A Tiny Tweak to Proof Generation in MiniSat-based SAT Solvers & A Complete and Efficient DRAT Proof Checker

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

European Master's Program in Computational Logic

eingereicht von

Johannes Altmanninger
Matrikelnummer 01455764

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Mitwirkung: Adrián Rebola-Pardo, MSc.

Wien, 1. Oktober 2019

Johannes Altmanninger	Georg Weissenbacher

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Tiny Tweak to Proof Generation in MiniSat-based SAT Solvers & A Complete and Efficient DRAT Proof Checker

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

European Master's Program in Computational Logic

by

Johannes Altmanninger
Registration Number 01455764

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Assistance: Adrián Rebola-Pardo, MSc.

Vienna, 1st October, 2019

Johannes Altmanninger Georg Weissenbacher

Erklärung zur Verfassung der Arbeit

Johannes Altmanninger

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Oktober 2019

Johannes Altmanninger

Kurzfassung

Das klauselbasierte Beweisformat DRAT ist der De-Facto-Standard, um Unerfüllbarkeitsergebnisse von SAT-Gleichungslösern zu verifizieren. DRAT-Beweise bestehen aus Lemmas, die Klauseln hinzufügen, und Klauseleliminierungen. Moderne DRAT-Beweisprüfer ignorieren Eliminierungsinstruktionen von sogenannten Unit-Klauseln, was bedeutet, dass sich die Semantik der Beweisprüfer von der DRAT-Spezifikation unterscheidet; infolgedessen können sie manche, von SAT-Lösern verwendete Vereinfachungstechniken, die Unit-Klauseln eliminieren unter Umständen nicht verifizieren. Moderne SAT-Löser generieren Beweise die von diesen DRAT-Prüfern akzeptiert werden, jedoch bezüglich der DRAT-Spezifikation inkorrekt sind, da sie Eliminierungsinstruktionen enthalten, die nicht-redundante Informationen löschen. Wir schlagen Korrekturen für prämierte SAT-Löser vor, wodurch diese in der Lage sind, Beweise ohne ebenjene kontraproduktiven Eliminierungen zu generieren, die ergo im Sinne der Spezifikation korrekt sind. Dennoch können Unit-Eliminierungen in Beweisen in Anbetracht der Verwendung von fortgeschrittenen Vereinfachungstechniken wohl kaum ausgeschlossen werden, sofern der Löseaufwand nicht darunter leiden soll. Die Durchführung von Unit-Eliminierungen in Beweisprüfern kann viel Rechenzeit beanspruchen. Wir haben den ersten wettbewerbsfähigen Prüfer implementiert, der Unit-Eliminierungen berücksichtigt and präsentieren unsere Versuchsergebnisse, die darauf hindeuten, dass der Rechenaufwand für das Überprüfen eines durchschnittlichen Beweises mit oder ohne Unit-Eliminierungen gleich ist. Weiters führen wir das SICK-Format ein, das, vergleichsweise kleine und effizient überprüfbare Gegenbeispiele zu DRAT-Beweisen beschreibt. Indem wir diese Gegenbeispiele mit einem unabhängigen Programm überprüfen, stärken wir das Vertrauen in die Stichhaltigkeit der Inkorrektheits-Ergebnisse. Außerdem kann diese Technik von Nutzen sein um Fehler in (der Beweisgenerierung von) SAT-Lösern und Prüfern zu finden.

Abstract

Clausal proof format DRAT is the de facto standard way to certify SAT solvers' unsatisfiability results. DRAT proofs consist of lemmas (clause introductions) and clause deletions. 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 may not be able to verify inprocessing techniques that use unit deletions. State-of-the-art SAT solvers produce proofs that are accepted by those DRAT checkers, but are incorrect under the DRAT specification, because they contain spurious deletions of unit clauses. We present patches for award-winning SAT solvers to produce proofs without those deletions that are thus correct with respect to the specification. However, handling unit deletions is still desirable, as they can be a byproduct of advanced inprocessing techniques in a solver that is hard to avoid without extra costs. Performing unit deletions in a proof checker can be computationally expensive. We implemented the first competitive checker that honors unit deletions and provide experimental results suggesting that, on average, checking costs are the same as when not applying unit deletions. As it is also expensive to determine the incorrectness of a proof, we introduce the SICK format which describes small and efficiently checkable certificates of the incorrectness of a DRAT proof. By checking this independently, we are able to increase trust in our incorrectness results. Additionally it can be useful when debugging solvers and checkers.

Contents

CHAPTER 1

Introduction

Over the past decades, there has been significant progress in SAT solving technology. At the same time, SAT solvers have had documented bugs [BB09] [BLB10]. 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* can be checked 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 during solving. In SAT competitions, solvers are required to give proofs of unsatisfiability in the DRAT proof format [HJW14]. 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 some clauses called *unit clauses* [RB18]. 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 [RB18]. The classes of proofs verified by checkers of these two flavors of DRAT are incomparable.

