitude faster.

In this paper, we present enriched certificates for full DRAT, along with a checker whose correctness is formally verified down to the integer sequence representing the formula. The simple unverified parser that reads a formula into an integer array is written in Standard ML [30], which guarantees that numeric and buffer overflows will not go unnoticed.

We use stepwise refinement techniques to obtain an efficient verified checker, and implement aggressive optimizations in the generator. As a result, our tool chain (generation plus checking) is more than two times faster than drat-trim, with the additional benefit of providing strong formal correctness guarantees. Another distinguishing is a multi-threaded mode for the generator, which allows us to trade hardware resources for additional speedup: With 8 threads, our tool chain verifies a DRAT-certificate seven times (on average) faster than drat-trim.

[1] Unfortunately, the available version history of drat-trim [9] only dates back to October 2016. We can only speculate whether the discovered bugs were present in the versions used for the 2014 and 2016 SAT competitions.

Building on the technology of our verified unsat certificate checker, we also provide a verified sat certificate checker, obtaining a complete, formally verified, and fast SAT solver certification tool chain. Our tools, formalizations, and benchmark results are available online [20].

Independently to us, Cruz-Filipe et al. also extended their work to DRAT [6]. Their certificate generator is still based on drat-trim, and first benchmarks indicate that our approach might be significantly faster.[2]

The rest of this paper is organized as follows: After briefly recalling the theory of DRAT certificates (§2), we introduce our enriched certificate format (§3). We then give a short overview of the Isabelle Refinement Framework (§4) and describe its application to verifying our certificate checker (§5). The paper ends with a brief description of our certificate generator (§6) and a report on the experimental evaluation of our tools (§7).

2 Unsatisfiability Certificates

We briefly recall the theory of DRAT unsatisfiability certificates. Let V be a set of variable names. The set of *literals* is defined as $L := V \dot{\cup} \{\neg v \mid v \in V\}$. We identify v and $\neg\neg v$. Let $F = C_1 \wedge \ldots \wedge C_n$ for $C_i \in 2^L$ be a formula in conjunctive normal form (CNF). F is *satisfied* by an *assignment* $A : V \Rightarrow \text{bool}$ iff instantiating the variables in F with A yields a true (ground) formula. We call F *satisfiable* iff there exists an assignment that satisfies F.

A clause C is called a *tautology* iff there is a variable v with $\{v, \neg v\} \subseteq C$. Removing a tautology from a formula yields an equivalent formula. In the following we assume that formulas do not contain tautologies. The empty clause is called a *conflict*. A formula that contains a conflict is unsatisfiable. A singleton clause $\{l\} \in F$ is called a *unit clause*. Removing all clauses that contain l, and all literals $\neg l$ from F yields an equisatisfiable formula. Repeating this exhaustively for all unit clauses is called *unit propagation*. When identifying formulas that contain a conflict, unit propagation is strongly normalizing. We name the result of unit propagation F^{u}, defining $F^{\mathrm{u}} = \{\emptyset\}$ if unit propagation yields a conflict.

A DRAT certificate $\chi = \chi_1 \ldots \chi_n$ with $\chi_i \in 2^L \dot{\cup} \{\mathrm{d}C \mid C \in 2^L\}$ is a list of clause addition and deletion items. The *effect* of a (prefix of) a DRAT certificate is to add/delete the specified clauses to/from the original formula F_0, and apply unit propagation:

$$\mathrm{eff}(\varepsilon) = (F_0)^{\mathrm{u}} \qquad \mathrm{eff}(\chi C) = (\mathrm{eff}(\chi) \wedge C)^{\mathrm{u}} \qquad \mathrm{eff}(\chi \mathrm{d}C) = \mathrm{eff}(\chi) \setminus C$$

where $F \setminus C$ removes one occurrence of clause C from F. We call the clause addition items of a DRAT certificate *lemmas*.

A DRAT certificate $\chi = \chi_1 \ldots \chi_n$ is *valid* iff $\mathrm{eff}(\chi) = \{\emptyset\}$ and each lemma has the RAT property wrt. the effect of the previous items:

$$\mathrm{valid}(\chi_1 \ldots \chi_n) := \forall 1 \leq i \leq n.\ \chi_i \in 2^L \Longrightarrow \mathrm{RAT}(\mathrm{eff}(\chi_1 \ldots \chi_{i-1}), \chi_i)$$

[2] However, we expect that most of our optimizations can be transferred to their tools.

where a clause C has the *RAT* (*resolution asymmetric tautology*) property wrt. formula F (we write $\mathrm{RAT}(F, C)$) iff either C is empty and $F^{\mathrm{u}} = \{\emptyset\}$, or if there is a *pivot literal* $l \in C$, such that for all *RAT candidates* $D \in F$ with $\neg l \in D$, we have $(F \wedge \neg(C \cup D \setminus \{\neg l\}))^{\mathrm{u}} = \{\emptyset\}$. Adding a lemma with the RAT property to a formula preserves satisfiability, and so do unit propagation and deletion of clauses. Thus, existence of a valid DRAT certificate implies unsatisfiability of the original formula.

