Frying the Egg, Roasting the Chicken: Unit Deletions in DRAT Proofs

Johannes Altmanninger
TU Wien
Austria
aclopte@gmail.com

Adrián Rebola-Pardo
TU Wien
Austria
arebolap@forsyte.at

Abstract

The clausal proof format DRAT is the standard de facto to certify SAT solvers' unsatisfiability results. DRAT proofs act as logs of clause inferences and clause deletions in the solver. The non-monotonic nature of the proof system makes deletions relevant. State-of-the-art proof checkers ignore deletions of unit clauses, differing from the standard in meaningful ways that require adaptions when proofs are generated or used for purposes other than checking. On the other hand, dealing with unit deletions in the proof checker breaks many of the usual invariants used for efficiency reasons. Furthermore, many SAT solvers introduce spurious unit deletions in proofs. These deletions are never intended to be applied in the checker but are nevertheless introduced, making many proofs generated by state-of-the-art solvers incorrect. We present the first competitive DRAT checker that honors unit deletions, as well as fixes for the spurious deletion issue in proof generation. Our experimental results confirm that unit deletions can be applied with similar average performance to state-of-the-art checkers. We also confirm that a large fraction of the proofs generated during the last SAT solving competition do not respect the DRAT standard. This result was confirmed with proof incorrectness certificates that were independently validated. We find that our proof incorrectness certificates can be of help when debugging SAT solvers and DRAT checkers.

CCS Concepts • **Theory of computation → Automated reasoning**.

Keywords DRAT proofs , DRAT checking , SAT solving

ACM Reference Format:
Johannes Altmanninger and Adrián Rebola-Pardo. 2020. Frying the Egg, Roasting the Chicken: Unit Deletions in DRAT Proofs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3372885.3373821

CPP '20, January 20–21, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7097-4/20/01.
https://doi.org/10.1145/3372885.3373821

1 Introduction

Over recent decades, SAT solving has overcome the status of an exploratory academic endeavour to become a widely used approach to solve industrial problems. SAT solvers are complex tools, and bugs in SAT solvers routinely occur. Since they solve NP-complete problems, negative results where the input formula is decided unsatisfiable are hard to confirm.

In order to certify SAT solvers' unsatisfiability results, the most widely used approach is to emit an unsatisfiability proof that can be checked with a different tool, a *proof checker*. The current state-of-the-art proof system is called Deletion Resolution Asymmetric Tautology (DRAT) [13], which has become an industry standard as well as a requirement for participation in SAT competitions[1]. In essence, a DRAT proof logs the clauses that were introduced or deleted at solving time. Checking a DRAT proof amounts to checking conditions which ensure that clauses are introduced in a satisfiability-preserving way [7, 28]. In a way, DRAT checkers replicate the solving process, which is reflected in similar runtimes to solving [11].

A design issue in the way proof checkers, as well as proof emission procedures within solvers, proceed was recently found [22]. On the one hand, proof checkers ignore a specific kind of clause deletions, called unit deletions [13]. On the other hand, proof emission procedures in many CDCL SAT solvers insert spurious unit deletions in generated proofs [22]. Two wrongs in this case do make a right, and so the checker ends up replicating the solving process.

In [22], Rebola-Pardo and Biere dispel concerns about the correctness of this method: one can conceive state-of-the-art DRAT checkers as fundamentally checking the proof against a different, yet logically sound, proof system. The originally defined proof system is referred to as specified DRAT, whereas the proof system implicitly defined by state-of-the-art checkers is called operational DRAT. Specified and operational DRAT proofs are incomparable: short examples exist of proofs that are correct for one flavor but incorrect for the other. However, both proof systems are also equivalent: any formula which is provable in operational DRAT is also provable in specified DRAT, and *vice versa*, albeit possibly

[1] http://sat-race-2019.ciirc.cvut.cz/

by different proofs. With both proof systems being sound, it is a matter of pragmatism whether to choose one proof system over the other.