Specified DRAT can be required to verify advanced inprocessing steps performed by state-of-the-art solvers whose proofs contain unit clause deletions [RB18]: since the correctness of the *non-monotonic* inferences relies not only on the presence, but also the absence of clauses, it is crucial that deletions are honored. The proofs for inprocessing techniques like XOR reasoning [PR16] and symmetry breaking [HJW15] are generated by hard-coded routines; it would require additional reasoning to detect unit deletions. Additionally, when a solver reuses a variable that only exists in clauses that have been deleted, an operational checker might give an incorrect result due to these clauses.

Moving from operational DRAT to specified DRAT requires changes in some state-of-the-art SAT solvers because their proofs can be incorrect under specified DRAT. To the best of our knowledge, all of those solvers are based on MiniSat [ES03]. Their proofs can be incorrect under specified DRAT because they may contain some deletions of unit clauses. As we did not see a motivation for those deletions, we investigated how they are being generated. We found that the solvers who emit the problematic deletions do not undo inferences made using the deleted clauses, hence acting as if they had not been deleted. Since operational DRAT ignores deletions of unit clauses, state-of-the-art proof checkers interpreted it as if the clauses were not deleted, matching the solvers' internal behavior. Luckily, fixes appear to be simple and non-intrusive: we provide small patches for award-winning solvers to make them generate proofs without counter-productive unit deletions, eliminating the need to ignore them.

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 costs [HJW14]. 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 [Heu18]. Clearly, it is essential for a proof checker to be as efficient as possible to deal with such workloads. When considering switching to specified DRAT, it is important to assess any additional costs. A checker for specified DRAT may incur an overhead due to the need to honor unit deletions. There is an efficient algorithm to check specified DRAT [RC18] that introduces several optimizations that are not necessary for checking operational DRAT. Previous results suggest that the checking costs do not change significantly. However, those results were based on a checker that was not as efficient as state-of-the-art operational DRAT checkers. This motivates our central research question:

Is it possible to check specified DRAT as efficiently as operational DRAT?

To answer this, 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 unit deletions may have a significant negative impact on checking performance for specified DRAT.

The majority of solvers at SAT competitions are derived from MiniSat and produce proofs that are incorrect under specified DRAT. For those incorrect 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, independent of the checker, which helps developers in debugging proof-generation and proof-checking algorithms. While checking operational DRAT efficiently is arguably easier than checking 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.

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

This thesis is organized as follows: In the next section we will introduce preliminary knowledge about SAT solving, proofs of unsatisfiability and proof checking, including the optimization challenges of specified DRAT checking. In Section 3 we propose how to change `MiniSat`-based solvers to produce unambiguously correct proofs. Section 4 concerns the efficient implementation of our specified DRAT checker: after briefly discussing other checkers we present our implementation and describe the SICK format for certificates of proof incorrectness. 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.

CHAPTER 2

Preliminaries

A *literal* is a propositional variable, like x, or the 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, complement-free 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 the assignment. Other literals are *unassigned.*

A clause is satisfied by an assignment I if any literal in the clause is satisfied by I. SAT solvers work on formulas in *conjunctive normal form* (CNF), conjunctions (or multisets) of clauses.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. Observe that every reason clause is a unit clause.

An assignment I is a *unit propagation model* (UP-model) of F when for each clause $C \in F$, I either satisfies C or there are at least two literals in C that are unassigned in I [RB18]. Formula F is *UP-satisfiable* if it has a UP-model. Otherwise — if complementary literals are implied by unit propagation, or the formula contains the empty clause — it is *UP-unsatisfiable.* If F is UP-satisfiable, its *shared UP-model* is the intersection of all

UP-models. It is the assignment consisting of all literals implied by unit propagation in F. It is a subset of every model of F. Clause C is called a *unique reason clause* in formula F if the shared UP-model of $F \setminus \{C\}$ is strictly smaller than the shared UP-model of F.

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 that represents the shared UP-model of the formula, 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*, temporarily adding size-one clauses to the formula. This triggers unit propagation, adding more literals to the trail. These literals are implied by unit propagation in the formula plus the current assumptions. Only assignments that are a superset of the trail are considered during search, which means that many assignments are pruned from the search space. Additionally, solvers may use inprocessing techniques to modify the formula without changing satisfiability. Once the trail falsifies a clause, the (UP-)unsatisfiability of the formula plus assumptions is established. If there are assumptions, some of them are undone (*backtracking* [MS99]) and the solver resumes search. Otherwise, if there are no assumptions, 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 [MMZ+01]. For each literal in the formula, it keeps a *watchlist* of clause references. Clauses in the watchlist of some literal are said to be *watched on* that literal. Each non-deleted clause is watched on two literals, which are also called its *watches*. Provided that Invariant 1 from [RC18] 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 its watches is falsified. When literal l is assigned, it is sufficient to visit clauses in the watchlist of $\overline{l}$ to find new unit clauses — that is clauses that became unit due to assigning l. For visited clauses that are not unit, the watches may need to be changed to restore Invariant 1.

As an example, we perform unit propagation on formula $F = \{x, \overline{y}\overline{x}z, \overline{x}y, xy\}$. Let the first two literals in each clause be watched. So only $\overline{y}$ and x are watched in the second clause. To change watches, we simply flip the order of the literals in the 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 changed, 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 triggers propagation of z, but no clause is watched on $\overline{z}$, so propagation ends at this point. Clause xy is never visited during propagation because it is not watched on any relevant literal. The fact that unrelated clauses do not need to be visited is the central advantage of the two-watched literal scheme.

