DRAT Proofs Without Deletions of Unique Reason Clauses & Complete and Efficient DRAT Proof-Checking

Johannes Altmanninger

September 18, 2019

Abstract: Clausal proof format DRAT is the de facto standard way to certify SAT solvers' unsatisfiability results. State-of-the-art DRAT proof checkers ignore deletions of unit clauses, which means that they are checking against a proof system that differs from the specification of DRAT and they cannot verify inprocessing techniques that use unit deletions. State-of-the-art SAT solvers produce proofs that are accepted by DRAT checkers, but are incorrect under the DRAT specification, because they contain spurious deletions of reason clauses. We present patches for award-winning SAT solvers to produce correct proofs with respect to the specification. Performing unit deletions in a proof checker can be computationally expensive. We implemented a competitive checker that honors unit deletions and provide experimental results that on average checking costs are the same as when not doing unit deletions. As it is also expensive to determine the (in-)correctness of a proof we present the SICK format which describes small and efficiently checkable certificates of the incorrectness of a DRAT proof. This increases trust in incorrectness results and can be useful when debugging solvers and checkers.

Contents

1. Introduction

Over the past decades, there has been significant progress in SAT solving technology. SAT solvers have had documented bugs [1] [2]. To detect incorrect results, there are *checker* programs that *verify* a solver's result based on a witness given by the solver. Satisfiability witnesses, or *models* are trivial to check in linear time. Unsatisfiability witnesses, or proofs of unsatisfiability on the other hand can be much more costly to check.

A SAT solver operates on a formula that acts as knowledge base. It contains constraints that are called clauses. Starting from the input formula, clauses are added and deleted by the solver. In SAT competitions, solvers are required to give proofs of unsatisfiability in the DRAT proof format [3]. A DRAT proof is the trace of a solver run, containing information on which clauses are added and deleted.

State-of-the-art proof checkers ignore deletions of unit clauses [4]. As a result, the checkers are not faithful to the specification of DRAT proofs. The original definition of the proof format is referred to as *specified* DRAT and the one that is implemented by state-of-the-art checkers as *operational* DRAT [4]. The classes of proofs verified by checkers of these two flavors of DRAT are incomparable.

State-of-the-art solvers produce proofs with deletions of specific reason clauses, which are the reason for the discrepancy between specified and operational DRAT. We did not see a motivation for current solvers delete those reason clauses, so we investigated why they do so. We found that the solvers that produce proofs such deletions do not undo inferences made using those reason clauses. Perhaps because of this mismatch, proof checkers ignore deletions of unit clauses (which includes those reason clauses), matching the solvers' internal behavior. We provide patches for award-winning solvers to make them generate proofs without spurious deletions of reason clauses, eliminating the need to ignore some deletion instructions.

DRAT proofs are designed to use minimal space per proof step but checking them is computationally expensive. In theory, checking costs are comparable to solving [3]. Consider the problem of the Schur Number Five, where solving took just over 14 CPU years whereas running the DRAT checker on the resulting proof took 20.5 CPU years [5]. Additionally, state-of-the-art SAT solvers use complex inprocessing techniques to modify a formula. When DRAT proof logging is desired, such a modification needs to be expressed by a DRAT proof consisting of clause introductions and clause deletions. If such a proof uses deletions of reason clauses then a checker for specified DRAT can be necessary to verify the inprocessing step [4] because a checker for operational DRAT would ignore the deletions which can cause future proof steps to be incorrectly rejected. The absence of an efficient checker for specified DRAT means that solvers cannot use such techniques in competitions. There is an efficient algorithm to check specified DRAT [6] that features several optimizations that are not necessary for checking operational DRAT. Previous implementations were not competitive. Our research question is whether it is possible to check specified DRAT as efficiently as operational DRAT.

We implemented a checker for specified DRAT with state-of-the-art performance. Our experimental results suggest that specified and operational DRAT are equally expensive to check on the average real-world instance. We also observe that a high number of reason deletions tends to have a significant negative impact on checking performance.

The majority of solvers at SAT competitions produce proofs that are incorrect under specified DRAT. For those proofs, our checker outputs a small, efficiently checkable incorrectness certificate in the SICK format which was originally developed along with the first checker for specified DRAT[1] but has not been published. The incorrectness certificate can be used to check the incorrectness of a proof independently of the checker, which helps developers debug proof-generation and proof-checking algorithms. While checking operational DRAT efficiently is arguably easier than specified DRAT, the straighforward semantics of specified DRAT facilitates reasoning about a proof, e.g. it allows the definition of the SICK format to be much simpler. We contribute an extension to the SICK format to support a slightly different semantics of DRAT checking.

This thesis is organized as follows: In the following section we will introduce preliminary knowledge about SAT solving, proofs of unsatisfiability and proof checking, including the optimization challenges of specified DRAT checking. Our first contribution, a proposal on how to change solvers to produce unambiguously correct proofs, can be found in Section 3. Section 4 concerns the efficient implementation of a specified DRAT checker: after briefly discussing other checkers we present our implementation and describe the SICK format for incorrectness certificates. Experimental results evaluating checker performance are given in Section 5. Finally, we draw a conclusion in Section 6 and give outlook on future work in the area of DRAT proof-checking in the last section.

[1] https://github.com/arpj-rebola/rupee

2. Preliminaries

A *literal* is a propositional variable, like x, or a negation of a variable, denoted by $\overline{x}$. A *clause* is a disjunction of literals, usually denoted by juxtaposition of the disjuncts, e.g. we write $xy\overline{z}$ for $x \vee y \vee \overline{z}$.

An *assignment* is a finite, set of literals. All literals in an assignment are considered to be *satisfied* by that assignment. Conversely, the complements of those literals are *falsified* by that assignment. Other literals are *unassigned*.

SAT solvers work on formulas in *conjunctive normal form* (CNF), conjunctions (or multisets) of clauses. A clause is satisfied by an assignment I if any literal in the clause is satisfied by I. A formula in CNF is satisfied by I if each of its clauses is satisfied by I. An assignment that satisfies a formula is called a *model* for that formula. A formula is *satisfiable* if there exists a model for it. Two formulas F and G are *satisfiability-equivalent* if F is satisfiable if and only G is satisfiable.

A *unit clause* with respect to some assignment contains only falsified literals except for a single non-falsified *unit literal*.

Unit Propagation Let F be a CNF formula. We say that a literal l is implied by *unit propagation* over F whenever there is a finite *propagation sequence* of clauses $(C_1, \ldots, C_n) \subseteq F$ such that for each $1 \leq i \leq n$ there is a literal $l_i \in C_i$ with $C_i \setminus \{l_i\} \subseteq \{\overline{l_1}, \ldots, \overline{l_{i-1}}\}$. We call C_i the *reason clause* for l_i with respect to this propagation sequence. Clause C is called the *unique reason clause* for literal l with respect to F if it is the reason clause for l in some propagation sequence in F and there is no propagation sequence in F where another clause is the reason for l. The *shared UP-model* is the assignment consisting of all literals implied by unit propagation [4] in the formula.

