Journal of Automated Reasoning (2020) 64:513–532
https://doi.org/10.1007/s10817-019-09525-z

Efficient Verified (UN)SAT Certificate Checking

Peter Lammich[1]

Received: 29 April 2019 / Accepted: 2 May 2019 / Published online: 4 June 2019
© The Author(s) 2019

Abstract
SAT solvers decide the satisfiability of Boolean formulas in conjunctive normal form. They are commonly used for software and hardware verification. Modern SAT solvers are highly complex and optimized programs. As a single bug in the solver may invalidate the verification of many systems, SAT solvers output certificates for their answer, which are then checked independently. However, even certificate checking requires highly optimized non-trivial programs. This paper presents the first SAT solver certificate checker that is formally verified down to the integer sequence representing the formula. Our tool supports the full DRAT standard, and is even faster than the unverified state-of-the-art tool *drat-trim*, on a realistic set of benchmarks drawn from the 2016 and 2017 SAT competitions. An optional multi-threaded mode further reduces the runtime, in particular for big certificates.

Keywords Unsat certificates · SAT solving · DRAT · Isabelle/HOL · Stepwise refinement · Formal verification · Verified software

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. [44–46]). Since 2013, the SAT competition [40] requires solvers to output unsat certificates. Since 2014, only certificates in the DRAT format [46] are accepted [41].

The standard tool to check DRAT certificates is *drat-trim* [9,46]. It is a highly optimized C program with many features, including forward and backward checking modes, 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

✉ Peter Lammich
peter.lammich@manchester.ac.uk

1 School of Computer Science, University of Manchester Oxford Road, M13 9PL Manchester, UK

of DRAT certificates, 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 [10], 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.

One method to ensure correctness of 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 [38] 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. [8,45]) 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 approach still has some deficits: GRIT only supports the less powerful DRUP fragment [45] of DRAT, making it unsuitable for SAT solvers that output full DRAT. 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. In contrast, an unverified implementation of their checker in C is two orders of magnitude faster.

Independently of us, Cruz-Filipe et al. also extended their tool to DRAT [6], and optimized their verified checker [18]. Their tool is called LRAT.

In this paper we present the GRAT toolchain: An enriched certificate format for full DRAT, a highly optimized certificate generator, and a certificate checker whose correctness is formally verified down to the integer array representing the formula. The simple unverified parser that reads a formula into an integer array is written in Standard ML [34], which guarantees that numeric and buffer overflows will not go unnoticed. On the same basis, we also implement and verify a sat certificate checker, obtaining a complete and formally verified SAT solver certification tool.

We use stepwise refinement techniques to obtain an efficient verified checker, and implement aggressive optimizations in the generator. A distinguishing feature is a multi-threaded mode for the generator, which allows us to trade computing resources for additional speedup.

We benchmark our tool against *drat-trim* and LRAT on a realistic benchmark suite drawn from the 2016 and 2017 SAT competitions: Already in single-threaded mode, our tool is faster than LRAT on every single problem, and, on most problems, even faster than the (unverified) *drat-trim*. In multi-threaded mode with 8 threads, we get an average speedup of 2.2.

This paper is an extended version of our conference papers [28,29]. It provides a unified description of both the certificate generator and checker, and extends the benchmark set to

include problems from the 2017 SAT competition. Our tools, formalizations, and benchmark results are available online [23].

The rest of this paper is organized as follows: After briefly recalling the theory of DRAT certificates (Sect. 2), we introduce our enriched certificate format (Sect. 3). We then give a short overview of the Isabelle Refinement Framework (Sect. 4) and describe its application to verifying our certificate checker (Sect. 5). Next, we describe our certificate generator (Sect. 6) and report on the experimental evaluation of our tools (Sect. 7). Finally, we discuss future work (Sect. 8) and give a conclusion (Sect. 9).

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 $F = C_1 \wedge \cdots \wedge C_n$ for $C_i \in 2^L$ be a formula in conjunctive normal form (CNF). F is *satisfied* by an *assignment* $A : V \Rightarrow$ 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*. We name the result of unit propagation F^{u}, defining $F^{\mathrm{u}} = \{\emptyset\}$ if unit propagation yields a conflict.[1]

A DRAT certificate $\chi = \chi_1 \cdots \chi_n$ 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:

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

where $F \backslash C$ 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:

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

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 $(F \wedge \neg(C \cup D \backslash \{\neg l\}))^{\mathrm{u}} = \{\emptyset\}$. Adding a lemma with the RAT property to a satisfiable formula preserves satisfiability [45], and so do unit propagation and deletion of clauses. Thus, existence of a valid DRAT certificate implies unsatisfiability of the original formula.