CDCL Predominant SAT solvers implement Conflict Driven Clause Learning (CDCL) [SLM09] which is based on the following principle: whenever the formula plus assumptions is UP-unsatisfiable, 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 an unsatisfiability-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 [HKB17]. There are various criteria of redundancy, with different levels of expressivity and computational costs.

- *RUP* — a clause C is a *reverse unit propagation* (RUP) inference in formula F if $F' := F \cup \{\bar{l} \,|\, l \in C\}$ is UP-unsatisfiable [GN03]. To compute whether C is RUP, the negated literals in C are added as assumptions and propagated to determine whether the formula is UP-satisfiable. If after some propagation steps the trail is a not a UP-model, C is RUP. A clause that is RUP in F is logical consequence of F [Gel08].
- *RAT* — a clause C is a *resolution asymmetric tautology* (RAT) [JHB12] 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 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. A clause that is RAT in F is not necessarily a logical consequence of F, yet adding it to F preserves satisfiability. RAT inference is non-monotonic: if C is RAT in F then it may not be RAT in $F' \supseteq F$.

DRAT Proofs Proofs based on RUP alone are not expressive enough to simulate all inprocessing techniques in state-of-the-art SAT solvers [HJW13b]. Because of this, the more powerful criterion RAT is used today [Gel12] [HJW13b]. A DRAT proof (*delete resolution asymmetric tautology*) [WHJ14] [Heu16] 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 to determine UP-satisfiability. 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 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 of the adjusted accumulated formula are ignored [RB18].

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 [HJW14] [WHJ14]. While deletions were added as an optimization, they can also enable additional inferences due to RAT being non-monotonic [PR17]. 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 [Heu18]. 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 formally verify. The lack of a formally verified DRAT checker is remedied by making the DRAT checker output an annotated proof in the LRAT format [CHJ$^+$17]. The LRAT proof can be checked by a formally verified checker without unit propagation, making sure that the formula formula is indeed unsatisfiable [HJKW17]. 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 guide unit propagation: for each resolution candidate, the resolvent is shown to be a RUP inference by giving a propagation sequence that shows UP-unsatisfiability of the formula with the negated literals in the resolvent.

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 state-of-the-art DRAT checkers, there is 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. 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 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 in a proof might not be necessary. *Backwards checking* [HJW13a] avoids checking superfluous lemmas by only checking *core* lemmas — lemmas that are part of the *unsatisfiable core*, an unsatisfiable subset of an unsatisfiable formula. Initially, the clauses from the propagation sequence that establishes UP-unsatisfiability of the formula is added to the core. Then the proof is processed backwards; only core lemmas are checked. Every lemma that is used in some propagation sequence to derive another 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.

Backwards checking can be implemented using two passes over the proof: a forward pass that merely performs unit propagation after applying each proof step [RC18] until UP-unsatisfiability is established, 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 an example for backwards checking: let $F = \{xyz, xy\overline{z}, \overline{x}y, x\overline{y}, \overline{xy}, x\overline{z}\}$ be an unsatisfiable formula with a proof consisting of two lemmas ($\textbf{add } xy, \textbf{add } x$). In the forward pass both lemmas are applied, then the accumulated formula is UP-unsatisifiable. To show UP-unsatisfiability, assume we propagate clauses x, $\overline{x}y$ and $\overline{xy}$ which are added to the core. Then backwards checking can start: first we check lemma x because it is in the core: it is RUP in $F \cup \{xy\}$ since $F \cup \{xy\} \cup \{\overline{x}\}$ is UP-unsatisfiable. We show this by propagating the clauses $x\overline{y}$, xyz and $xy\overline{z}$, which are then added to the core. Subsequently we can skip checking lemma xy because it is not in the core; it is virtually deleted from the proof.

Core-first Unit Propagation To keep the core small and reduce checking costs, core-first unit propagation was introduced [HJW13a]. It works by doing unit propagation in two alternating phases:

1. Propagate exhaustively using clauses already in the core.
2. If there is some non-core unit clause with an unassigned unit literal, add this literal to the trail and go to step 1, which will propagate this literal while only considering core clauses. Otherwise, terminate.

This results in a minimum of clauses being added to the core because whenever UP-unsatisfiability can be shown without adding a new clause to the core, this will be done by core-first unit propagation. This generally makes checking faster because the number of visited clauses while propagating decreases on average.

Consider the same example as above but assume we do check lemma xy, even though it is not in the core. To show that xy is RUP in F we show that $F \cup \{\overline{x}, \overline{y}\}$ is UP-unsatisfiable. This can be done by propagating clauses xyz and $x\overline{z}$. However, $x\overline{z}$ is not in the core, so core-first propagation will use $xy\overline{z}$ instead.

Reason Deletions Under operational DRAT, unit deletions are ignored. Only proofs with unique reason deletions have different semantics under specified and operational DRAT, because only those deletions alter the shared UP-model. State-of-the-art DRAT checkers do not implement a way to detect unique reason deletions, but they can emit a warning whenever a unit clause is deleted. To detect unique reason deletions, it is necessary check whether a reason deletion alters the shared UP-model, which is only implemented in specified DRAT checkers. 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 [RC18] does along other non-trivial routines to maintain the watch invariants.