In that same paper, a few issues with each flavor of DRAT were presented. Operational DRAT requires expensive unit propagation even to compute basic properties about the proof without checking it, and from a theoretical perspective having no-op steps in a proof is somewhat inelegant. Furthermore, generating operational DRAT proofs for inprocessing techniques can be problematic if unit lemmas are introduced. On the other hand, specified DRAT is essentially unsupported: most solvers emit proofs in operational DRAT [21], and all state-of-the-art checkers verify operational DRAT proofs [13, 17]. Whether SAT solvers were first in generating spurious deletions which checkers' developers decided to ignore, or checkers instead started ignoring unit deletions for simplicity rendering solvers free to spuriously emit them is a chicken-and-egg problem whose solution is unknown to the authors.

The problem of checking specified DRAT proofs is more complex than it might look like. In particular, applying unit deletions over a formula breaks a number of invariants maintained by state-of-the-art DRAT checkers, which in turn are essential for efficient proof checking. A recently proposed method for specified DRAT checking suggests that this can be done with negligible overhead over operational DRAT [21]. Unfortunately, the only existing implementation is about one order of magnitude slower than state-of-the-art DRAT checkers for both specified and operational DRAT due to the lack of some optimizations, so the question whether specified DRAT checking is possible at competitive runtimes stood on. Furthermore, even if an efficient DRAT checker was available, most solvers still generate spurious unit deletions that are meant to be ignored by checkers, and in fact many such proofs are incorrect when those deletions are applied [21].

Contributions In this paper, we propose both solver-side and checker-side solutions to the issue of unit deletions in DRAT proofs.

On the one hand, we propose patches to avoid generating spurious unit deletions in `MiniSat`-based solvers, which amount to 30 out of 36 solvers participating in the SAT competition 2018[2]. This is interesting because, within the bounds of plain CDCL without preprocessing, avoiding spurious unit deletions is feasible and yields proofs in the shared fragment of operational and specified DRAT. A patch for `CryptoMiniSat` [24] in the same spirit would need a bit of engineering effort. The remaining five solvers are unaffected because they do not generate unit deletions that delete information. They are: `CaDiCaL`[3], `Lingeling`[4], `Riss`[5], `Sparrow2Riss`[6] (which adds a preprocessor) and `varisat`[7].

On the other hand, we present a reimplementation of the ideas for specified DRAT checking from [21] in our DRAT checker `rate`. This is the first competitive checker for specified DRAT, even outperforming the de facto standard (operational) DRAT checker `drat-trim` [29]. While the tool `rupee` from [21] had similar performance when checking both specified and operational DRAT proofs, its overall performance was notably worse than state-of-the-art checkers. Our experimental results confirm that specified DRAT proofs can be checked not only without a significant overhead over operational DRAT, but also with a performance in line with modern DRAT checkers.

We observe, in line with previous results [21], that a large amount of our benchmarks are correct under the operational flavor yet rejected by `rate`. This raises the question whether this is due to proofs actually being incorrect under the specified flavor `rate` checks proofs against, or instead to implementation errors in `rate`. In order to provide guarantees that no proof is incorrectly rejected, we present the SICK certificate format. SICK certificates act as witnesses of the *incorrectness* of a proof; our `sick-check` tool is then able to independently check that the proof contains indeed an incorrect inference, hence confirming that `rate` does not mistakenly reject proofs.

The rest of this paper is organized as follows. Section 2 introduces the specified and operational DRAT proof systems. In Section 3 we propose a fix to the generation of spurious unit deletions in `MiniSat`-based SAT solvers. Our checker `rate` is presented in Section 4, and is experimentally analyzed in Section 5. In order to confirm the correctness of our results, SICK incorrectness certificates are presented in Section 6. Finally, we present our conclusions in Section 7.