2.1 SAT Solving

A SAT solver takes as input a formula and finds a model if the formula is satisfiable. Otherwise, the solver provides a proof that the formula is unsatisfiable. While searching for a model, a solver maintains an assignment along with the order in which the literals were assigned. We call this data structure the *trail*. SAT solvers search through the space of all possible assignments. They make *assumptions*, virtually adding clauses of size one to the formula temporarily. This triggers unit propagation, adding more literals to the trail. These literals are logically implied by the formula with the current assumptions. Assignments that falsify the literals in the trail are pruned from the search space. Additionally, solvers may use inprocessing techniques to modify the formula. Once the trail falsifies a clause, the solver has derived the unsatisfiable empty clause and therefore established unsatisfiability of the current formula plus assumptions. If there are assumptions, some of them are undone and the solver resumes search. Otherwise the input formula is unsatisfiable.

Efficient Implementation of Unit Propagation To efficiently keep track of which clauses can become unit, competitive solvers and checkers use the two-watched-literal scheme [7]. It consists of a watchlist for each literal in the formula, which is a list of clause references. Clauses in the watchlist of some literal are said to be *watched on* that literal. Each clause is watched on two literals, which are also called its *watches*. Provided that Invariant 1 from [6] is maintained, it suffices to look at the watches to determine that a clause is not unit.

Invariant 1. If a clause is watched on two distinct literals l and k, and the current trail I falsifies l, then I satisfies k.

A clause that is not already satisfied, can only become unit if one of it is watches is falsified. When literal l is assigned, it is sufficient to visit clauses in the watchlist of $\bar{l}$ to find unit literals that are not yet assigned. For clauses that are not unit, the watches are changed to restore Invariant 1.

As an example, we perform unit propagation on formula $F = \{x, \overline{y}\overline{x}z, \overline{x}y, xy\}$. Only the first two literals in each clause are watched, so only $\overline{y}$ and x are watched in the second clause. Initially only the size-one clause x is unit. Two clauses are watched on $\overline{x}$, so they need to be inspected: Invariant 1 is violated in $\overline{y}\overline{x}z$, so the watches will be shuffled, making the clause $\overline{y}z\overline{x}$. Additionally, $\overline{x}y$ is unit, which triggers propagation of y. Only $\overline{y}z\overline{x}$ is watched on $\overline{y}$, which assigns unit z and causes no further propagation. Clause xy is never visited during propagation.

CDCL Predominant SAT solvers implement Conflict Driven Clause Learning (CDCL) [8] which is based on the following principle: whenever some clause in the formula is falsified with respect to the trail, the current set of assumptions makes the formula unsatisfiable. Therefore a subset of the assumptions is undone and a *conflict clause* is learned — it is added to the formula to prevent the solver from revisiting those wrong assumptions. As the number of clauses increases, so does memory usage, and the time spent on unit propagation. Because of this, learned clauses are regularly deleted from the formula in a satisfiability-preserving way if they are not considered useful.

2.2 Proofs of SAT Solvers' Unsatisfiability Results

Redundancy Criteria A clause C is redundant in a formula F if F and $F \cup \{C\}$ are satisfiability equivalent [9]. There are various criteria of redundancy, with different levels of expressivity and computational costs.

1. *RUP*: a clause C is a *reverse unit propagation* (RUP) inference in formula F if the shared UP-model of $F' := F \cup \{\bar{l} \mid l \in C\}$ falsifies a clause in F [10]. To compute whether C is RUP, the negated literals in C are added as assumptions and propagated until a clause is falsified by the trail. A clause that is RUP F is logical consequence of F [10].
2. *RAT* a clause C is a *resolution asymmetric tautology* (RAT) [11] on some literal $l \in C$ with respect to formula F whenever for all clauses $D \in F$ where $\bar{l} \in D$, the resolvent on l of C and D, which is $(C \setminus \{l\}) \cup (D \setminus \{\bar{l}\})$ is a RUP in F. Clause D is called a *resolution candidate* for C and l is called the *pivot*. Computing whether a clause is RAT can be done with one RUP check for each resolution candidate.

DRAT Proofs Proofs based on RUP alone are not expressive enough to simulate all inprocessing techniques in state-of-the-art SAT solvers [12]. Because of this, the more powerful criterion RAT is used today [12]. A DRAT proof (*delete resolution asymmetric tautology*) [13] is a sequence of lemmas (clause introductions) and deletions, which can be applied to a formula to simulate the clause introductions, clause deletions and inprocessing steps that the solver performed. The *accumulated formula* at each proof step is the result of applying all prior proof steps to the input formula. Based on the accumulated formula, the checker can compute the shared UP-model at each step which eventually falsifies some clause, just like in the solver. Every lemma in a correct DRAT proof is a RUP or RAT inference with respect to the accumulated formula. In practice, most lemmas are RUP inferences, so a checker first tries to check RUP and only if that fails, falls back to RAT.

In *specified DRAT*, checking is performed with respect to the accumulated formula, while *operational DRAT* uses an *adjusted accumulated formula* that is computed the same way as the accumulated formula except that deletions of clauses that are unit with respect to the shared UP-model are ignored [4].

A proof solely consisting of clause introductions will result in the checker's propagation routines slowing down due to the huge number of clauses just like in a CDCL solver that does not delete learned clauses. To counteract this, clause deletion information has been added, making the proof-checking time comparable to solving time [3] [13]. While deletions were added as an optimization, they can also enable additional inferences due to RAT being non-monotonic [14]. This means that ignoring deletions may prevent RAT inferences which is why a proof that is correct under specified DRAT may be incorrect under operational DRAT.

LRAT Proofs The runtime and memory usage of DRAT checkers can exceed the ones of the solver that produced the proof [5]. The resulting need for a DRAT checker to be as efficient as possible requires mutable data structures that rely on pointer indirection which are difficult to verify. The lack of a formally verified DRAT checker is remedied by making the DRAT checker output an annotated proof in the LRAT format [15]. The LRAT proof can be checked by a formally verified checker[2] without unit propagation, making sure that the formula formula is indeed unsatisfiable. Most solvers can only generate DRAT proofs but DRAT checkers can be used to produce an LRAT proof from a DRAT proof. The LRAT proof resembles DRAT, but it includes clause hints to aid propagation: for each resolution candidate, it contains a propagation sequence that falsifies some clause to show that the resolvent is a RUP inference.

2.3 Proof Checking

We say that some tool *verifies* some property of an artifact when that artifact has this property according to the semantics encoded in the tool. This is not to be confused with formal verification; for our checkers we have no proof that the implementation corresponds to any formal specification.

The naïve way to verify that a proof is correct consists of performing each instruction in the proof from the first to the last one, while checking each lemma.

Backwards Checking During search, SAT solvers cannot know which learned clauses are useful in a proof, so they add all of them as lemmas. This means that many lemmas might not be necessary in a proof of unsatisfiability. *Backwards checking* [16] avoids checking superfluous lemmas by only checking *core* lemmas — lemmas that are part of the *unsatisfiable core*, an unsatisfiable subset of an unsatisfiable formula. Starting from the falsified clause, the proof is traversed backwards. Initially, the falsified clause is added to the core. Each core lemma is checked. Every lemma that is used in a propagation sequence to derive some other lemma is added to the core as well. Clauses and lemmas that are not in the core do not influence the unsatisfiability result and are virtually dropped from the proof. Tools like `drat-trim` can output a *trimmed* proof that only contains core lemmas in DRAT or LRAT format.