A more restrictive 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 [19], 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

[1] This is well-defined as unit-propagation is strongly normalizing up to conflicts.

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* [13,19]: 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 [46] prefers marked unit clauses over unmarked ones.

In practice, DRAT certificate checkers spend most time on unit propagation,[2] for which highly optimized implementations of rather complex algorithms are used (e. g. *drat-trim* uses a two-watched-literals algorithm [36]). 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, *finding* the next unit or conflict clause is replaced by simply *checking* whether a clause is unit or conflict. A modified version of *drat-trim* can be 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.

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 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 (stored in reverse order) of 32 bit signed integers in 2's complement little endian format. The sequence is 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

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

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 count 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.[3] 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 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 f or 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 then 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 $D \backslash \{\neg l\}$ 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 conflict clause found by unit propagation after adding the last lemma of the certificate (*root conflict*). It is checked that the ID actually refers to a conflict clause.

4 Program Verification with Isabelle/HOL

Isabelle/HOL [38] 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

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

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 [1,2,48], 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 [24,26,27,32] provides a powerful stepwise refinement tool chain, featuring a nondeterministic shallowly embedded programming language [32], a library of efficient collection data structures and generic algorithms [26,27,30], and convenience tools to simplify canonical refinement steps [24,26]. It has been used for various software verification projects (e. g. [25,31,47]), including a fully fledged verified LTL model checker [4,11].

5 A Verified GRAT Certificate Checker

We give an overview of our Isabelle/HOL formalization of a GRAT certificate checker (cf. Sect. 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/HOL source text, and try to explain its syntax as far as needed to get a basic understanding. Isabelle/HOL uses a mixture of common mathematical notations and Standard ML [34] 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

For the abstract syntax of CNF formulas, we represent variables by natural numbers, use an algebraic data type to specify positive and negative literals, model clauses as sets of literals, and a CNF formula as a set of clauses:

```
datatype literal = Pos nat | Neg nat
type_synonym clause = literal set
type_synonym cnf = clause set
```

The concrete syntax that our tool accepts is a list (array) of integers, representing a formula in the well-known DIMACS format. Variables are positive natural numbers. Literals are nonzero integers of the form v or $-v$, representing positive and negative literals on variable v. A clause is a list of literals, and a formula is the concatenation of its clauses, separated and terminated by nulls. The following definitions specify the restrictions on the concrete syntax (`xxx_invar`) and the translation from concrete to abstract syntax (`xxx_α`):

```
definition lit_invar l ≡ l≠0
definition lit_α l ≡ if l<0 then Neg (nat (-l)) else Pos (nat l)
definition clause_invar l ≡ ∀x∈set l. lit_invar x
definition clause_α l ≡ lit_α`set l
definition F_invar lst ≡ lst≠[] ⟹ last lst = 0
definition F_α lst ≡ set (map clause_α (tokenize lst))
```

where `nat` converts an integer to a natural number, `set` converts a list to the set of its elements, and `tokenize l` splits the concatenation of null-terminated lists into a list of lists. Note that every list that ends with a null represents a valid formula.

We define the semantics of literals, clauses, and formulas wrt. a *valuation*, which is a function from variables to Booleans. A positive literal is true if its variable is assigned to true, a negative literal is true if its variable is false, a clause is true if it contains a true literal, and a formula is true if all its clauses are true:

```
type_synonym valuation = nat ⇒ bool
fun sem_lit :: literal ⇒ valuation ⇒ bool where
  sem_lit (Pos x) σ = σ x
| sem_lit (Neg x) σ = ¬ σ x
definition sem_clause :: clause ⇒ valuation ⇒ bool where
  sem_clause C σ ≡ ∃l∈C. sem_lit l σ
definition sem_cnf :: cnf ⇒ valuation ⇒ bool where
  sem_cnf F σ ≡ ∀C∈F. sem_clause C σ
```

Note that type specifications on constant definitions are optional in Isabelle/HOL, and if they are omitted, the most general type is inferred automatically.

We define the *models* of a formula to be the set of all valuations that make the formula true, and we define a formula to be satisfiable if it has a model:

```
definition models F ≡ {σ. sem_cnf F σ}
definition sat F ≡ models F ≠ {}
```

While unit propagation can be presented by modifying the formula (removing false literals and true clauses), practical implementations use a partial assignment (where variables can be true, false, or undecided) and do not change the formula on unit propagation. At this point, we have the design choice to either formalize unit-propagation by modifying the formula, and then refine this model to partial assignments, or to formalize unit propagation on partial assignments directly. We decided for the latter, as we found it to be convenient, and it saves the overhead of one refinement step.