A strictly weaker property than RAT is *RUP* (*reverse unit propagation*): A lemma C has the RUP property wrt. formula F iff $(F \wedge \neg C)^{\mathrm{u}} = \{\emptyset\}$. Adding a lemma with the RUP property yields an equivalent formula. The predecessor of DRAT is DRUP [18], which admits only lemmas with the RUP property.

Checking a lemma for RAT is much more expensive than checking for RUP, as the clause database must be searched for candidate clauses, performing a unit propagation for each of them. Thus, practical DRAT certificate checkers first perform a RUP check on a lemma, and only if this fails they resort to a full RAT check. Exploiting that $(F \wedge \neg(C \cup D))^{\mathrm{u}}$ is equivalent to $((F \wedge \neg C)^{\mathrm{u}} \wedge \neg D)^{\mathrm{u}}$, the result of the initial unit propagation from the RUP check can even be reused. Another important optimization is *backward checking* [13, 18]: The lemmas are processed in reverse order, marking the lemmas that are actually needed in unit propagations during RUP and RAT checks. Lemmas that remain unmarked need not be processed at all. To further reduce the number of marked lemmas, *core-first* unit propagation [42] prefers marked unit clauses over unmarked ones.

In practice, DRAT certificate checkers spend most time on unit propagation[3], for which highly optimized implementations of rather complex algorithms are used (e. g. drat-trim uses a two watched literals algorithm [32]). Unfortunately, verifying such highly optimized code in a proof assistant is a major endeavor. Thus, a crucial idea is to implement an unverified tool that enriches the certificate with additional information that can be used for simpler and more efficient verification. For DRUP, the GRIT format has been proposed recently [7]. It stores, for each lemma, a list of unit clauses in the order they become unit, followed by a conflict clause. Thus, unit propagation is replaced by simply checking whether a clause is unit or conflict. A modified version of drat-trim is used to generate a GRIT certificate from the original DRAT certificate.

3 The GRAT Format

The first contribution of this paper is to extend the ideas of GRIT from DRUP to DRAT. To this end, we define the GRAT format. Like for GRIT, each clause is identified by a unique positive ID. The clauses of the original formula implicitly get the IDs $1 \ldots N$. The lemma IDs explicitly occur in the certificate, and must be strictly ascending.

For memory efficiency reasons, we store the certificate in two parts: The lemma file contains the lemmas, and is stored in DIMACS format. During certificate

[3] Our profiling data indicates that, depending on the problem, up to 93% of the time is spent for unit propagation.

checking, this part is entirely loaded into memory. The proof file contains the hints and instructions for the certificate checker. It is not completely loaded into memory, but only streamed during checking.

The proof file is a binary file, containing a sequence of 32 bit signed integers stored in 2's complement little endian format. The sequence is reversed (or the file is streamed backwards), and then interpreted according to the following grammar:

```
proof      ::= rat-counts item* conflict
literal    ::= int32 != 0
id         ::= int32 > 0
count      ::= int32 > 0
rat-counts ::= 6 (literal count)* 0
item       ::= unit-prop | deletion | rup-lemma | rat-lemma
unit-prop  ::= 1 id* 0
deletion   ::= 2 id* 0
rup-lemma  ::= 3 id id* 0 id
rat-lemma  ::= 4 literal id id* 0 cand-prf* 0
cand-prf   ::= id id* 0 id
conflict   ::= 5 id
```

The checker maintains a *clause map* that maps IDs to clauses, and a *partial assignment* that maps variables to true, false, or undecided. Partial assignments are extended to literals in the natural way. Initially, the clause map contains the clauses of the original formula, and the partial assignment maps all variables to undecided. Then, the checker iterates over the items of the proof, processing each item as follows:

rat-counts This item contains a list of pairs of literals and the number how often they are used in RAT proofs. This map allows the checker to maintain lists of RAT candidates for the relevant literals, instead of gathering the possible RAT candidates by iterating over the whole clause database for each RAT proof, which is expensive. Literals that are not used in RAT proofs at all do not occur in the list. This item is the first item of the proof.

unit-prop For each listed clause ID, the corresponding clause is checked to be unit, and the unit literal is assigned to true. Here, a clause is unit if the unit literal is undecided, and all other literals are assigned to false.

deletion The specified IDs are removed from the clause map.

rup-lemma The item specifies the ID for the new lemma, which is the next unprocessed lemma from the lemma file, a list of unit clause IDs, and a conflict clause ID. First, the literals of the lemma are assigned to false. The lemma must not be blocked, i. e. none of its literals may be already assigned to true[4]. Note that assigning the literals of a clause C to false is equivalent to adding the conjunct $\neg C$ to the formula. Second, the unit clauses are checked and the corresponding unit literals are assigned to true. Third, it is checked that the conflict clause ID actually identifies a conflict clause, i. e. that all its