To check an inference, the checker needs to compute the shared UP-model of the accumulated formula. This is stored in the trail. Instead of recomputing the shared UP-model from scratch at each proof step, the trail is modified incrementally: whenever a clause is added to the formula, the propagation routine adds the missing literals to the trail. When a reason clause is deleted, some literals may be removed.

Backwards checking can be implemented using two passes over the proof: a forward pass that merely performs unit propagation after each proof step [6] until

[2] https://github.com/acl2/acl2/

some clause is falsified, and a backward pass that checks lemmas as required. The forward pass enables the checker to record the state of the trail at each proof step and efficiently restore it during the backward pass. If there are no deletions of unique reason clauses, the shared UP-model grows monotonically, and the trail can be restored by simply truncating it.

Here is a small example for backwards checking: let $F = \{x, \overline{x}\}$ be an unsatisfiable formula with a proof with two lemmas ($\mathbf{add}\ y, \mathbf{add}\ \square$), where $\square$ is the empty clause. For simplicity, we assume that checking is not short-cut as soon as a clause is falsified. The forward pass applies both lemmas in the proof, which makes the accumulated formula $\{x, \overline{x}, y, \square\}$ and its shared UP-model is $\{x, \overline{x}, y\}$. The first step is to check that $\square$ is a RUP inference. This is done by removing it from the formula, and then performing the RUP check: the reverse literals in $\square$ are added as assumptions and propagated. This is a no-op here since the $\square$ contains no literals. After these propagations, to show that $\square$ is RUP we need to find a clause that is falsified by the trail. In fact, there are two clause: x and $\overline{x}$, so we know that $\square$ is a RUP inference on the formula $\{x, \overline{x}, y\}$. Then we need to perform conflict analysis to add to the core a falsified clause, and all clauses that are reasons for falsifying that clause. Assuming we pick $\overline{x}$ as falsified clause, then the reason for falsifying $\overline{x}$ is the clause x. Hence the core is $\{\overline{x}, x\}$. As y is not in the core, we know that it did not contribute to a RUP inference at any step after its introduction. Therefore it is virtually deleted from the proof, and the redundancy check is not performed.

Core-first Unit Propagation To keep the core small and reduce checking costs, core-first unit propagation was introduced [16]. It works by doing unit propagation in two phases:

1. Propagate using clauses already in the core.
2. Examine non-core clauses:
   - If there is some unit clause with an unassigned unit literal, propagate that and go to 1.
   - Otherwise terminate.

This results in a minimum of clauses being added to the core because if a falsified clause can be found without adding another clause to the core it will always be found. This usually seems to make checking faster, probably because unit propagation is mostly done in the core instead of all clauses.

Consider a similar example as above: let $F = \{x, \overline{x}, \overline{x}y\}$ be an unsatisfiable formula with proof ($\mathbf{add}\ xy, \mathbf{add}\ \square$). To make such a small example possible we assume that, even though backwards checking is used, all lemmas are checked and not only core lemmas. As above, after performing a successful RUP check of $\square$, the core has two clauses $\{x, \overline{x}\}$. For checking xy, the reverse literals $\overline{x}$ and $\overline{y}$ are added to the formula as assumptions and propagated. Then the trail contains $\{x, \overline{x}, \overline{y}\}$. This means that the clause $\overline{x}y$ in the formula is falsified and could be added to the core. However, with core-first propagation the first falsified clause will always be found using only clauses that are already in the core if possible, so in this case it will choose x or $\overline{x}$.

Reason Deletions Under operational DRAT unit deletions are ignored. Only proofs with unique reason deletions have different semantics under specified and operational DRAT. To detect unique reason deletions, it is necessary to implement specified DRAT, as it is to verify inprocessing steps with reason deletions, A reason is a a clause that was used in some propagation sequence to compute a literal l in the trail. When a unique reason clause is deleted, l is no longer implied by unit propagation and needs to be removed from the trail. This means that it is not possible anymore to revert this modification of the trail in the backward pass by truncating the trail. Instead, for each reason deletion, the removed literals are recorded by the checker, along with their positions in the trail and their reasons. This information can be used in the backward pass to restore the state of the trail to be exactly the same as in the forward pass for each proof step, which is what the algorithm from [6] does along other non-trivial techniques to maintain the watch invariants.

3. DRAT Proofs without Deletions of Unique Reason Clauses

Some state-of-the-art solvers produce proofs with deletions of unique reason clauses. A significant fraction of their proofs are incorrect under specified DRAT. Since these solvers act as if reason clauses were not deleted we propose patches to avoid deletions of unique reason clauses, matching the solver's internal behavior. For the fragment of proofs without unique reason deletions, operational and specified DRAT coincide because the accumulated formula and the adjusted accumulated formula coincide, hence these proofs can be checked with a checker of either flavor.

Out of the solvers submitted to the main track of the 2018 SAT competition, the ones based on `MiniSat` and `CryptoMiniSat` produce proofs with deletions of unique reasons while, to the best of our knowledge, others do not.

Let us explain how `DRUPMiniSat`[3] emits unique reason deletions. This solver performs simplification of the formula when there are no assumptions (decision-level 0), so the trail is equivalent to the shared UP-model of the formula without assumptions. The literals in the trail at this point are never unassigned by `DRUPMiniSat`.

On step in the simplification phase is the method `Solver::removeSatisfied`, which for each clause C that is satisfied by the shared UP-model, removes C from the clause database and emits a deletion of C to the DRAT proof output. Such a clause C remains satisfied indefinitely for the rest of the search because it is already satisfied by some literal that will never be unassigned as stated above. For example, consider the formula $F = \{x, xy\}$. The shared UP-model is $\{x\}$ and clause x is the reason for literal x. Because both clauses are satisifed by the shared UP-model, the solver would remove them and add deletions of x and xy to the proof.

[3]The original patch to `MiniSat` to produce DRUP/DRAT proofs on which other solvers' proof generation procedures seem to be based. See https://www.cs.utexas.edu/~marijn/drup/

In MiniSat, *locked* clauses are reason clauses, the reason for having propagated some literal in the trail. The method Solver::removeSatisfied also deletes locked clauses, however, the literals assigned because of such a locked clause will not be unassigned. This suggests that the locked clause is implicitly kept in the formula, even though it is deleted. State-of-the-art DRAT checkers ignore deletions of unit clauses, which means that they do not unassign any literal when deleting clauses, matching DRUPMiniSat's behavior. In above example, after deleting both clauses from F, the unique reason clause for literal x is gone. Therefore the shared UP-model does not contain literal x anymore.

We propose two possible changes to make DRUPMiniSat produce proofs that do not require ignoring unit deletions when checking.