A *partial assignment* has type `nat ⇒ bool option`, which is abbreviated as `nat ⇀ bool`. 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' :: literal ⇒ (nat⇀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
```

For a fixed formula F, we define the models induced by a partial assignment A to be all total extensions that satisfy the formula, and a predicate `sat'` which holds iff such a model exists:

```
definition compat_assignment :: (nat ⇀ bool) ⇒ valuation ⇒ bool
  where compat_assignment A σ ≡ ∀x v. A x = Some v ⟹ σ x = v
definition models' F A ≡ models F ∩ {A. compat_assignment A}
definition sat' F A ≡ models' F A ≠ {}
```

Obviously, a formula is satisfiable wrt. the empty partial assignment if, and only if, it is satisfiable:

```
lemma sat'_empty_iff: sat' F Map.empty = sat F
```

Two assignments A and A' are *equivalent* iff they induce the same models:

```
definition equiv' F A A' ≡ models' F A = models' F A'
```

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)
```

For a fixed formula F and assignment A, a clause C is *implied* if adding it to F does not change the models, and *redundant* if it does not change satisfiability:

```
definition implied_clause F A C ≡ models' (insert C F) A = models' F A
definition redundant_clause F A C ≡ sat' (insert C F) A = sat' F A
```

Recall that DRAT proofs work by deleting clauses, adding redundant clauses, and applying unit propagations, until the formula becomes trivially unsatisfiable (cf. Sect. 2). A clause is accepted as redundant if it has the RAT property. Abstractly, the RAT property is justified by the following lemma:

```
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
```

To test whether a clause is implied, we use the RUP property:

```
lemma one_step_implied:
  assumes RC: ¬is_blocked A C ⟹
    ∃A₁. equiv' F (and_not_C A C) A₁ ∧ (∃E∈F. is_conflict_clause A₁ E)
  shows implied_clause F A C
```

where the assignment A_1 will be computed by unit propagation.

5.3 Abstract Checker Algorithm

Having formalized the basic theory of CNF formulas, unit propagation, and RAT, 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. Sect. 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 an ID or literal, and fail if this is not possible. The state of the checker is a tuple (`CM,A`). The *clause map* `CM` contains the current formula as a mapping from IDs to clauses, and also maintains the RAT candidate database. The *assignment* `A` is the current partial assignment.

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

```
1   check_rup_proof ≡ λ(CM,A0) it prf. do {
2     (i,prf) ← parse_id prf;
3     check (i∉cm_ids CM);
4     (C,A′,it) ← parse_check_blocked A0 it;
5     (A′,prf) ← apply_units CM A′ prf;
6     (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 ((CM,A0),it,prf)
11  }
```

We use do-notation to conveniently express monadic programs. First, the ID for the new lemma is pulled from the proof stream (line 2) and checked to be available (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 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 the new 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 (CM,A)
  shows check_rup_proof (CM,A) it prf
      ≤ spec True (λ((CM′,A′), it′, prf′).
          invar (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,[4] and that the resulting formula and partial assignment is satisfiable if the original formula and partial assignment was.

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

```
1   definition verify_unsat F_begin F_end it prf ≡ do {
2     (CM,prf) ← init_rat_counts prf;
```

[4] The invariant states that there are no syntactic tautologies, i. e. clauses that contain both a positive and negative literal over the same variable, and that the RAT candidate database, which is used to quickly identify RAT candidates (cf. Sect. 3), is accurate.

```
3     CM ← read_cnf F_end F_begin CM;
4     let s = (CM, λ_. None);
5     while (λso. so≠None) (λso. do {
6       let (s,it,prf) = the so;
7       check_item s it
8     }) (Some (s,it,prf));
9   }
```

The parameters *F_begin* and *F_end* indicate the range that holds the representation of the formula, *it* points to the first lemma, and *prf* is the proof stream. First, the RAT literal counts are read (2) and the formula is parsed into the clause map (3). Then, the assignment is initialized to everything undecided (4). The function then iterates over the proof stream and checks each item (5–9), 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. The function *the (Some x) = x* extracts the value from an option. 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 ($\neg$*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 refine the abstract checker to an efficient implementation, 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 while 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:

$$bt_assign_rel\ A_0 \equiv \{\ ((A,T),A)\ \mid\ A\ T.\ T \subseteq dom\ A \land A_0 = A{\restriction}(-T)\ \}$$

where $A{\restriction}(-T)$ restricts a partial assignment A to the variables not in 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 *prf*, then the results of *apply_units_bt* and *apply_units* are related by *bt_assign_rel* $A_0 \times$ *Id*, i.e. the returned assignments are again related by *bt_assign_rel* A_0, and the new proof streams 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 (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 [26,27]. 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 : $('b \times 'a)$ *set* and a default concrete element d that does not correspond to any abstract element ($\nexists a.\ (d, a) \in R$). 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 [26]. For example, the command

```
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*

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

(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 prf_next F_end it prf
  <λresult. DBi ↦a DB * ↑(¬isl result ⟹ verify_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 range *1... F_end* in the array[6] represents a valid unsatisfiable formula in DIMACS format:

```
definition verify_unsat_spec DB F_end ≡ 1 ≤ F_end ∧ F_end ≤ length DB ∧
  (let lst = tl (take F_end DB) in F_invar lst ∧ ¬sat (F_α lst))
```

We also define a checker for satisfiability certificates, which are null-terminated lists of non-contradictory literals starting at index *F_end*, and prove:

```
theorem verify_sat_impl_correct:
  <DBi ↦a DB>
    verify_sat_impl DBi F_end
  <λresult. DBi ↦a DB * ↑(¬isl result ⟹ verify_sat_spec DB F_end)>

definition verify_sat_spec DB F_end ≡ 1 ≤ F_end ∧ F_end ≤ length DB ∧
  (let lst = tl (take F_end DB) in F_invar lst ∧ sat (F_α lst))
```

Finally, to obtain our verified sat and unsat checker *gratchk*, Isabelle/HOL's code generator is used to extract Standard ML code for *verify_(un)sat_impl*. We add a command line interface and a small (40 LOC) parser to read the formula 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 [35].

5.5 Concise Correctness Statement

We have shown that our checker only accepts arrays containing (un)satisfiable formulas in DIMACS format. To describe a satisfiable input (cf. Sect. 5.1), we have first mapped the array to a formula (constants *tokenize*, *F_α*, *clause_α*, *lit_α*). Then, we defined a semantics to describe satisfiability of a formula (*sat*, *models*, *sem_cnf*, *sem_clause*, *sem_lit*). In this section, we outline a more direct specification, which only uses elementary list and set operations of Isabelle/HOL, and show that it is equivalent to our original specification. This can be seen as a sanity check for our semantics.

We again use tokenization to convert the input into a list of lists. We further justify tokenization by showing that it is the unique inverse of concatenation:

```
definition concat0 ll = concat (map (λl . l@[0]) ll)
lemma unique_tokenization:
```

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

```
  assumes l≠[] ⟹ last l = 0
  shows ∃₁ls. (0∉⋃set (map set ls) ∧ concat0 ls = l)
    and tokenize l = (THE ls. 0∉⋃set (map set ls) ∧ concat0 ls = l)
```

where *THE* is the definite description operator. Next, we define an assignment from integers to Booleans to be consistent iff a negative value is mapped to the opposite of its absolute value:

```
definition assn_consistent :: (int ⇒ bool) ⇒ bool
  where assn_consistent σ = (∀x. x≠0 ⟹ ¬ σ (-x) = σ x)
```

Finally, we characterize an (un)satisfiable input by the (non)existence of a consistent assignment that assigns at least one literal of each clause to true:

```
lemma verify_sat_spec DB F_end = (1≤F_end ∧ F_end ≤ length DB ∧ (
    let lst = tl (take F_end DB) in
      (lst≠[] ⟹ last lst = 0)
    ∧ (∃σ. assn_consistent σ ∧ (∀C∈set (tokenize 0 lst). ∃l∈set C. σ l))))

lemma verify_unsat_spec DB F_end = (1 < F_end ∧ F_end ≤ length DB ∧ (
    let lst = tl (take F_end DB) in
       last lst = 0
    ∧ (∄σ. assn_consistent σ ∧ (∀C∈set (tokenize 0 lst). ∃l∈set C. σ l))))
```

In the case of unsatisfiability, the bounds have been adjusted to exclude the empty formula, which is trivially satisfiable.

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. As the certificate generator is not part of the trusted code base, we can afford aggressive optimizations. Our generator *gratgen* started as a reimplementation of the backward mode of *drat-trim* [9,45] in C++, to which we added certificate generation. Later, we implemented multithreading and some additional optimizations. The multithreading mode allows us to trade computing resources for faster response time. It makes sense in settings where parallelization on the granularity of whole problems does not exhaust the available computing resources, e. g. when one is interested in a quick answer for a single problem.

The following displays high-level pseudocode of our certificate generator:

```
1  fun forward_phase:
2    F := original formula
3    propagate units in F; if F has conflict then exit "s UNSAT"
4    for item in certificate do
5      if item = d C then
6        remove clause C from F
7      else if item = C then
8        add C to F
9        propagate units in F
10       if F has conflict then
11         mark clauses required for conflict
12         truncate certificate
13         return F

14   exit "s ERROR"