[4] Blocked lemmas are useless for unsat proofs, such that there is no point to include them into the certificate.

literals are assigned to false. Finally, the lemma is added to the clause-map and the assignment is rolled back to the state before checking of the item started.

rat-lemma The item specifies a pivot literal l, an ID for the lemma, an initial list of unit clause IDs, and a list of candidate proofs. First, as for `rup-lemma`, the literals of the lemma are assigned to false and the initial unit propagations are performed. Second, it is checked that the provided RAT candidates are exhaustive, and the corresponding `cand-prf` items are processed: A `cand-prf` item consists of the ID of the candidate clause D, a list of unit clause IDs, and a conflict clause ID. To check a candidate proof, the literals of $D \setminus \{\neg l\}$ are assigned to false, the listed unit propagations are performed, and the conflict clause is checked to be actually conflict. Afterwards, the assignment is rolled back to the state before checking the candidate proof. Third, when all candidate proofs have been checked, the lemma is added to the clause map and the assignment is rolled back.

To simplify certificate generation in backward mode, we allow candidate proofs referring to arbitrary, even invalid, clause IDs. Those proofs must be ignored by the checker.

conflict This is the last item of the certificate. It specifies the ID of the *root conflict* clause, i. e. the conflict found by unit propagation after adding the last lemma of the certificate. It is checked that the ID actually refers to a conflict clause.

4 Program Verification with Isabelle/HOL

Isabelle/HOL [34] is an interactive theorem prover for higher order logic. Its design features the LCF approach [14], where a small logical inference kernel is the only code that can produce theorems. Bugs in the non-kernel part may result in failure to prove a theorem, but never in a false proposition being accepted as a theorem. Isabelle/HOL includes a code generator [15–17] that translates the executable fragment of HOL to various functional programming languages, currently OCaml, Standard ML, Scala, and Haskell. Via Imperative HOL [5], the code generator also supports imperative code, modeled by a heap monad inside the logic.

A common problem when verifying efficient implementations of algorithms is that implementation details tend to obfuscate the proof and increase its complexity. Hence, efficiency of the implementation is often traded for simplicity of the proof. A well-known approach to this problem is stepwise refinement [1, 2, 43], where an abstract version of the algorithm is refined towards an efficient implementation in multiple correctness preserving steps. The abstract version focuses on the algorithmic ideas, leaving open the exact implementation, while the refinement steps focus on more and more concrete implementation aspects. This modularizes the correctness proof, and makes verification of complex algorithms manageable in the first place.

For Isabelle/HOL, the Isabelle Refinement Framework [22, 24, 25, 29] provides a powerful stepwise refinement tool chain, featuring a nondeterministic shallowly embedded programming language [29], a library of efficient collection data structures and generic algorithms [24–26], and convenience tools to simplify canonical refinement steps [22,24]. It has been used for various software verification projects (e. g. [23,27,28]), including a fully fledged verified LTL model checker [4,12].

5 A Verified GRAT Certificate Checker

We give an overview of our Isabelle/HOL formalization of a GRAT certificate checker (cf. Section 3). We use the stepwise refinement techniques provided by the Isabelle Refinement Framework to verify an efficient implementation at manageable proof complexity.

Note that we display only slightly edited Isabelle source text, and try to explain its syntax as far as needed to get a basic understanding. Isabelle uses a mixture of common mathematical notations and Standard ML [30] syntax (e. g. there are algebraic data types, function application is written as `f x`, functions are usually curried, e. g. `f x y`, and abstraction is written as `λx y. t`).

5.1 Syntax and Semantics of Formulas

The following Isabelle text specifies the abstract syntax of CNF formulas:

```
datatype 'a literal = Pos 'a | Neg 'a
type_synonym 'a clause = 'a literal set
type_synonym 'a cnf = 'a clause set
```

We abstract over the type `'a` of variables, use an algebraic data type to specify positive and negative literals, and model clauses as sets of literals, and a CNF formula as set of clauses.

A partial assignment has type `'a ⇒ bool option`, which is abbreviated as `'a ⇀ bool` in Isabelle. It maps a variable to `None` for undecided, or to `Some True` or `Some False`. We specify the semantics of literals and clauses as follows:

```
primrec sem_lit' :: 'a literal ⇒ ('a⇀bool) ⇀ bool where
  sem_lit' (Pos x) A = A x | sem_lit' (Neg x) A = map_option Not (A x)
definition sem_clause' C A ≡
  if (∃l∈C. sem_lit' l A = Some True) then Some True
  else if (∀l∈C. sem_lit' l A = Some False) then Some False
  else None
```

Note that we omitted the type specification for `sem_clause'`, in which case Isabelle automatically infers the most general type.