1. Do not remove locked clauses during simplification. In our example, this would mean that x is not deleted, so shared UP-model stays the same.
2. Before removing locked clauses, emit the corresponding propagated literal as addition of a unit clause in the DRAT proof. Suggested by Mate Soos[4], this option is also the preferred one to the authors of mergesat[5] and varisat[6]. Additionally, this is implemented in CaDiCaL's[7] preprocessor. This does not influence the correctness of future inferences because the unit clause that is added and the reason clause that is removed are equivalent, so in conjunction they preserve the equivalence of the formula. The added and removed clauses are equivalent because under the shared UP-model they are the same clause. The literals of the shared UP-model will never be unassigned throughout the solver's runtime because this is done during the simplification phase at decision level zero. In our example this means that another instance of x is added, before one x is deleted, which preserves the formula.

We provide patches implementing these for MiniSat version 2.2 (1.[8] and 2.[9]), and the winner of the main track of the 2018 SAT competition (1.[10] and 2.[11]). Both patches are arguably very simple and we do not expect any significant impacts in terms of solver runtime, memory usage or proof size: the additional clauses will not be added to the watchlists and do therefore not slow down propagation. There can be at most one locked clause per variable, so their memory usage is small. The proof will be larger only with the second variant by adding unit clause additions and deletions, also at most one each per variable, which is small compared to the rest of a proof. The patches can be easily adapted to other DRUPMiniSat-based solvers.

[4]https://github.com/msoos/cryptominisat/issues/554#issuecomment-496292652

[5]https://github.com/conp-solutions/mergesat/pull/22/

[6]https://github.com/jix/varisat/pull/66/

[7]http://fmv.jku.at/cadical/

[8]https://github.com/krobelus/minisat/commit/keep-locked-clauses/

[9]https://github.com/krobelus/minisat/commit/add-unit-before-deleting-locked-clause/

[10]https://github.com/krobelus/MapleLCMDistChronoBT/commit/keep-locked-clauses/

[11]https://github.com/krobelus/MapleLCMDistChronoBT/commit/add-unit-before-deleting-locked-clause/

4. Complete and Efficient DRAT Proof-Checking

We implement a checker to compare the costs of checking specified and operation DRAT. Additionally an efficient checker for specified DRAT can be useful to verify solvers' inprocessing steps that contain unit deletions. In this section, we discuss our checker implementation after introducing existing checkers. Finally we describe the format for SICK witnesses which can be produced by our checker to certify the rejection.

4.1 Existing Checkers

We heavily draw upon existing checkers. In fact, our implementation contains no algorithmic novelties but merely combines the ideas present in existing checkers.

`DRAT-trim` The seminal reference implementation; Marijn Heule's `DRAT-trim` can produce a trimmed proof in the DRAT or LRAT format. We mimic their way of producing LRAT proofs and ensure that all our proofs are verified by the formally verified checker[12]. This gives us confidence in the correctness of our implementation and allows for a comparison of our checker with `DRAT-trim` since both have the same input and output formats.

`DRAT-trim` pioneered deletions, backwards checking and core-first propagation. Additionally it employs an optimization which we also use: during RAT checks, resolution candidates that are not in the core are ignored, because the proof can be rewritten to delete them immediately before the current lemma. Let l be the pivot literal and D a non-core clause that is a resolution candidate, so $\bar{l} \in D$. During the backwards pass, a RAT check is performed using l as pivot. Since D is not in the core, it was never used in a **later** inference since we are checking lemmas from last to first. By ignoring D as RAT candidate it is virtually removed from the proof. This is sound, that is, a correct RAT inference on pivot l does not depend on the clause D, so it can be freely removed. This

[12] https://github.com/acl2/acl2/

is the case because in the RAT check, the resolution candidate becomes unit after propagating the reverse literals in resolvent, so unit literal $\bar{l}$ is satisfied, or rather l is falsified. This makes D a tautology which will never be used to derive a conflict and thus make an inference[13].

GRAT Toolchain More recently, Peter Lammich has published the GRAT toolchain [17] that is able to outperform `DRAT-trim` [18].

They first produce a GRAT proof which is similar to LRAT with the `gratgen` tool, after which formally verified `gratchk` can be used to check the certificate, guaranteeing that the original formula is indeed unsatisfiable. We also implement GRAT generation in our tool. However, the `gratchk` tool ignores unit deletions, so GRAT proofs are only useful for operational DRAT.

They introduce two optimizations:

1. Separate watchlists for core and non-core clauses[14]. This speeds up core-first unit propagation, so we use it in our implementation. The clauses in the core are kept in different watchlists than non-core clauses. Since most time is spent in propagation, this can give a significant speed-up to core-first propagation. Without separate watchlists, core-first propagation traverses the same watchlists twice, first in core-mode and then in non-core-mode. For each visited clause they check if it is in the core and propagate if that matches the current mode. This branch can be moved outside the loop by partitioning the watchlists into core and non-core clauses.
2. Resolution candidate caching / RAT run heuristic [18]: DRAT proofs tend to contain sequences of RAT lemmas with the same pivot, in which case they only compute the list of RAT candidates once per sequence and reuse it for all lemmas with the same pivot. We did not implement that since we don't have benchmarks with a significant number of RAT introductions compared to the number of RUP introductions.

Among state-of-the-art DRAT checkers, `gratgen` is arguably the easiest to understand (even though can do a parallel checking), so we advise interested readers to study that.

`rupee` This is the original implementation[15] of the algorithm to honor unique reason deletions. We use the same algorithm. During our research we found one issue in the implementation which was fixed[16].

[13]http://www21.in.tum.de/~lammich/grat/gratgen-doc/Unmarked_RAT_Candidates.html

[14]http://www21.in.tum.de/~lammich/grat/gratgen-doc/Core_First_Unit_Propagation.html

[15]https://github.com/arpj-rebola/rupee

[16]https://github.com/arpj-rebola/rupee/compare/b00351cbd3173d329ea183e08c3283c6d86d18a1..b00351cbd3173d329ea183e08c3283c6d86d18a1~~~

In previous experiments, `rupee` was an order of magnitude slower than `DRAT-trim` [6]. We believe that this overhead is primarily not a consequence of algorithmic differences but of implementation details such as parsing[17] and missing function inlining. Additionally, `rupee` does not use core-first unit propagation while the other checkers do.

4.2 Checker Implementation

Our checker is called `rate`[18]. It is a drop-in replacement for a subset of `drat-trim`'s functionality — the unsatisfiability check with core extraction — with the important difference that it checks specified DRAT by default. When a proof is verified, `rate` can output core lemmas as DIMACS, LRAT or GRAT. Otherwise, the rejection of a proof can be supplemented by a SICK certificate of incorrectness. The representation of the DRAT proof — binary or textual – is automatically detected the same way as `drat-trim`. Additionally, compressed input files (Gzip, Zstandard, Bzip2, XZ, LZ4) are transparently uncompressed.

We provide two options that alter the semantics of the checker:

1. Unit deletions can be skipped with the flag `-d`. This makes `rate` check operational DRAT instead of specified DRAT.
2. If the flag `--assume-pivot-is-first` is given, the pivot must be the first literal in a RAT lemma, otherwise the proof will be rejected.