2 Preliminaries

We consider an infinite set of *propositional variables*. For every variable x, a *literal* is either x or its negation $\overline{x}$; each of these literals is the *complement* of each other, and we denote the complement of a literal l by $\overline{l}$ by a slight abuse of notation. An *interpretation* I maps every variable to either 0 or 1. We also consider as a separate notion *partial interpretations*, which map variables to either 0, ?, or 1, with only a finite number of them not mapped to ?. Observe that there is a trivial correspondence between partial interpretations and finite, complement-free sets of variables; we will assume these notions are interchangeable. In SAT solving, formulas are assumed to be in *clausal normal form* (CNF). A *clause* is a finite, complement-free set of literals; we denote clauses by

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

[3] https://github.com/arminbiere/cadical

[4] https://github.com/arminbiere/lingeling

[5] https://github.com/nmanthey/riss-solver

[6] https://github.com/adrianopolus/Sparrow

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

juxtaposition, e.g. we write $xy\overline{z}$ to denote the clause $\{x, y, \overline{z}\}$. The empty clause is denoted by $\square$. A *CNF formula* is a finite set of clauses. Given a clause C, we consider the CNF formula $\overline{C}$ which consists of the size one clauses $\overline{l}$ where l is each literal in C.

The semantics of these elements are given as usual. An interpretation I *satisfies* a variable x whenever $I(x) = 1$, and satisfies its negation $\overline{x}$ whenever $I(x) = 0$. We say that I satisfies a clause C whenever I satisfies some literal in C; and I satisfies a CNF formula F whenever I satisfies all clauses in F. For any of these constructs X, we write $I \vDash X$ to mean that I satisfies X. Furthermore, if every interpretation I satisfying a construct X also satisfies a construct Y, then we write $X \vDash Y$. If for a construct X there is some interpretation I with $I \vDash X$, we say that X is *satisfiable*. A partial interpretation I satisfies a construct X whenever every interpretation I' that agrees with I on its assigned variables satisfies X.

2.1 Unit Propagation and UP-Models

The main reasoning technique used in SAT solving is *unit propagation*. Unit propagation is performed over a CNF formula F and some *assumed literals* A which form a partial interpretation. Intuitively, unit propagation extends A to another partial interpretation P in such a way that $F \wedge A \vDash P$; sometimes unit propagation finds that $F \wedge A$ is unsatisfiable.

From an operational perspective, unit propagation starts off with a partial interpretation $P := A$. Unit propagation iteratively refines P in such a way that $F \wedge A \vDash P$. At each iteration, unit propagation finds *triggered clauses*, which are clauses $C \in F$ such that some literal $l \in C$ is not satisfied, i.e. $P(l) \neq 1$; and furthermore every other literal $k \in C \setminus \{l\}$ is falsified, i.e. $P(k) = 0$. In the case that $P(l) = ?$, we let $P := P \cup \{l\}$. Otherwise, $P(l) = 0$, and then unit propagation *fails* and we deduce that $F \wedge A$ is unsatisfiable. The loop repeats until unit propagation fails or until no more triggered clauses remain in F.

In [22] a different perspective on unit propagation is proposed, which we adapt here to the goals of this paper. Given a partial interpretation P, we say that P UP-satisfies a clause whenever either $P(l) = 1$ for some literal $l \in C$, or $P(l) = P(k) = ?$ for some distinct literals $l, k \in C$; we write this as $P \vDash_{\text{up}} F$. Furthermore, P UP-satisfies a CNF formula F whenever all clauses in F are UP-satisfied by P. Intuitively, P UP-satisfies F whenever running unit propagation over F and P does not fail and leaves P unchanged.