Reason Deletions in Inprocessing Steps Some inprocessing techniques introduce or delete variables [RB18]. A deletion of a variable means that both of its literals are deleted from all clauses. A deletion of literal l from a clause C is modelled in the proof by first adding $C \setminus \{l\}$ and subsequently deleting C. Under operational DRAT, the deletion of C is not performed if C is a unit. This may cause a misinterpretation of the proof if the variable l is later reused by being introduced by another inprocessing step.

Inprocessing steps for XOR reasoning [PR16] and symmetry breaking [HJW15] are supported by proof fragments generated by hard-coded routines [RB18]. It may not be easy or efficient for such routines to produce proof fragments any without reason deletions. Under operational DRAT the deleted reason clauses would linger, thus the proof is checked in a different way than intended.

CHAPTER 3

A Tiny Tweak to Proof Generation in **MiniSat**-based SAT Solvers

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 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`[1] emits unique reason deletions. This solver performs simplification of the formula when there are no assumptions (*decision-level zero*), so the trail is equivalent to the shared UP-model of the formula without assumptions. The literals that are in the trail at this point will never be unassigned by `DRUPMiniSat`.

One 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

[1] 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/

$F = \{x, xy\}$. 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.

In `MiniSat`, reason clauses are called *locked*. The method `Solver::removeSatisfied` also deletes locked clauses, however, the literals propagated 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 literals when deleting clauses, matching `DRUPMiniSat's` internal (but not external) 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 have proposed[2] 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 the shared UP-model stays the same.

2. Before removing a locked clause C, emit its unit literal $l \in C$ as a introduction of size-one clause l in the DRAT proof. Suggested by Mate Soos[3], this option is also preferred by the authors of `mergesat`[4] and `varisat`[5]. Additionally, this is implemented in `CaDiCaL's` preprocessor. This does not influence the correctness of future inferences because C and l are logically equivalent with respect to the shared UP-model — they will behave identically for all future inferences since the literals in assignment at decision-level zero will never be unassigned. In our example this means that another instance of x is added in the proof, before one x is deleted, which preserves the formula.

We provide patches implementing these for `MiniSat` version 2.2 (1. [6] and 2.[7]), and the winner of the main track of the 2018 SAT competition (1.[8] and 2.[9]). Both patches are arguably very simple and we do not expect any significant impact 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 at most one unit clause addition per variable, which is

[2] https://groups.google.com/d/msg/minisat/8AXELMFPzPY/8K8Tq-WVBQAJ

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

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

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

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

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

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

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

tiny compared to the rest of a typical proof. The patches can be easily adapted to other `DRUPMiniSat`-based solvers.

CHAPTER 4

A Complete and Efficient DRAT Proof Checker

We have implemented 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 verify proof incorrectness.

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[1] their way of producing LRAT proofs and ensure that all our proofs are verified by a formally verified checker. 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. Here is why this is sound: 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

[1] See the function `sortSize()` in `DRAT-trim`.

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 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[2].

GRAT Toolchain More recently, Peter Lammich has published the GRAT toolchain [Lam17b]. Given a DRAT proof, they first produce a GRAT proof which is similar to LRAT with the `gratgen` tool, which outperforms `DRAT-trim` [Lam17a]. Subsequently, the formally verified `gratchk` can be used to check the GRAT proof, 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 as of now.

Here are two of their optimizations:

- Separate watchlists for core and non-core clauses[3]. 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 boils down to to a branch instruction which can be moved outside the loop by partitioning the watchlists into core and non-core clauses.

- Resolution candidate caching / RAT run heuristic [Lam17a]: 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 do not 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 it can do a parallel checking — so we advise interested readers to study that.

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

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

rupee This is the original implementation[4] of the first efficient algorithm to honor unique reason deletions. We use the same algorithm. During our research we found one issue in the implementation which was fixed[5].

In previous experiments, rupee was an order of magnitude slower than DRAT-trim [RC18]. We believe that this overhead is primarily not a consequence of algorithmic differences but of implementation details such as missing optimizations in the parser and the lack of 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 which may stand for "rate ain't trustworthy either". It is a MIT-licensed clausal proof checker that aims to be user-friendly, easy-to-understand and efficient. Source code and documentation can be found at https://github.com/krobelus/rate/.

To facilitate adoption, rate 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 supported by a SICK certificate of incorrectness. The representation of the input 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[6]. Other checkers cannot provide the latter. This might be useful to validate that a proof contains no unique reason deletion, since most solvers do not use advanced inprocessing techniques requiring unique reason deletions and thus probably do not need to use them.

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

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

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

As other state-of-the-art checkers, rate deviates from the specification of DRAT [Heu16] where that is convenient or necessary for competitive performance. For example, it fails when given 2^{30} or more clauses. Like other checkers, rate accepts proofs that do not contain the empty clause. Due to backwards checking many lemmas are skipped, so proofs with incorrect lemmas may be accepted as well. We allow proofs that are missing a zero-terminator in their last proof step because some solvers in the 2018 SAT competition did not always write that zero.