Among other metrics, `rate` can output the number of reason deletions and unique reason deletions[19]. Other checkers cannot provide the latter. This might be useful to validate that a SAT solver's proof contains no unique reason deletion, since we are not aware of a good reason why proofs by current solvers should contain such deletions (apart from some inprocessing techniques).

As other state-of-the-art checkers, `rate` actually deviates from the specification of DRAT where it is convenient or necessary for competitive performance. For example we permit sloppy input, and fail when given 2^{30} or more clauses.

We also support a more powerful clausal proof format, DPR (*delete propagation redundancy*) [9].

To automatically minimize inputs that expose bugs in our checker we have developed a set of scripts to delta-debug CNF and DRAT instances.

Rust We chose the modern systems programming language Rust[20] for our implementation because of its feature parity with C in the domain of SAT solving.

[17] Both `rupee` and `DRAT-trim` use a fixed-size hash table to locate deleted clauses but `rupee`'s is smaller by one order of magnitude, which may explain parts of the difference in performance.

[18] https://github.com/krobelus/rate

[19] The metric for the number of unique reason deletions is called **`reason deletions shrinking trail`** in the output of `rate`.

[20] https://www.rust-lang.org/

Among the respondents of the 2019 Stack Overflow Developer Survey[21] it is the most loved programming language and Rust developers have the highest contribution rate to open source projects.

Based on our experience, we believe that it is a viable alternative to C or C++ for SAT solving, assuming people are willing to learn the language. The first Rust-based solver to take part in competitions `varisat`[22] is a great example of this. They use a library that implements a missing language feature, adding convenience to the type system that is sometimes inflexible due to Rust's borrow checker.[23]

Rust aims to avoid any undefined behavior. For example, buffer overflows are prevented by performing runtime bounds checks upon array access. While for most programs those bounds checks have negligible impact on performance (branch prediction can handle them seamlessly), we removed bounds checking by default, which gave speedups of around 15% in preliminary tests. Furthermore, our checker implementation contains a variety of cheap runtime assertions, including checks for arithmetic overflows and narrowing conversions that cause a change of value.

4.3. SICK Format

For DRAT proofs that are verified by `rate`, it can produce an LRAT proof containing core lemmas. The formally verified LRAT checker can be used to certify that the LRAT proof is a correct proof of unsatisfiability, which suggests that the original DRAT proof is correct as well. However, many proofs are rejected by our checker for specified DRAT. To trust those results, we want to verify the incorrectness of proofs.

A proof is incorrect if any of its lemmas is not a RUP or RAT inference. To show that a lemma C is not RUP in the accumulated formula F, it suffices to show that the shared UP-model of $F \cup \{\bar{l} \,|\, l \in C\}$ does not falsify any clause in F. On top of that, to show that C is not RAT, it suffices to show that any resolvent with C is not RUP.

Since our checker already computes the shared UP-models for the RUP checks, we can output that and check the inference with an independent tool. This tool can be much simpler than the checker because it does not need to implement unit propagation. This is useful because unit propagation with watch lists is non-trivial to implement correctly for specified DRAT. A bug in watch list implementation often cause problems when it causes some clauses to not be watched when they are unit which means that the propagation may be incomplete. If so, the shared UP-model computed by the checker is smaller than the actual shared UP-model. This can be detected easily by checking each individual clause

[21] https://insights.stackoverflow.com/survey/2019
[22] https://github.com/jix/varisat/
[23] https://jix.one/introducing-partial_ref/

if it is a unit that is not yet assigned. On the other hand, if the checker's shared UP-model is bigger than the actual shared UP-model, this is a bug that will not easily be detected by such a tool.

The format we use for the incorrectness certificates is called *SICK*. It was originally developed for `rupee`. A certificate in our SICK format can be used by our tool `sick-check` to verify incorrectness of the proof without doing any unit propagation. Furthermore, the incorrectness certificate is tiny compared to the formula. We have fixed some bugs in our checker that were exposed by `sick-check`. The SICK file format is using TOML[24] syntax. See Figure 1 for a grammar. An example of a SICK certificate is in Figure 2. The first two columns show a satisfiable formula with two binary clauses in DIMACS format and an incorrect DRAT proof for this formula. The proof consists of two lemmas, a size-one clause, and the empty clause. The third column shows the corresponding SICK certificate, stating that the RAT check failed for the first lemma in the proof.

SICK	:=	ProofFormat ProofStep NaturalModel Witness*
ProofFormat	:=	`proof_format` = (`"DRAT-arbitrary-pivot"` | `"DRAT-pivot-is-first-literal"`)
ProofStep	:=	`proof_step` = Integer
NaturalModel	:=	`natural_model` = ListOfLiterals
Witness	:=	`[[witness]]` FailingClause FailingModel Pivot
FailingClause	:=	`failing_clause` = ListOfLiterals
FailingModel	:=	`failing_model` = ListOfLiterals
Pivot	:=	`pivot` = Literal
ListOfLiterals	:=	`[` (Literal `,`)* `]`

Figure 1: The grammar of a SICK certificate

Formula	Proof	SICK Certificate
`p cnf 2 2`	`1 0`	`proof_format   = "DRAT-arbitrary-pivot"`
`-1 -2 0`	`0`	`proof_step     = 1`
`-1  2 0`		`natural_model  = [-1, ]`
		`[[witness]]`
		`failing_clause = [-2, -1, ]`
		`failing_model  = [2, ]`
		`pivot          = 1`

Figure 2: Example SICK certificate for an incorrect proof

Explanation

- `proof_step` specifies the proof step that failed (by offset in the proof, starting at one for the first proof step). For the remainder of this section,

[24] https://github.com/toml-lang/toml

let the *lemma* denote the clause that is introduced by the referenced proof step. For a textual proof that has each proof step on a separate line, this corresponds to the line number of the introduction instruction that failed.

- `proof_format` describes the proof format to use. We added the distinction between these two formats because it was not clear which one should be used exclusively.

  - `DRAT-arbitrary-pivot`: DRAT checking where the pivot can be any literal in the lemma. This requires one witness (counter-example) for each possible pivot in the lemma. The pivot has to be specified for each witness.
  - `DRAT-pivot-is-first-literal`: Similar, but there is only one witness. The pivot needs to be the first literal in the lemma.

  Not all current solvers put the pivot as first literal of a RAT lemma, therefore in practise `DRAT-arbitrary-pivot` is usually desired. New proof formats such as PR however require explicitly specifying the witness.

- `natural_model` gives the shared UP-model before checking this proof step.

Each witness is a counter-example to some redundancy check.

- `failing-clause`: A clause in the formula, which is a resolution candidate for the lemma. This means that the RUP check failed for the resolvent on the pivot literal of the lemma and the failing clause.
- `failing-model`: The literals that were added to the natural model (trail) when performing the failed redundancy check.
- `pivot`: This specifies the pivot literal.

The absence of a witness means that a RUP check failed. If the lemma is the empty clause, a witness is never needed, since the empty clause cannot be RAT.

Semantics Our tool `sick-check` verifies SICK certificates that fulfill below properties.