15 fun backward_phase(F):
```

```
16    for item in reverse certificate do
17      if item = d C then
18        add clause to F
19      else if item = C
20        remove C from F; undo unit propagations due to C
21        if is_marked(C) and acquire(C) then
22          verify C and mark required clauses
23          synchronize marked clauses

24  fun main:
25    F = forward_phase
26    for parallel 1..N do
27      backward_phase(copy(F))
```

The initial forward phase starts with the original formula (line 2), and performs unit propagation (3). If this already yields a conflict, the formula is unsatisfiable. Otherwise, we iterate over the certificate (4). Each item of the certificate either deletes (6) or adds (8) a clause. After adding, we perform unit propagation (9), and if this yields a conflict, we mark all clauses required for the conflict (11), discard the remaining items in the certificate if any (12), and finish the forward phase.

Next, the backward phase iterates over the certificate in reverse order (line 16), undoes the effects of the items (18 and 20), verifies the lemmas, and marks all lemmas required for verification (22). Unmarked lemmas are skipped (21). The enriched certificate is generated during the backwards phase, while undoing unit propagations and verifying lemmas. For better readability, we have omitted certificate generation from the above listing.

The backwards phase is parallelized: Each thread maintains its own copy of the clause database. Before proving a lemma, a thread needs to acquire it (21), thus ensuring that each lemma is only proved by a single thread. The information about marked lemmas is periodically synchronized between the threads (23), such that a thread can generate work for other threads. Interestingly, there is no synchronization apart from lemma acquisition and sharing of marked clauses between the threads. In theory, one thread could prove most of the lemmas, while the other threads quickly run to the beginning of the certificate, seeing only very few marked lemmas. However, we have not observed such behavior in practice, and thus did not implement any further synchronization between the threads.

In the remainder of this section, we describe the most important optimizations that we have implemented in *gratgen*: RAT-run heuristics and separate watchlists.

6.1 RAT-Run Heuristics

To verify that a lemma has the RAT property (cf. Sect. 5.2), one has to collect all RAT candidate lemmas, i. e. those clauses in the database that contain the negated pivot literal. As it is not known in advance which of the lemmas will actually be marked and require a full RAT proof (most lemmas are proved by a RUP-proof), maintaining a database of candidate lemmas for each literal would be inefficient. Thus, *drat-trim* iterates over the whole clause database on each RAT proof. Our profiling indicated that a significant amount of the runtime may be spent on searching candidate lemmas. However, we observed that certificates usually contain runs of multiple RAT lemmas with the same pivot. Thus, we store the result of the last search through the database, and reuse it if we should encounter a RAT lemma over the same pivot. Moreover, in multi-threaded mode, we always allocate a run of lemmas with the same pivot to the same thread, as the stored search result is maintained thread-locally. Our benchmarks (Sect. 7) indicate that this optimization is very efficient if actual RAT lemmas

are present. If there are no RAT lemmas, the heuristics' overhead is effectively a single check in the outer loop of the backwards phase, and, as expected, we observed no decrease in performance.

6.2 Separate Watchlists

Another important heuristics, which is already implemented in *drat-trim*, is core-first unit propagation. On unit propagation, marked lemmas are preferred over unmarked ones. This way, the unit clauses used for a proof are more likely to be already marked, thus reducing the overall number of marked lemmas.

Unit propagation is done by a two-watched-literals data structure [36], where, for each clause, two distinct literals are marked as watched. As long as the clause is not blocked or conflict, the two watched literals must be undecided. When assigning a new variable, this invariant may be broken, and is then restored by the unit propagation algorithm: For each clause watching a literal that has been assigned to false, a new watched literal is searched. If no new watched literal can be found, the clause is unit or conflict, in which case the unit literal is assigned to true or unit propagation stops with a conflict. To efficiently iterate over the clauses watching a literal, they are stored in a *watchlist* for each literal.