The checker is extensible to other clausal proof formats, for example we support the proof format, DPR (*delete propagation redundancy*) [HKB17] which supersedes DRAT.

To automatically minimize inputs that expose bugs in our checker we have developed a set of scripts to delta-debug CNF and DRAT instances. Consider that we had proof instances of several gigabytes that provoked crashes in rate. Using a combination of binary search and deletion of random lines we were usually able to minimize interesting instances to mere kilobytes. One important tool do do binary search in proofs is our apply-proof, which takes a formula and a clausal proof and applies a given number of proof steps, outputting the accumulated formula as well as the rest of the proof.

We chose the modern systems programming language Rust[7] for our implementation because of its feature parity with C in the domain of SAT solving. Among the respondents of the 2019 Stack Overflow Developer Survey[8] 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[9], is a great example of this. They use a library[10] to avoid writing a lot of boilerplate code necessary to satisfy Rust's borrow checker.

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 disable bounds checking by default in most routines, 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 The 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

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

[8] https://insights.stackoverflow.com/survey/2019

[9] https://github.com/jix/varisat/

[10] https://jix.one/introducing-partial_ref/

is correct as well. However, many proofs are rejected by our checker for specified DRAT. To trust those results, we want to independently verify the incorrectness of such proofs.

A DRAT 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 $F \cup \{\bar{l} \,|\, l \in C\}$ is UP-satisfiable. On top of that, to show that C is not RAT, it suffices to show that any resolvent with C is not RUP. For example, consider formula $F = \{xy, \overline{x}y, x\overline{y}\}$ and lemma $\overline{x}\overline{y}$ which is neither RUP nor RAT. To refute RUP for lemma $\overline{x}\overline{y}$ we find a UP-model of $F \cup \{xy\}$, for example $\emptyset$ or $\{xy\}$. To refute RAT for $\overline{x}\overline{y}$ on pivot $\overline{x}$ we check that some resolvent on $\overline{x}$ is not RUP. The first resolvent $y\overline{y}$ is a tautology and thus trivially RUP but the second resolvent $\overline{y}$ is not: $F \cup \{y\}$ has a UP-model $\{xy\}$. The same thing can be shown for the resolvents on the other pivot $\overline{y}$.

Since our checker already computes the shared UP-model for each RUP check, we can output that for an incorrect inference 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 watchlists is non-trivial to implement correctly for specified DRAT. A bug in a watchlist implementation typically provokes problems when it causes some clause to not be watched when it is 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 looking for a clause that is unit but not satisifed. On the other hand, if the checker's shared UP-model is bigger than the actual shared UP-model, that would be a bug that is not easily be detected by such a tool. However, a proof is never incorrectly rejected because the computed shared UP-model is too large, since a larger model increases the likelihood of finding a conflict.

Proof incorrectness certificates have been proposed in [RB18]. The format we use 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[11] syntax; see Figure 4.1 for a grammar. An example application of a SICK certificate is shown in Figure 4.2. 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 RUP and RAT checks failed for the first lemma in the 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, let the *lemma*

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

SICK	:=	ProofFormat [ProofStep] NaturalModel Witness*
ProofFormat	:=	`proof_format = ` FormatSpec
FormatSpec	:=	(`"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 4.1: 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 4.2: Example SICK certificate for an incorrect proof

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. If `proof_step` is omitted, it means that the proof does not add enough clauses to make the accumulated formula UP-unsatisifiable.

- `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 specifying the witness explicitly.

- natural_model gives the shared UP-model before checking this proof step. If proof_step is omitted, this is the shared UP-model after applying all proof steps.

Each witness is a counter-example to some inference and comprises the following elements:

- 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 inference check.
- pivot: This specifies the pivot literal.

If the lemma is the empty clause, no witness is necessary, since the empty clause cannot be RAT. Additionally, if the proof_step is omitted, no witness is necessary either.

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 (if present).
2. The given natural_model is a UP-model of F.
3. 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 a UP-model of $F \cup \{\bar{l} \,|\, l \in r\}$ where r is the resolvent on pivot of the lemma and the failing_clause.
4. If the format is DRAT-arbitrary-pivot, the lemma is equal to the set of the pivots in the given witnesses.

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 the new syntax that takes into account the variants of DRAT with a fixed or an arbitrary pivot.

CHAPTER 5

Experimental Evaluation

Here we present a performance evaluation of our checker. Technical details are available[1]. We analyze 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.

Experimental Setting Each individual benchmark consists of a SAT problem instance and a solver to produce a proof for this instance. For each benchmark, we run the solvers with the same limits as in the SAT competition — a maximum of 5000 seconds CPU time and 24 GB memory, both imposed by runlim[2]. Then we run the checkers on the resulting proof using a time limit of 20000 seconds, as in the competition. For `rate`, `rate-d` and `DRAT-trim`, we did ensure that the LRAT proof is verified by the verfied checker `lrat-4`[3] in preliminary runs. However, we do not generate LRAT (or GRAT) proofs for the final measurements because based on preliminary experiments we do not expect any interesting differences stemming from LRAT proof output routines. For proofs rejected by `rate`, we run `sick-check`, to double-check that the proof is incorrect under to the semantics of specified DRAT. For the final evaluation we also disabled assertions and logging in `rate` and `rate-d` which seems to give small speedups.