Let F be the accumulated formula up to and excluding the lemma.

1. The proof contains the `proof_step`.
2. The given `natural_model` is the shared UP-model of F.
3. For format `DRAT-arbitrary-pivot`, the pivots are identical to the literals in the lemma.
4. For each witness, consisting of `failing_clause`, `failing_model` and `pivot`.
   1. The `failing_clause` is in F.
   2. The union of `natural_model` and `failing_model` is the shared UP-model of $F \cup \{\bar{l} \mid l \in r\}$ where r is the resolvent on `pivot` of the lemma and the `failing_clause`.

A SICK certificate can only be produced by checker of specified DRAT, because to compute the accumulated formula in an operational checker, one would need to do unit propagation which is avoided by design in the SICK checker. This is a potential benefit of a specified checker: the accumulated formula at each proof step can be computed without unit propagation.

Contribution Our contribution to the SICK format consists of the design of a this new syntax that takes into account the different variants of DRAT.

5. Experimental Evaluation

Here we present a performance evaluation of our checker. Technical details are available[25].

We compare the performance of four checkers:

1. `rate`
2. `rate-d` (the flag `-d` means *"skip unit deletions"*)
3. `drat-trim`
4. `gratgen`

Only `rate` checks specified DRAT, the other three implement operational DRAT.

Benchmark Selection Each individual benchmark consists of a SAT problem instance and a solver to produce the proof for this instance. We take from the 2018 SAT competition[26] both the SAT instances and the solvers from the main track. We exclude benchmarks that are not interesting for our purpose of evaluating `rate's` performance: firstly we discard all benchmarks where the instance is satisfiable according to the competition results[27]. Secondly we discard the benchmarks where the solver timed out in the competition because these runs will probably time out in our experiments as well. For the remaining benchmarks we have have run the solver to generate the proof. Some solvers would time out, failing to generate a proof. As a final measure to include only interesting benchmarks, we exclude all proofs that are rejected by `rate`. This ensures a fair comparison in terms of checker performance: when `rate` rejects a proof it exits as soon as an incorrect instruction is encountered in the backward pass. This means that it processed only a fraction of the proof while other checkers would process the entire proof. Hence it is not useful for benchmarking checker performance to include proofs that are rejected under specified DRAT.

In total we analyze 39 solvers and 120 unsatisfiable instances, making for over 4000 potential solver-instances pairs as benchmarks. Roughly half of the instances are satisfiable, and more than half of the proofs are rejected by `rate`, so as a

[25] https://github.com/krobelus/rate-experiments
[26] http://sat2018.forsyte.tuwien.ac.at/
[27] http://sat2018.forsyte.tuwien.ac.at/results/main.csv

result of above steps discarding benchmarks that are not relevant for our purpose, we are left with 810 benchmarks where the proof is verified by all checkers.

Experimental Setup We ran all checkers on the selected benchmarks. For `rate`, `rate-d` `DRAT-trim`, we ensured that the LRAT proof is verified by the LRAT checker[28] in preliminary runs, but we do not generate LRAT (or GRAT) proofs for the final measurments. For proofs rejected by `rate`, we always run `sick-check`, to double-check that the proof is incorrect under to the semantics of specified DRAT. For our this evaluation we also disabled assertions and logging in `rate` which seems to give small speedups.

For running the solvers we used the same limits as in the SAT competition — 5000 seconds CPU time and 24 GB memory using runlim[29]. Similarly, for checking, where the timeout is 20000 seconds. We present performance data as reported by `runlim` — time in seconds and memory usage in megabytes (2^{20} bytes).

We performed all experiments on a machine with two AMD Opteron 6272 CPUs with 16 cores each and 220 GB main memory. We used GNU parallel [19] to run 32 jobs simultaneously. This high load slows down the solvers and checkers, most likely due to increased memory pressure, however, based on preliminary experiments we believe that the checkers are affected equally hence it is still a fair comparison.

5.1 Comparison of Checkers

On an individual instance two checkers might have different performance because of different propagation order and, as a result, different clauses being added to the core. In Figure 3 and 4 we show the distribution of performance narrowed down to the most difficult proof instances. For easier instances the differences are smaller. Figure 3 suggests that `gratgen` is a bit faster, and `DRAT-trim` is slower than `rate`. Moreover `rate`, and `rate -d` show roughly the same distribution of runtimes. Figure 4 indicates that `drat-trim` and `rate` use roughly the same amount of memory, while `gratgen` needs a bit more. This is not surprising because we use almost the same data structures as `drat-trim`.

We take a closer look, comparing the performance of two checkers on each instance, see Figures 5, 6 and 7: in Figure 5 we see that `rate` exhibits small differences in specified and operational mode. Figure 6 shows that `gratgen` is faster than `rate` on most instances. Similarly, Figure 7 shows that `rate` is faster than `DRAT-trim` on most instances.