In *drat-trim*, core first unit propagation is implemented by two iterators over the watchlists of the newly assigned literals. The first iterator ignores unmarked clauses, while the second iterator processes unmarked clauses. The second iterator is only advanced when the first iterator cannot be advanced further. By advancing the second iterator, new literals may be assigned, which makes advancing the first iterator possible again. However, skipping the iterators over irrelevant clauses may yield considerable overhead in the performance sensitive inner loop of unit propagation. Thus, for *gratgen*, we have implemented two watchlists for each literal, one for the marked and one for the unmarked lemmas. This way, the iterators during unit propagation never have to skip over irrelevant clauses. On the other hand, when marking a clause, we have to spend additional time to move it from the unmarked to the marked watchlists.[7] In practice, we found this optimization to be effective.

7 Benchmarks

We have benchmarked GRAT with one and eight threads against *drat-trim* and LRAT [18] on problems taken from the main tracks of the 2016 and 2017 SAT competitions [42,43]. We consider the problems solved by 2017 gold medalist Maple, and the problems solved by 2016 silver medalist Riss6. We chose the silver medalist for 2016, as the gold medalist is, again, Maple. Moreover, we consider the problems solved by CryptoMiniSat in 2016. Although not among the Top 3 solvers, we included CryptoMiniSat because it seems to be the only prover that produces a significant amount of RAT lemmas. For the 2017 competition, only abcdSAT seems to produce RAT lemmas, and we did not include it in our benchmarks.[8]

All tested tools verified all but four unsatisfiability certificates: On two certificates, *drat-trim* timed out (using the default timeout of 20.000 s), and it segfaulted on a third one. A

[7] We have also experimented with lazily moving marked clauses if they are encountered in the unmarked list during unit propagation but this turned out to be less efficient.

[8] Due to constraints on available computation time. Note that this shifts the benchmark results in favor of *drat-trim*, as our tool has optimizations specifically tailored to handle RAT lemmas.

Table 1 Solving times for problems where at least one tool failed

Problem	SAT-solver	drat-trim	GRAT-1	GRAT-8
sokoban-p16.sas.cr.37	cmsat	Timeout	2.3h	39m
valves-gates-1-k617-unsat	cmsat	Timeout	2.1h	27m
sokoban-p20.sas.ex.13	cmsat	SEGV	1.1h	38m
10pipe_k	riss6	52m	1.1h	OOM

The drat-trim column displays the times required by only *drat-trim* (without LRAT). The GRAT-1 column displays the times for the whole GRAT toolchain in single-threaded mode, and GRAT-8 displays the times when using 8 threads

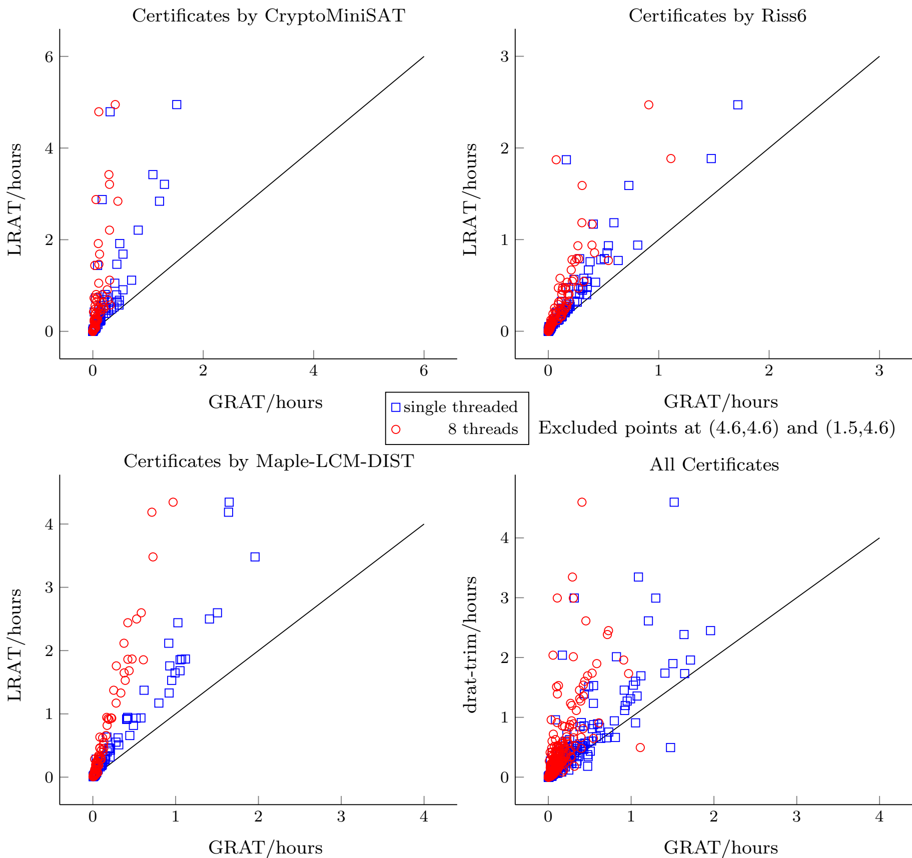

Fig. 1 Comparison of *drat-trim* and GRAT, ran on a server board with a 22-core XEON Broadwell CPU @2.2GHz and 128GiB RAM

fourth certificate led to an out-of-memory error of multithreaded *gratgen*. Table 1 displays the results for these certificates.

Figure 1 shows the results for the other unsatisfiability certificates. The first three scatter plots compare the wall-clock time of LRAT against GRAT with one and eight threads. Here, already single-threaded GRAT is faster than LRAT on every problem. As expected, the difference is more significant on problems that contain RAT lemmas: for the problems that

Table 2 Runtimes in minutes of the second phase, summed over all certificates generated by a prover

Prover	gratchk	lrat-check
Maple	275	346
Cmsat	66	613
Riss6	253	575

actually contain RAT lemmas, single-threaded GRAT is about 3 times faster than LRAT, while, for the other problems, it is only 1.7 times faster.

Except for small problems, multithreading yields a significant speedup: Considering only problems where single-threaded *gratgen* needs longer than 100 s, the average speedup with eight threads is 2.1, the average speedup of the backwards checking phase of *gratgen*, which is the only parallelized part, is 3.3.

Finally, the last scatter plot in Fig. 1 compares GRAT against *drat-trim*. Although we compare a verified tool against an unverified one, GRAT wins this comparison: Except for a few certificates, it is faster than *drat-trim*, and there is only a single outlier where GRAT is significantly slower than *drat-trim*.