For a fixed formula F, we define the *models* induced by a partial assignment to be all total extensions that satisfy the formula. We define two partial assignments to be *equivalent* if they induce the same models.

5.2 Unit Propagation and RAT

We define a predicate to state that, wrt. a partial assignment A, a clause C is unit, with unit literal l:

```
definition is_unit_lit A C l
  ≡ l∈C ∧ sem_lit' l A = None ∧ sem_clause' (C-{l}) A = Some False
```

Assigning a unit literal to true yields an equivalent assignment:

```
lemma unit_propagation:
  assumes C∈F and is_unit_lit A C l
  shows equiv' F A (assign_lit A l)
```

In Isabelle, all variables that occur free in a lemma (here: C,F,A,l) are implicitly universally quantified.

Having formalized the basic concepts, we can show the essential lemma that justifies RAT (cf. Section 2):

```
lemma abs_rat_criterion:
  assumes l∈C and sem_lit' l A ≠ Some False
  assumes ∀D∈F. neg_lit l ∈ D ⟹ implied_clause F A (C∪(D-{neg_lit l}))
  shows redundant_clause F A C
```

Where a clause is *implied* if it can be added to the formula without changing the models, and it is *redundant* if adding the clause preserves satisfiability (but not necessarily the models).

5.3 Abstract Checker Algorithm

Having formalized the basic theory of CNF formulas wrt. partial assignments, we can specify an abstract version of the certificate checker algorithm. Our specifications live in an exception monad stacked onto the nondeterminism monad of the Isabelle Refinement Framework. Exceptions are used to indicate failure of the checker, and are never caught. We only prove soundness of our checker, i. e. that it does not accept satisfiable formulas. Our checker actually accepted all certificates in our benchmark set (cf. Section 7), yielding an empirical argument that it is sufficiently complete.

At the abstract level, we model the proof as a stream of integers. On this, we define functions parse_id and parse_lit that fetch an element from the stream, try to interpret it as ID or literal, and fail if this is not possible. The state of the checker is a tuple (last_id,CM,A). To check that the lemma IDs are strictly ascending, last_id stores the ID of the last processed lemma. The *clause map* CM contains the current formula as a mapping from IDs to clauses, and also maintains the RAT candidate lists. Finally, A is the current assignment.

As first example, we present the abstract algorithm that is invoked after reading the item-type of a rup-lemma item (cf. Section 3), i. e. we expect a sequence of the form id id* "0" id.

```
check_rup_proof ≡ λ(last_id,CM,A₀) it prf. do {
  (i,prf) ← parse_id prf;
  check (i>last_id);
  (C,A',it) ← parse_check_blocked A₀ it;
  (A',prf) ← apply_units CM A' prf;
  (confl_id,prf) ← parse_id prf;
```

```
 7    confl ← resolve_id CM confl_id;
 8    check (sem_clause' confl A' = Some False);
 9    CM ← add_clause i C CM;
10    return ((i,CM,A₀),it,prf)
11  }
```

We use do-notation to conveniently express monad operations. First, the lemma ID is pulled from the proof stream (line 2) and checked to be greater than *last_id* (3). The `check` function throws an exception unless the first argument evaluates to true. Next, *parse_check_blocked* (4) parses the next lemma from the lemma file, checks that it is not blocked, and assigns its literals to false. Then, the function *apply_units* (5) pulls the unit clause IDs from the proof stream, checks that they are actually unit, and assigns the unit literals to true. Finally, we pull the ID of the conflict clause (6), obtain the corresponding clause from the clause map (7), check that it is actually conflict (8), and add the lemma to the clause map (9). We return (10) the lemma ID as new last ID, the new clause map, and the *old* assignment, as the changes to the assignment are local and must be backtracked before checking the next clause. Additionally, we return the new position in the lemma file (*it*) and in the proof stream (*prf*). Note that this abstract specification contains non-algorithmic parts: For example, in line 8, we check for the semantics of the conflict clause to be *Some False*, without specifying how to implement this check. We prove the following lemma for *check_rup_proof*:

```
lemma check_rup_proof_correct:
  assumes invar (last_id,CM,A)
  shows check_rup_proof (last_id,CM,A) it prf
      ≤ spec True (λ((last_id',CM',A'), it', prf').
          invar (last_id',CM',A') ∧ (sat' (cm_F CM) A ⟹ sat' (cm_F CM') A'))
```

Here, `spec` Φ Ψ describes the postcondition Φ in case of an exception, and the postcondition Ψ for a normal result. As we only prove soundness of the checker, we use *True* as postcondition for exceptions. For normal results, we show that an invariant on the state is preserved, and that the resulting formula and partial assignment is satisfiable if the original formula and partial assignment was.