[1] https://github.com/krobelus/rate-experiments

[2] http://fmv.jku.at/runlim/

[3] https://github.com/acl2/acl2/tree/master/books/projects/sat/lrat

We performed all experiments on a machine with two AMD Opteron 6272 CPUs with 16 cores each and 220 GB main memory running Linux version 4.9.189. We used GNU parallel [Tan18] to run 32 jobs simultaneously. Such a high load slows down the solvers and checkers, likely due to increased memory pressure. Based on preliminary experiments we expect that the checkers are affected equally so we assume that a comparison between the checkers is fair.

Benchmark Selection We take from the 2018 SAT competition[4] both the SAT instances and the solvers from the main track, excluding benchmarks that are not interesting for our purpose of evaluating `rate's` performance. We consider only unsatisfiable instances where the solver does not time out, and where the resulting proof is not rejected by `rate`. The latter condition ensures a fair comparison in terms of checker performance: when `rate` rejects a proof it terminates as soon as an incorrect instruction is encountered in the backward pass. This means that it has verified only a fraction of the proof while other checkers would verify the entire proof. Hence it is not useful for benchmarking checker performance to include proofs that are rejected under specified DRAT.

Starting from the benchmarks where — according to the competition results[5] — the solver successfully produced a proof of unsatisfiability, here is how many benchmarks were removed by the criteria mentioned above:

All benchmarks	3653
where the solver does not time out	3605
from which `rate` rejects the proof	2762
from which `rate` rejects the proof due to an overflow	1
selected benchmarks	842
from which `rate` verifies the proof	839
from which `rate` times out	2
from which `rate` runs out of memory	1

5.1 Comparison of Checkers

We present performance data as reported by `runlim` — time in seconds and memory usage in megabytes (2^{20} bytes).

On an individual instance, two checkers can exhibit different performance because of different propagation orders and, as a result, different clauses being added to the core. Instead we compare the distribution of the checkers' performance. The distribution has a long tail of instances where the checkers' performance is similar. In Figure 5.1 we show only the head of that distribution where some differences emerge. We conclude that `gratgen` is a bit faster, and `DRAT-trim` is slower than `rate`. As expected, `rate`, and

[4] http://sat2018.forsyte.tuwien.ac.at/

[5] http://sat2018.forsyte.tuwien.ac.at/results/main.csv

`rate -d` show roughly the same distribution of runtimes. Because `DRAT-trim` and `rate` use almost the same data structures they use roughly the same amount of memory, while `gratgen` needs a bit more.

For a more detailled view, we compare each checker to `rate-d` on individual instances in Figure 5.2: we see that `rate` and `rate-d` behave alike on most instances; `gratgen` is faster than `rate-d` on most instances, and `rate-d` is faster than `DRAT-trim` on most instances.

5.2 Overhead of Reason Deletions

Handling reason deletions may require extra time and memory. Figure @fig:correlation-reason-deletions shows the number of reason deletions and the overhead of `rate` compared to `rate-d` — among our benchmarks runtime at most doubles. Currently, `rate` incurs these extra costs also for proofs that contain no unique reason deletions (where unit deletions could simply be ignored altogether) — these instances are shown with red markers.

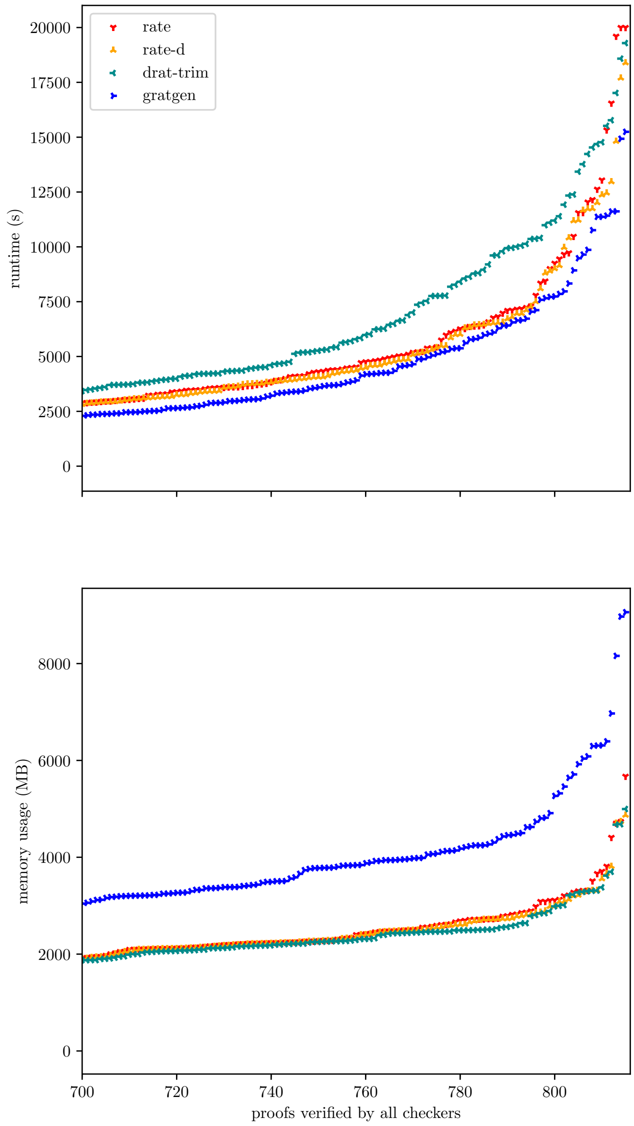

Figure 5.1: Cactus plot showing the distribution of checkers' runtime and memory usage.