We also compare the memory consumption: In single threaded mode, *gratgen* needs roughly three times more memory than *drat-trim*, with eight threads, this figure increases to roughly nine times more memory. Due to the garbage collection in Standard ML, we could not measure meaningful memory consumptions for *gratchk*. The extra memory in single-threaded mode is mostly due to the proof being stored in memory, the extra memory in multithreaded mode is due to the duplication of data for each thread.

Finally, Table 2 shows the runtimes of the verified second phase only. Here, *gratchk* is significantly faster than LRAT's verified phase *lrat-check*. In particular, due to the RAT-counts field that is available in GRAT, but not in LRAT (cf. Sect. 3), true RAT lemmas are handled more efficiently. This explains the large discrepancy for the CryptoMiniSat prover.

For completeness, we also report on the satisfiable problems in our benchmark set: The 241 sat certificates in our benchmark set have a size of 566MiB and could be checked in roughly 100 s.

8 Discussion and 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 next step would be to verify the parser, too. Moreover, verification stops at the Isabelle/HOL code generator, whose correctness is only proved on paper [16,17]. There is work on the mechanical verification of code generators [37], and even the subsequent compilers [22]. This technology became available for Isabelle/HOL only recently [20] but does not yet support the imperative arrays required for our application.

If the memory consumption of *gratgen* should become a problem, we could easily write out the proof to disk instead of storing it in RAM. We expect that this would yield a memory consumption similar to *drat-trim*. For multi-threaded mode, we plan 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. A main challenge would be to manage the size of the

enriched certificates, which, without reductions as done by, e. g. backward checking, may become prohibitively large. Moreover, such modifications are probably SAT-solver specific, whereas DRAT certificates are designed to be easily integrated into virtually any CDCL based SAT solver.

An alternative to certification would be to verify the SAT solver itself. While this has been attempted several times (e. g. [33,39]), including our own work [12], we do not expect that verified SAT solvers will become competitive to unverified solvers in the near future.

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

9 Conclusions

We have presented a formally verified SAT solver certification tool. Already in single threaded mode, it is significantly faster than the unverified standard tool *drat-trim*, on a benchmark suite taken from the 2017 and 2016 SAT competitions. 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 by which it is represented.

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. Together with novel optimizations in the first phase, this makes our tool faster than the unverified *drat-trim*. Although most computational work is done in the first phase, optimizing the second phase is important: While *gratchk* was quite efficient from the beginning, the LRAT checker has seen several improvements over time [18]. The initial version was purely functional, and often dominated the runtime of the whole certification process [6].

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-months to develop.

Acknowledgements We thank Maximilian Kirchmeier for proposing and evaluating optimizations for *gratgen* [21]. Moreover, we thank Mathias Fleury and Simon Wimmer 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. Finally, we thank the anonymous reviewers for their useful comments. This work has been supported by the DFG Grant LA 3292/1 "Verifizierte Model Checker" and the VeTSS Grant "Formal Verification of Information Flow Security for Relational Databases".

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Back, R.-J.: On the correctness of refinement steps in program development. Ph.D. thesis, Department of Computer Science, University of Helsinki (1978)
2. Back, R.-J., von Wright, J.: Refinement Calculus—A Systematic Introduction. Springer, Berlin (1998)
3. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, 1st edn. Springer, Berlin (2010)
4. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. In: Proceedings of NFM, pp. 307–321. Springer (2016)
5. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: TPHOL, volume 5170 of LNCS, pp. 134–149. Springer (2008)
6. Cruz-Filipe, L., Heule, M., Hunt, W., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: Proceedings of CADE. Springer (2017)
7. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Proceedings of TACAS, pp. 118–135. Springer (2017)
8. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Proceedings of ICTAC, pp. 260–274. Springer (2010)
9. DRAT-trim homepage. https://www.cs.utexas.edu/~marijn/drat-trim/
10. DRAT-trim issue tracker. https://github.com/marijnheule/drat-trim/issues
11. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: CAV, volume 8044 of LNCS, pp. 463–478. Springer (2013)
12. Fleury, M., Blanchette, J. C., Lammich, P.: A verified SAT solver with watched literals using imperative HOL. In: Proceedings of CPP, pp. 158–171 (2018)
13. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proceedings of DATE, IEEE (2003)
14. Gordon, M.: From LCF to HOL: a short history. In: Plotkin, G., Stirling, C.P., Tofte, M. (eds.) Proof, Language, and Interaction, pp. 169–185. MIT Press, Cambridge (2000)
15. Haftmann, F.: Code generation from specifications in higher order logic. Ph.D. Thesis, Technische Universität München (2009)
16. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Proceedings of ITP, pp. 100–115. Springer (2013)
17. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS 2010, LNCS. Springer (2010)
18. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Proceedings of ITP. Springer (2017)
19. Heule, M., Hunt, W., Wetzler, N.: Trimming while checking clausal proofs. In: 2013 Formal Methods in Computer-Aided Design, FMCAD 2013, pp. 181–188. IEEE (2013)
20. Hupel L., Nipkow T.: A verified compiler from Isabelle/HOL to CakeML. In: Ahmed, A. (ed.) Programming Languages and Systems. ESOP 2018. volume 10801 of LNCS, pp. 999–1026. Springer, Cham (2018)
21. Kirchmeier, M.: Functional implementation of an optimized UNSAT proof-checker. Bachelor's Thesis, Technische Universität München (2017)
22. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Proceedings of POPL, pp. 179–192. ACM (2014)
23. Lammich, P.: Grat tool chain homepage. http://www21.in.tum.de/~lammich/grat/
24. Lammich, P.: Automatic data refinement. In: ITP, volume 7998 of LNCS, pp. 84–99. Springer (2013)
25. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: ITP, volume 8558 of LNCS, pp. 325–340. Springer (2014)
26. Lammich, P.: Refinement to imperative/HOL. In: ITP, volume 9236 of LNCS, pp. 253–269. Springer (2015)
27. Lammich, P.: Refinement based verification of imperative data structures. In: CPP, pp. 27–36. ACM (2016)
28. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: Proceedings of CADE. Springer (2017)
29. Lammich, P.: The GRAT tool chain—efficient (UN)SAT certificate checking with formal correctness guarantees. In: SAT, pp. 457–463 (2017)
30. Lammich, P., Lochbihler, A.: The Isabelle collections framework. In: Proceedings of ITP, volume 6172 of LNCS, pp. 339–354. Springer (2010)
31. Lammich, P., Sefidgar, S.R.: Formalizing the Edmonds–Karp algorithm. In: Proceedings of ITP, pp. 219–234 (2016)

32. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Proceedings of ITP, volume 7406 of LNCS, pp. 166–182. Springer (2012)
33. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. **411**(50), 4333–4356 (2010)
34. Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML. The MIT Press, Cambridge (1997)
35. MLton Standard ML compiler. http://mlton.org/
36. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of DAC, pp. 530–535. ACM (2001)
37. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. J. Funct. Program. **24**(2–3), 284–315 (2014)
38. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS. Springer, Berlin (2002)
39. Oe, D., Stump, A., Oliver, C., Clancy, K.: `versat`: a verified modern SAT solver. In: VMCAI, volume 7148 of LNCS, pp. 363–378. Springer (2012)
40. SAT competition, 2013. http://satcompetition.org/2013/
41. SAT competition, 2014. http://satcompetition.org/2014/
42. SAT competition, 2016. http://baldur.iti.kit.edu/sat-competition-2016/
43. SAT competition, 2017. https://baldur.iti.kit.edu/sat-competition-2017/
44. Sinz, C., Biere, A.: Extended resolution proofs for conjoining BDDs. In Proceedings of CSR, pp. 600–611. Springer (2006)
45. Wetzler, N., Heule, M. J. H., Hunt, W. A.: Mechanical verification of SAT refutations with extended resolution. In: Proceedings of ITP, pp. 229–244. Springer (2013)
46. Wetzler, N., Heule, M. J. H., Hunt, W. A.: Drat-trim: efficient checking and trimming using expressive clausal proofs. In: Proceedings of SAT 2014, pp. 422–429. Springer (2014)
47. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. volume 10805 of LNCS, pp. 61–78. Springer, Cham (2018)
48. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.