Finally, we present the specification of the checker's main function:

```
definition verify_unsat F_begin F_end it prf ≡ do {
  let A = λ_. None;
  (CM,prf) ← init_rat_counts prf;
  (CM,last_id) ← read_cnf F_end F_begin CM;
  let s = (last_id,CM,A);
  (so,_) ← while (λ(so,it). so≠None) (λ(so,it).
  do {
    let (s,it,prf) = the so;
    check_item s it
  }) (Some (s,it,prf));
}
```

The parameters *F_begin* and *F_end* indicate the range that hold the representation of the formula, *it* points to the first lemma, and *prf* is the proof stream. After initializing the assignment (line 2, all variables undecided), the RAT literal counts are read (3), and the formula is parsed into the clause map (4). Then, the function iterates over the proof stream and checks each item (6–10), until the formula has been certified. (or an exception terminates the program) Here, the checker's state is wrapped into an option type, where *None* indicates that the formula has been certified. Correctness of the abstract checker is expressed by the following lemma:

```
lemma verify_unsat_correct:
  assumes seg F_begin lst F_end
  shows verify_unsat F_begin F_end it prf
      ≤ spec True (λ_. F_invar lst ∧ ¬sat (F_α lst))
```

Intuitively, if the range from *F_begin* to *F_end* is valid and contains the sequence *lst*, and if *verify_unsat* returns a normal value, then *lst* represents a valid CNF formula (*F_invar lst*) that is unsatisfiable (¬*sat* (*F_α lst*)). Note that the correctness statement does not depend on the lemmas (*it*) or the proof stream (*prf*). This will later allow us to use an optimized (unverified) implementation for streaming the proof, without impairing the formal correctness statement.

5.4 Refinement towards an Efficient Implementation

The abstract checker algorithm that we described so far contains non-algorithmic parts and uses abstract types like sets. Even if we could extract executable code, its performance would be poor: For example, we model assignments as functions. Translating this directly to a functional language results in assignments to be stored as long chains of function updates with worst-case linear time lookup.

We now show how to refine the abstract checker to an efficient algorithm, replacing the specifications by actual algorithms, and the abstract types by efficient data structures. The refinement is done in multiple steps, where each step focuses on different aspects of the implementation. Formally, we use a *refinement relation* that relates objects of the refined type (e. g. a hash table) to objects of the abstract type (e. g. a set). In our framework, refinement is expressed by propositions of the form $(c,a)\in R \Longrightarrow g\ c \le\Downarrow S\ (f\ a)$: if the concrete argument c is related to the abstract argument a by R, then the result of the concrete algorithm $g\ c$ is related to the result of the abstract algorithm $f\ a$ by S. Moreover, if the concrete algorithm throws an exception, the abstract algorithm must also throw an exception.

In the first refinement step, we record the set of variables assigned during checking a lemma, and use this set to reconstruct the original assignment from the current assignment after the check. This saves us from copying the whole original assignment before each check. Formally, we define an A_0*-backtrackable assignment* to be an assignment A together with a set of assigned variables T, such that unassigning the variables in T yields A_0. The relation *bt_assign_rel* relates A_0-backtrackable assignments to plain assignments:

```
bt_assign_rel A0 ≡ { ((A,T),A) | A T. T ⊆ dom A ∧ A0 = A↾(-T) }
```

We define `apply_units_bt`, which operates on A_0-backtrackable assignments. If applied to assignments (`A′,T`) and `A` related by `bt_assign_rel` A_0, and to the same proof stream position `prf`, then the results of `apply_units_bt` and `apply_units` are related by `bt_assign_rel` A_0 × `Id`, i. e. the returned assignments are again related by `bt_assign_rel` A_0, and the new proof stream positions are the same (related by `Id`):

```
lemma apply_units_bt_refine: assumes ((A′,T),A)∈bt_assign_rel A0
  shows apply_units_bt CM A′ T prf
    ≤ ⇓(bt_assign_rel A0 × Id) (apply_units CM A prf)
```

In the next refinement step, we implement clauses by iterators pointing to the start of a null-terminated sequence of integers. Thus, the clause map will only store iterators instead of (replicated) clauses. Now, we can specify algorithms for functions on clauses. For example, we define:

```
check_conflict_clause1 A cref ≡ iterate_clause cref (λl _. do {
  check (sem_lit′ l A = Some False)
}) ()
```

i. e. we iterate over the clause, checking each literal to be false. We show:

```
lemma check_conflict_clause1_refine: assumes CR: (cref,C)∈cref_rel
  shows check_conflict_clause1 A cref
        ≤⇓Id (check (sem_clause′ C A = Some False))
```

where the relation `cref_rel` relates iterators to clauses.