[28]https://github.com/acl2/acl2/
[29]http://fmv.jku.at/runlim/

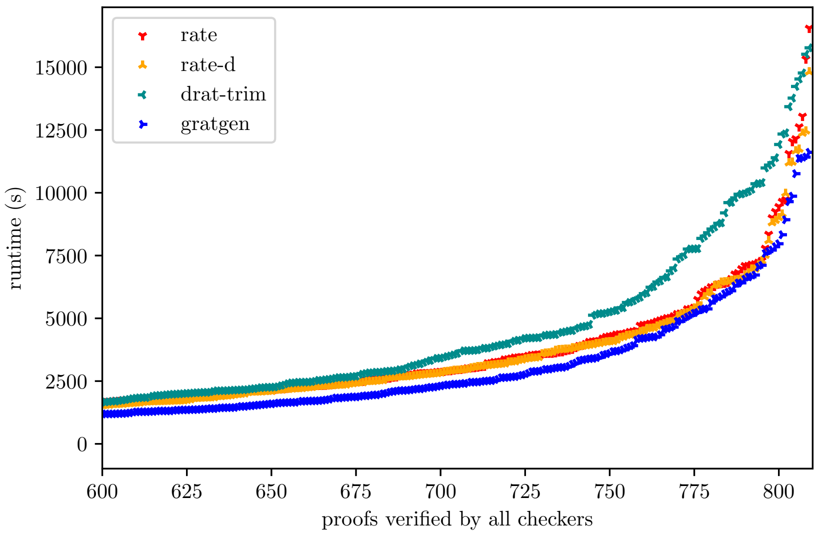

Figure 3: Cactus plot showing the distribution of checker runtimes

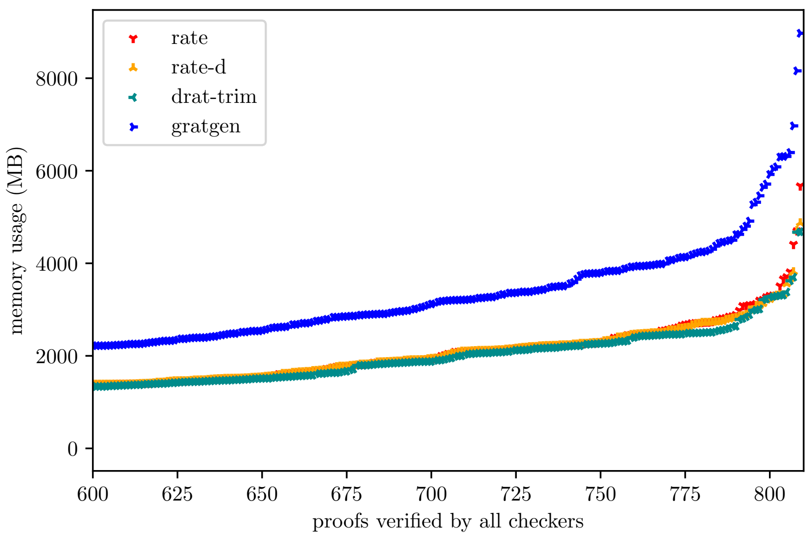

Figure 4: Cactus plot showing the distribution of the checkers' memory usage

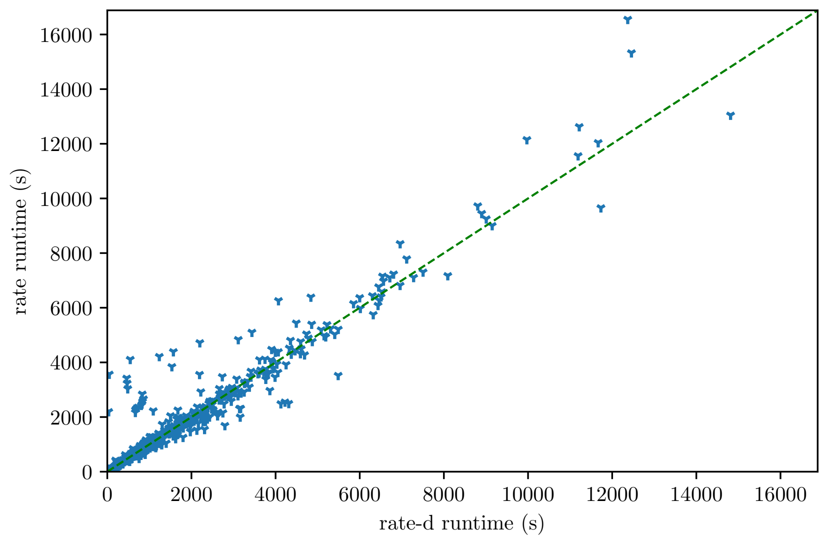

Figure 5: Cross plot comparing the individual runtimes of `rate -d` with `rate`. Each marker represents a proof instance.

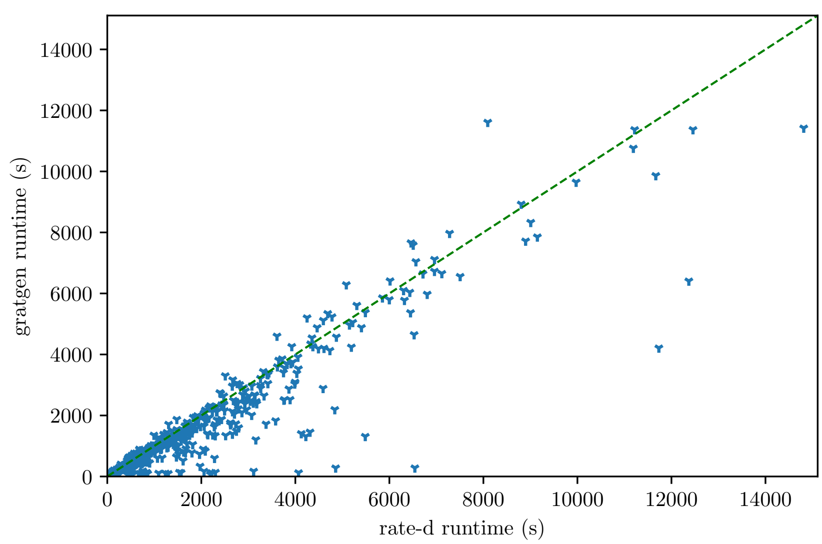

Figure 6: Cross plot comparing the individual runtimes of `rate -d` with `gratgen`. Each marker represents a proof instance.

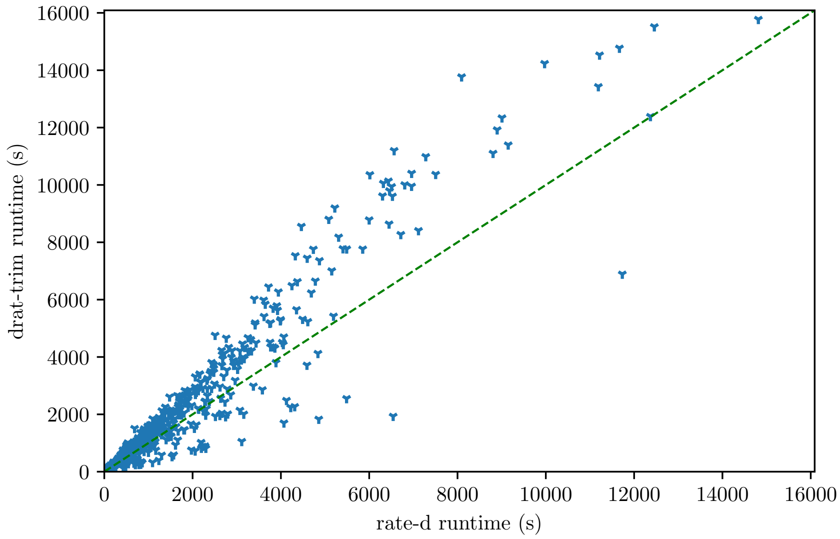

Figure 7: Cross plot comparing the individual runtimes of `rate -d` with `DRAT-trim`. Each marker represents a proof instance.

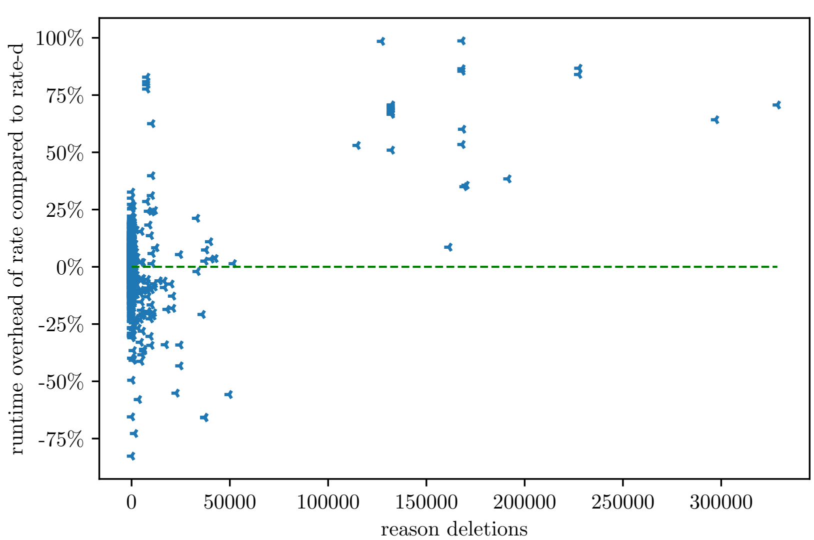

Figure 8: Juxtaposition of the number of reason deletions and the relative runtime overhead of checking specified DRAT over operational DRAT.