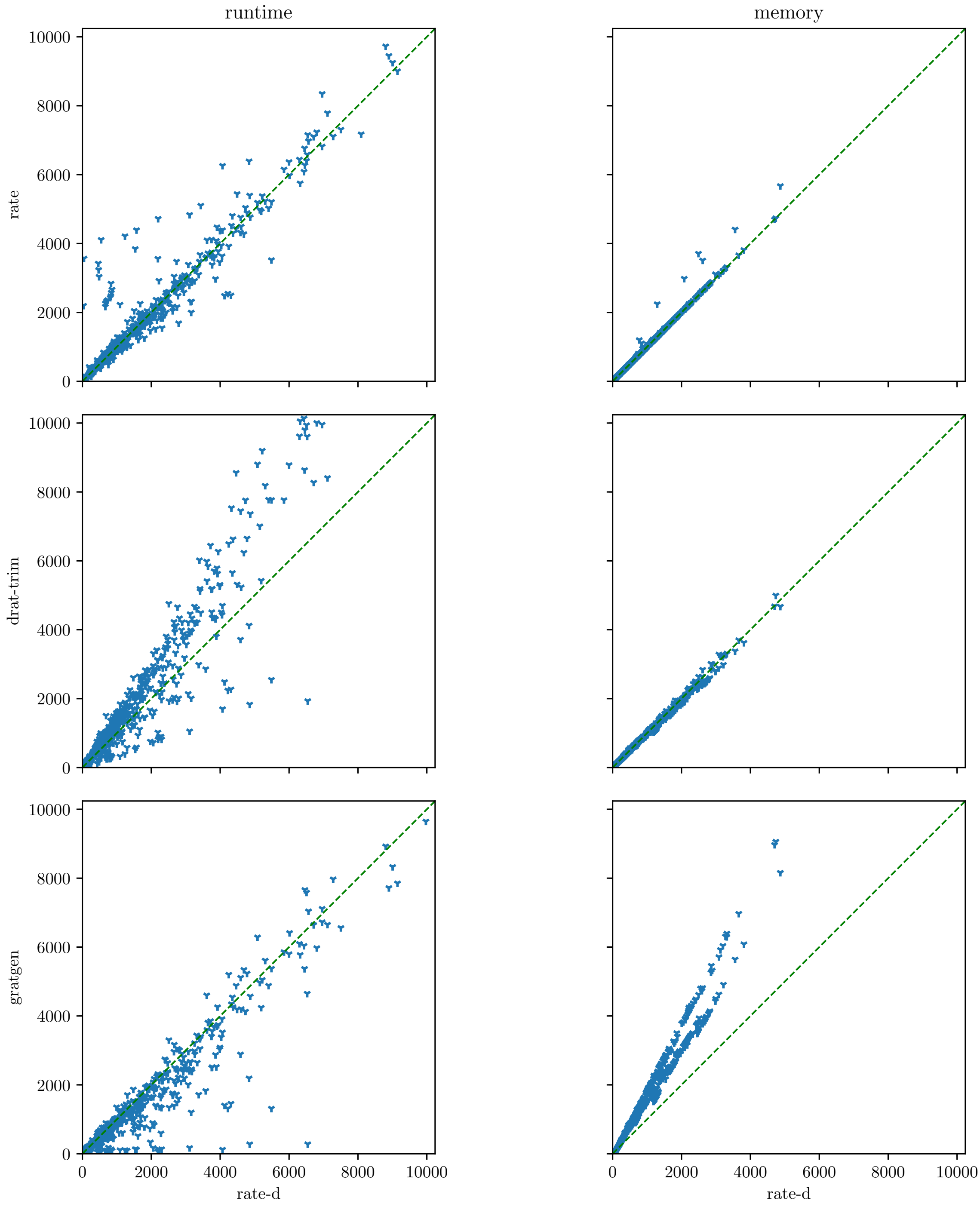

Figure 5.2: Cross plot comparing the runtime and memory usage of `rate -d` with the other checkers. Each marker represents a proof instance.