In the next refinement step, we introduce efficient data structures. For example, we implement the iterators by indexes into an array of integers that stores both the formula and the lemmas. For many of the abstract types, we use general purpose data structures from the Isabelle Refinement Framework [24,25]. For example, we refine assignments to arrays, using the `array_map_default` data structure, which implements functions of type `nat⇒′a option` by arrays of type `′b array`. It is parameterized by a relation `R : (′b×′a) set` and a default concrete element `d` that does not correspond to any abstract element (`∄a. (d,a)∈R`). The implementation uses `d` to represent the abstract value `None`. We define:

```
definition vv_rel ≡ {(1, False), (2, True)}
definition assignment_assn ≡ amd_assn 0 id_assn (pure vv_rel)
```

i. e. we implement `Some False` by 1, `Some True` by 2, and `None` by 0. Here, `amd_assn` is the relation of the `array_map_default` data structure[5]. The refined programs and refinement theorems in this step are automatically generated by the Sepref tool [24]. For example, the command

[5] The name suffix `_assn` instead of `_rel` indicates that the data structure may be stored on the heap.

```
sepref_definition check_rup_proof3 is check_rup_proof2
  :: cdb_assn^k * state_assn^d * it_assn^k * prf_assn^d
      → error_assn + state_assn × it_assn × prf_assn
```

takes the definition of *check_rup_proof2*, generates a refined version, and proves the corresponding refinement theorem. The first parameter is refined wrt. *cdb_assn* (refining the set of clauses into an array), the second parameter is refined wrt. *state_assn* (refining the clause map and the assignment into arrays), the third parameter is refined wrt. *it_assn* (refining the iterator into an array index), and the fourth parameter is refined wrt. *prf_assn* (refining the stream position). Exception results are refined wrt. *error_assn* (basically the identity relation), and normal results are refined wrt. *state_assn*, *it_assn*, and *prf_assn*. The x^d and x^k annotations indicate whether the generated function may overwrite a parameter (d like *destroy*) or not (k like *keep*).

By combining all the refinement steps and unfolding some definitions, we prove the following correctness theorem for the implementation of our checker:

```
theorem verify_unsat_impl_correct:
  <DBi ↦_a DB>
    verify_unsat_impl DBi F_end it prf
  <λresult. DBi ↦_a DB * ↑(¬isl result ⟹ formula_unsat_spec DB F_end)>
```

This Hoare triple states that if *DBi* points to an array holding the elements *DB*, and we run *verify_unsat_impl*, the array will be unchanged, and if the return value is no exception, the formula represented by the range *1*...*F_end* in the array is unsatisfiable. We have experimented with many equivalent formulations of *formula_unsat_spec*, trying to reduce the *trusted base*, i. e. the concepts and definitions the specification depends on. A concise one is:

```
definition assn_consistent :: (int ⇒ bool) ⇒ bool
  where assn_consistent σ = (∀x. x≠0 ⟹ ¬ σ (-x) = σ x)
definition formula_unsat_spec DB F_end ≡ (
  let lst = tl (take F_end DB) in
    1 < F_end ∧ F_end ≤ length DB ∧ last lst = 0
    ∧ (∄σ. assn_consistent σ ∧ (∀C∈set (tokenize 0 lst). ∃l∈set C. σ l)))
```

Here, a *consistent assignment* is a mapping from integers to Booleans, such that a negative value is mapped to the opposite as its absolute value. The specification then defines *lst* to be the elements *1,*...*,F_end* of the array[6], and states that *F_end* is in bounds, the last element of *lst* is a null, and that there is *no* assignment such that each clause contains a literal assigned to true. We define *tokenize 0 lst* to be the unique list of lists of non-null integers whose concatenation as null-terminated lists yields *lst*. This way, we specify an unsatisfiable formula down to the list of integers that represents it, only using basic list functions. The last section of the proof outline of our formalization [21] contains a detailed discussion of the correctness theorem.

The final step to a verified efficient unsat checker is to use Isabelle/HOL's code generator to extract Standard ML code for *verify_unsat_impl* and to link

[6] Element 0 is used as a guard in our implementation.

this code with a small (40 LOC) parser to read the formula (and the lemmas) into an array. Moreover, we implement a buffered reader for the proof file. This, however, does not affect the correctness statement, which is valid for all proof stream implementations. The resulting program is compiled with MLton [31].

6 Multithreaded Generation of Enriched Certificates

In order to generate GRAT certificates, we extend a DRAT checker algorithm to record the unit clauses that lead to a conflict when checking each lemma.

Our certificate generator started as a reimplementation of the backward mode of drat-trim [10,41] in C++, to which we then added GRAT certificate generation. As the certificate generator is not part of the trusted code base, we could afford to add aggressive novel optimizations: We maintain separate watchlists for marked and unmarked lemmas, which allows a more efficient implementation of core-first unit propagation. Moreover, we detect runs of lemmas with the same pivot element, which allows to reuse the results of (expensive) RAT candidate searches in certain cases. These optimizations alone make our generator more than two times faster than drat-trim.