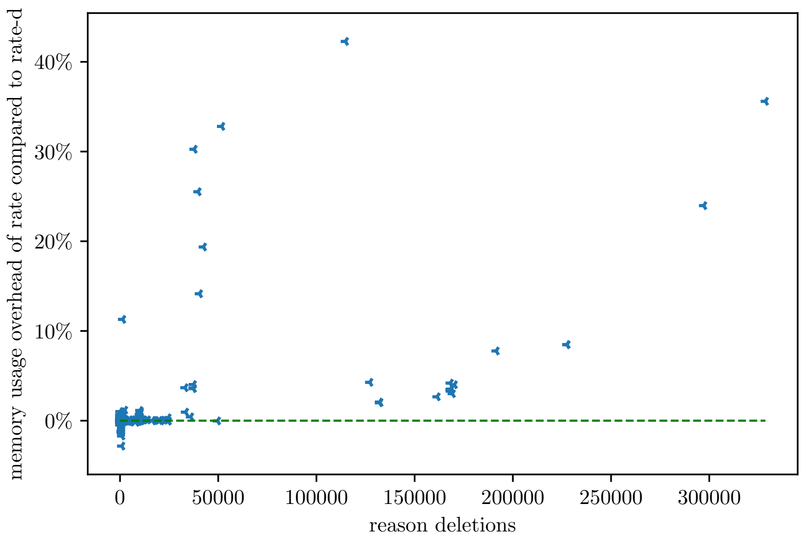

Figure 9: Juxtaposition of the number of reason deletions and the relative overhead in terms of memory usage of checking specified DRAT over operational DRAT.

5.2 Overhead of Reason Deletions

Figure 8 suggests that a large number of reason deletions brings about some runtime overhead in `rate` when checking specified DRAT as opposed to operational DRAT. Same holds for memory usage as can be seen in Figure 9. Currently, `rate` incurs these extra costs also for proofs that contain no unique reason deletions.

6. Conclusion

In Section 3 we have explained why operational DRAT is required to verify `DRUPMiniSat`-based solvers' proofs. We have proposed patches for these solvers to create proofs that are correct under either flavor and do not require ignoring unit deletions.

Specified DRAT is necessary to verify solvers' inprocessing steps that employ deletions of unique reason clauses [4]. We implement an efficient checker, `rate`, that supports both specified and operational DRAT. Furthermore, specified DRAT features the advantage that the accumulated formula is easy to compute without performing unit propagation. This enables us to produce SICK certificates which are small, efficiently checkable witnesses of a proof's incorrectness. They report which proof step failed, which can be used to detect bugs in checkers and pinpoint bugs in solvers. We provide a tool, `sick-check` to check SICK witnesses.

Our initial research question was whether specified DRAT can be checked as efficiently as operational DRAT. Based on our benchmarks we provided evidence that the cost for specified DRAT is, on average, the same but an excessive number of reason deletions can make it significantly more costly.

7. Future Work

If a checker for specified DRAT were to be adopted, it might be beneficial to implement a way to perform deletions of non-unique reasons more efficiently than `rate` does. These deletions do not alter the shared UP-model, but `rate` does not know this. An optimization could consist of an efficiently computable criterion to determine if some reason clause is unique. A simple criterion is as follows: if a reason clause for some literal l is deleted, check if unit clause l is in the formula. If it is, then the deleted reason is not unique and the shared UP-model will definitely not change. This criterion might be sufficient for the proofs produced by the second variant of the patches from section 3.

State-of-the-art DRAT checkers are heavily optimized for speed but they keep the entire input proof and the resulting LRAT proof in memory. If the available memory is at premium, some changes could be made to do backwards checking in a streaming manner. Additionally, the LRAT proof output could be streamed as well, with some postprocessing to fix the clause IDs.

It might be possible to forego DRAT completely and directly generate LRAT in a solver which is done by `varisat`. This removes the need for a complex checker at the cost of a larger proof artifact.

References

[1] R. Brummayer and A. Biere, “Fuzzing and delta-debugging smt solvers,” *ACM International Conference Proceeding Series*, pp. 1–5, Jan. 2009.

[2] R. Brummayer, F. Lonsing, and A. Biere, “Automated testing and debugging of sat and qbf solvers,” 2010, vol. 6175, pp. 44–57.

[3] M. J. H. Heule, W. A. Hunt, and N. Wetzler, “Bridging the gap between easy generation and efficient verification of unsatisfiability proofs,” *Software Testing, Verification and Reliability*, vol. 24, no. 8, pp. 593–607, Sep. 2014.

[4] A. Rebola-Pardo and A. Biere, “Two flavors of drat,” *Pragmatics of SAT*, vol. 2018, 2018.

[5] M. J. H. Heule, “Schur number five,” Nov. 2017.

[6] A. Rebola-Pardo and L. Cruz-Filipe, “Complete and efficient drat proof checking,” 2018.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *Proceedings of the 38th annual design automation conference*, 2001, pp. 530–535.

[8] J. Marques-Silva, I. Lynce, and S. Malik, “Conflict-driven clause learning sat solvers,” in *Handbook of satisfiability*, 1st ed., Netherlands: IOS Press, 2009, pp. 131–153.

[9] M. J. H. Heule, B. Kiesl, and A. Biere, “Short proofs without new variables,” *Lecture Notes in Computer Science*, pp. 130–147, 2017.

[10] A. Van Gelder, “Verifying rup proofs of propositional unsatisfiability.” 2008.

[11] M. Järvisalo, M. Heule, and A. Biere, “Inprocessing rules,” 2012, vol. 7364.

[12] M. J. H. Heule, W. A. Hunt, and N. Wetzler, “Verifying refutations with extended resolution,” in *Automated deduction – cade-24*, 2013, pp. 345–359.

[13] N. Wetzler, M. J. H. Heule, and W. A. Hunt, “DRAT-trim: Efficient checking and trimming using expressive clausal proofs,” *Theory and Applications of Satisfiability Testing – SAT 2014*, pp. 422–429, 2014.

[14] T. Philipp and A. Rebola-Pardo, "Towards a semantics of unsatisfiability proofs with inprocessing."

[15] L. Cruz-Filipe, M. J. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified rat verification," in *International conference on automated deduction*, 2017, pp. 220–236.

[16] M. J. H. Heule, W. A. Hunt, and N. Wetzler, "Trimming while checking clausal proofs," *2013 Formal Methods in Computer-Aided Design*, Oct. 2013.

[17] P. Lammich, "The grat tool chain," in *International conference on theory and applications of satisfiability testing*, 2017, pp. 457–463.

[18] P. Lammich, "Efficient verified (un) sat certificate checking," in *International conference on automated deduction*, 2017, pp. 237–254.

[19] O. Tange, *GNU parallel 2018.* Ole Tange, 2018.