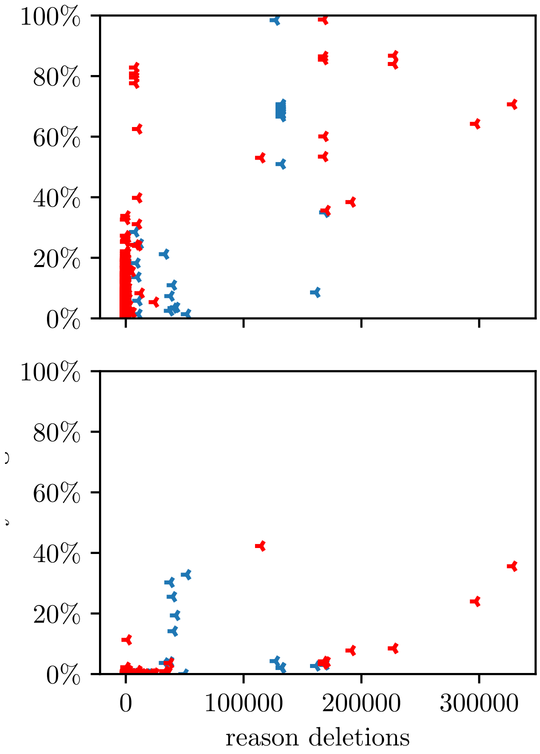

Figure 5.3: The number of reason deletions compared to the runtime and memory overhead of checking specified DRAT over operational DRAT.

CHAPTER 6

Conclusion

State-of-the-art SAT solvers produce proofs with deletions of unique reason clauses. These proofs are often incorrect under specified DRAT. Under operational DRAT they are correct because those deletions will effectively be removed from the proof. In Section 3 we have explained how `DRUPMiniSat`-based solvers produce proofs with unique reason deletions that render them incorrect under specified DRAT, and we have proposed patches to avoid those deletions, removing the need to ignore deletions to verify their proofs.

As we explained at the end of Section 2, specified DRAT is necessary to verify solvers' inprocessing steps that employ deletions of unique reason clauses [RB18]. Our research question was whether specified DRAT can be checked as efficiently as operational DRAT. Previous work has yielded an efficient algorithm but no competitive checker. We have implemented the first checker delivering state-of-the-art performance while supporting both specified and operational DRAT. We provide experimental results suggesting that that the cost for specified DRAT is, on average, the same but a high number of reason deletions may make it significantly more costly.

The two-watched literal scheme is difficult to implement correctly, and the optimizations from [RC18] to check specified DRAT efficiently further complicate that. Our checker implementation is able to output LRAT and GRAT certificates that can be verified by a formally verified checker, giving some confidence that `rate` gave the right answer. However, many proofs are rejected by `rate`. We needed a way to trust those incorrectness results. We extended the previously unpublished SICK format for proof incorrectness certificates and implemented a tool, `sick-check` that verifies those certificates, independent of `rate`. Since `sick-check` merely computes the accumulated formula up to the failed proof step and then checks that step without doing any propagation it is much simpler than a DRAT checker. These certificates can be used to detect bugs in checkers (we did find some in `rate`) and pinpoint bugs in solvers.

CHAPTER 7

Future Work

If 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` assumes they do and does more work than necessary, which sometimes even doubles the runtime for our benchmarks, as we showed in Figure @fig:correlation-reason-deletions. 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 an online fashion, processing one proof step at a time. Similarly, an LRAT proof line could be written to disk immediately after checking the corresponding lemma, 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 already supported by `varisat`. This removes the need for a complex checker at the cost of a larger proof artifact.

Bibliography

[BB09] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. *ACM International Conference Proceeding Series*, pages 1–5, 01 2009.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

[CHJ$^+$17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[Gel08] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008.

[Gel12] Allen Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.

[GN03] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10886–10891. IEEE Computer Society, 2003.

[Heu16] Marijn J. H. Heule. The DRAT format and drat-trim checker. *CoRR*, abs/1610.06229, 2016.

[Heu18] Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6598–6606. AAAI Press, 2018.

[HJKW17] Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasilia, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2017.

[HJW13a] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.

[HJW13b] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013.

[HJW14] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test., Verif. Reliab.*, 24(8):593–607, 2014.

[HJW15] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 591–606. Springer, 2015.

[HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2017.

[JHB12] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.

[Lam17a] Peter Lammich. Efficient verified (UN)SAT certificate checking. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2017.

[Lam17b] Peter Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 457–463. Springer, 2017.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[MS99] J. P. Marques-Silva and K. A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[PR16] Tobias Philipp and Adrian Rebola-Pardo. DRAT proofs for XOR reasoning. In Loizos Michael and Antonis C. Kakas, editors, *Logics in Artificial Intelligence - 15th European Conference, JELIA 2016, Larnaca, Cyprus, November 9-11, 2016, Proceedings*, volume 10021 of *Lecture Notes in Computer Science*, pages 415–429, 2016.

[PR17] Tobias Philipp and Adrián Rebola-Pardo. Towards a semantics of unsatisfiability proofs with inprocessing. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 65–84. EasyChair, 2017.

[RB18] Adrián Rebola-Pardo and Armin Biere. Two flavors of DRAT. In Daniel Le Berre and Matti Järvisalo, editors, *Proceedings of Pragmatics of SAT 2015, Austin, Texas, USA, September 23, 2015 / Pragmatics of SAT 2018, Oxford, UK, July 7, 2018.*, volume 59 of *EPiC Series in Computing*, pages 94–110. EasyChair, 2018.

[RC18] Adrian Rebola-Pardo and Luís Cruz-Filipe. Complete and efficient DRAT proof checking. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[SLM09] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[Tan18] Ole Tange. *GNU Parallel 2018*. Ole Tange, April 2018.

[WHJ14] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.