Another common optimization is parallelization: If one has more DRAT certificates to check than processors available (e. g. when evaluating a SAT competition), one can simply run multiple instances of the certificate generator and checker in parallel. However, if one has only a few certificates to check (e. g. when using SAT solvers for checking a single model), a more fine grained parallelization is required to keep the available processors busy. To this end, our certificate generator provides a multi-threaded mode, which parallelizes the processing of lemmas, at the cost of using more memory. It uses all optimizations of the single-threaded mode, some of them slightly adjusted for multi-threading. For example, the lemmas of a run with the same pivot element are preferably scheduled to the same thread.

The basic idea is to let multiple threads run backwards over the certificate, verifying the lemmas in parallel. A thread tries to acquire a lemma before it starts verification. If the lemma is already acquired by another thread, this thread proceeds with the next lemma. This way, each lemma is only proved by one thread. For the marking of lemmas, the only required synchronization is that a thread sees its own markings: As every thread runs to the beginning, and on processing a lemma only earlier lemmas are marked, every thread will try to acquire at least the lemmas that it marked itself — and process them if no other thread was faster. However, in order to improve the effectiveness of core-first unit propagation, the threads periodically synchronize on their marking data.

7 Benchmarks

We present the experimental evaluation of our tools on a realistic set of benchmarks. We used CryptoMiniSat [37,40] to generate DRAT certificates for the 110 unsatisfiable problems it solved at the 2016 SAT competition [38]. We ran the benchmarks on a standard server board with a 22 core Intel XEON Broadwell processor with 2.2 GHz and 128 GiB of RAM. To minimize interferences, we ran only one benchmark at a time, with no other load on the server. Due to the page limit of this paper, we only provide a short summary of our benchmark results. The complete results are available on the tool's homepage [20].

On each DRAT certificate, we ran drat-trim (version Nov 10 2016)[7] and our tool chain (version 1.2) with 1 and 8 threads. We measured the wall-clock time and memory consumption. First of all, our tools successfully checked all certificates, indicating that our approach is sufficiently complete. (Recall that only soundness is formally proved)

We start with comparing drat-trim to our tool in single-threaded mode: drat-trim timed out after 20.000 seconds on two certificates, and crashed on a third one. For checking the remaining 107 certificates, drat-trim required 42.3 hours, while our tool chain required only 17.3 hours. Out of the 17.3 hours, only 1.1 hours were required to run the verified certificate checker, i. e. its runtime is almost negligible compared to certificate generation time. Our tool-chain verified the three certificates for which drat-trim failed in 5.3 hours.

Our certificate generator requires roughly two times more memory than drat-trim. This is due to the generated certificate being stored in memory. We could not measure meaningful memory consumption values for our verified checker: The MLton garbage collector only gets active when memory falls short, resulting in unrealistic memory consumption values when being the only process running on a machine with 128 GiB of RAM.

Next, we report on running the certificate generator with 8 threads: The wall clock times required for generation and checking add up to only 8.3 hours. Excluding certificates that required less than one minute to check, the average speed up is 2.6 [min: 1.1, max: 4.9] compared to single-threaded mode, and 7.1 [min: 0.5, max: 36.0] compared to drat-trim. However, certificate generation requires significantly more memory, as the DRAT certificate is duplicated for each thread.

To complete the presentation, we briefly report on the results of our formally verified satisfiability checker: The certificates for the 64 satisfiable problems that CryptoMiniSat solved at the 2016 SAT competition [38] have a size of 229 MiB and could be verified in 40 seconds.

8 Conclusions

We have presented a formally verified tool chain to check DRAT unsatisfiability certificates. In single-threaded mode, our approach is more than two times faster than the (unverified) standard tool drat-trim, on a benchmark suite taken from the 2016 SAT competition. Additionally, we implemented a multi-threaded mode, which allows us to trade computing resources for significantly smaller response times. The formal proof covers the actual implementation of the checker and the semantics of the formula down to the sequence of integers that represents it.

[7] The current version at the time of writing this paper.

Our approach involves two phases: The first phase generates an enriched certificate, which is then checked against the original formula by the second phase. While the main computational work is done by the first phase, soundness of the approach only depends on the second phase, which is also algorithmically less complex, making it more amenable to formal verification. Using stepwise refinement techniques, we were able to formally verify a rather efficient implementation of the second phase.

We conclude with some statistics: The formalization of the certificate checker is roughly 5k lines of code. In order to realize this formalization, several general purpose libraries (e. g. the exception monad and some imperative data structures) had to be developed. These sum up to additional 3.5k lines. The time spent on the formalization was roughly three man months. The multi-threaded certificate generator has roughly 3k lines of code, and took two man month to develop.

8.1 Future Work

Currently, the formal proof of our verified checker goes down to the representation of the formula as integer array, thus requiring a (small) unverified parser. A logical next step would be to verify the parser, too. Moreover, verification stops at the Isabelle code generator, whose correctness is only proved the classical way on paper [16, 17]. There is work aiming at the mechanical verification of code generators [33], and even the subsequent compilers [19]. Unfortunately, this is not (yet) available for Isabelle/HOL.

We plan to attack the high memory consumption of our multi-threaded generator by trying to share more (read-only) data between the threads.

An interesting research topic would be to integrate enriched certificate generation directly into SAT solvers. The performance decrease in the solver could be weighed against the cost of generating an enriched certificate. However, such modifications are probably complex and SAT-solver specific, whereas DRAT certificates are designed to be easily integrated into virtually any CDCL based SAT solver.

Finally, we chose a benchmark set which is realistic, but can be run in a few days on the available hardware. We plan to run our tools on larger benchmark suites, once we have access to sufficient (supercomputing) hardware.

Acknowledgements We thank Jasmin Blanchette and Mathias Fleury for very useful comments on the draft version of this paper, and Lars Hupel for instant help on any problems related to the benchmark server.

References

1. R.-J. Back. *On the correctness of refinement steps in program development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.
2. R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction.* Springer, 1998.
3. Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* Springer, 1st edition, 2010.
4. J. Brunner and P. Lammich. Formal verification of an executable LTL model checker with partial order reduction. In *Proc. of NFM*, pages 307–321. Springer, 2016.
5. L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOL*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008.
6. L. Cruz-Filipe, M. Heule, W. Hunt, K. Matt, and P. Schneider-Kamp. Efficient certified RAT verification. In *Proc. of CADE.* Springer, 2017. to appear.
7. L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. *Efficient Certified Resolution Proof Checking*, pages 118–135. Springer, 2017.
8. A. Darbari, B. Fischer, and J. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In *Proc. of ICTAC*, pages 260–274. Springer, 2010.
9. DRAT-trim github repository. `https://github.com/marijnheule/drat-trim`.
10. DRAT-trim homepage. `https://www.cs.utexas.edu/~marijn/drat-trim/`.
11. DRAT-trim issue tracker. `https://github.com/marijnheule/drat-trim/issues`.
12. J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.
13. E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proc. of DATE.* IEEE, 2003.
14. M. Gordon. From LCF to HOL: A short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.
15. F. Haftmann. *Code Generation from Specifications in Higher Order Logic.* PhD thesis, Technische Universität München, 2009.
16. F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In *Proc. of ITP*, pages 100–115. Springer, 2013.
17. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, LNCS. Springer, 2010.
18. M. Heule, W. Hunt, and N. Wetzler. Trimming while checking clausal proofs. In *2013 Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 181–188. IEEE, 2013.
19. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ML. In *Proc. of POPL*, pages 179–192. ACM, 2014.
20. P. Lammich. Grat tool chain homepage. `http://www21.in.tum.de/~lammich/grat/`.
21. P. Lammich. Gratchk proof outline. `http://www21.in.tum.de/~lammich/grat/outline.pdf`.
22. P. Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.
23. P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *ITP*, volume 8558 of *LNCS*, pages 325–340. Springer, 2014.

24. P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.
25. P. Lammich. Refinement based verification of imperative data structures. In *CPP*, pages 27–36. ACM, 2016.
26. P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In *Proc. of ITP*, volume 6172 of *LNCS*, pages 339–354. Springer, 2010.
27. P. Lammich and R. Neumann. A framework for verifying depth-first search algorithms. In *CPP '15*, pages 137–146, New York, NY, USA, 2015. ACM.
28. P. Lammich and S. R. Sefidgar. Formalizing the Edmonds-Karp algorithm. In *Proc. of ITP*, pages 219–234, 2016.
29. P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.
30. R. Milner, R. Harper, D. MacQueen, and M. Tofte. *The Definition of Standard ML (MIT Press)*. The MIT Press, 1997.
31. MLton Standard ML compiler. `http://mlton.org/`.
32. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC*, pages 530–535. ACM, 2001.
33. M. O. Myreen and S. Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.
34. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
35. SAT competition, 2013. `http://satcompetition.org/2013/`.
36. SAT competition, 2014. `http://satcompetition.org/2014/`.
37. *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, volume B-2016-1. University of Helsinki, 2016.
38. SAT competition, 2016. `http://baldur.iti.kit.edu/sat-competition-2016/`.
39. C. Sinz and A. Biere. Extended resolution proofs for conjoining bdds. In *Proc. of CSR*, pages 600–611. Springer, 2006.
40. M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT 2009*, pages 244–257. Springer, 2009.
41. N. Wetzler, M. J. H. Heule, and W. A. Hunt. Mechanical verification of SAT refutations with extended resolution. In *Proc. of ITP*, pages 229–244. Springer, 2013.
42. N. Wetzler, M. J. H. Heule, and W. A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Proc. of SAT 2014*, pages 422–429. Springer, 2014.
43. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), Apr. 1971.