The utility of UP semantics will become clear in Section 6: checking that a partial interpretation UP-satisfies a formula is simple, and rules out the possibility that the formula contains a clause falsified by the partial interpretation. We call a CNF formula F *UP-satisfiable* under some assumed literals A whenever there is some partial interpretation P which UP-satisfies F such that $A \subseteq P$. Assumming that a CNF formula F is UP-satisfiable under A, we can consider the *shared UP-model* of F under the assumed literals A, given by:

$$\mathcal{M}_A(F) = \bigcap \{P \mid A \subseteq P \text{ and } P \vDash_{\text{up}} F\}$$

We can then show the following result, adapted from [22]:

Theorem 2.1. *Let F be a CNF formula and A a set of assumed literals. Then, F is UP-satisfiable if and only if unit propagation over F and A does not fail. Furthermore, in the affirmative case the partial interpretation produced by unit propagation over F and A is precisely $\mathcal{M}_A(F)$.*

2.2 SAT Solvers

SAT solvers are decision procedures for the satisfiability problem on CNF formulas, i.e. whether an input formula F is satisfiable. In the affirmative case, a partial intepretation satisfying F is produced. In the negative case, an *unsatisfiability proof* of F is generated; unsatisfiability proofs are our main object of study, but we defer their discussion to a later point in this section. Both objects can be used as certificates for the correctness of the solving result.

The predominant paradigm in SAT solving is *conflict-driven clause learning* (CDCL) [23]. CDCL SAT solvers try to search a satisfying interpretation by maintaining a *trail* stack which contains temporarily assigned literals under some arbitrary assumed literals. Unit propagation is used to derive new implied literals under those assumptions; the trail contains the shared UP-model of the formula under the assumed literals. The trail keeps track as well of the clauses that were triggered in order to propagate each literal, called *reason clauses*. Whenever the assumptions are shown to be inconsistent, a *conflict* is derived, and some assumptions are undone in a process called *backtracking*. *Conflict analysis* methods allow to decide which assumptions must be backtracked. The efficiency of CDCL is greatly improved by the use of *watchlists* which keep track of the potentially triggered clauses [18]. A side effect of the use of watchlists is that the invariants required for efficiency are brittle, and literals may only be removed from the top of the trail stack. Clause learning is very efficient, which quickly leads to memory exhaustion; to prevent this, redundant clauses are routinely *deleted* from memory in a satisfiability-equivalent way [7]. Finally, *inprocessing techniques* are used as *ad hoc* methods to alleviate some of the shortcomings of CDCL by modifying the formula in satisfiability-equivalent ways, such as Gaussian elimination [24] and symmetry breaking [2].

2.3 DRAT Proofs

DRAT proofs are the *de facto* standard for the certification of solvers' unsatisfiability results. DRAT proofs are a record of the clause introductions deletions performed by the SAT solver; powerful redundancy criteria enable efficiently checking whether clauses were correctly introduced without significant overhead at solving time. The redundancy notions

DRAT proofs are based upon are called *reverse unit propagation* (RUP) [5, 6] and *resolution asymmetric tautology* (RAT) [10, 15].

- A clause C is a *RUP* clause in a CNF formula F whenever unit propagation over F and $\overline{C}$ fails, or equivalently, whenever F is UP-unsatisfiable under $\overline{C}$. RUP clauses are implied by the formula, i.e. $F \vDash C$.
- A clause C is a *RAT* clause in a CNF formula F upon a literal $l \in C$ whenever the clause $C \cup D \setminus \{\bar{l}\}$ is a RUP in F for each clause $D \in F$ with $\bar{l} \in D$. A clause C is a RAT in F if it is a RAT in F upon any literal $l \in C$. In this case, the clause C is not in general implied by F, but rather it is redundant: whenever F is satisfiable, so is $F \cup \{C\}$. Furthermore, the RAT property is non-monotonic: in general, given two formulas $F \subseteq G$ and a RAT clause C in F, we cannot conclude that a clause C is a RAT in G [20].

A *Deletion Resolution Asymmetric Tautology* (DRAT) proof, as formally defined by Heule in [13], is a sequence of *clause introductions* of the form $\mathbf{i}: C$ and *clause deletions* of the form $\mathbf{d}: C$. Given a DRAT proof π, we define its *accumulated formula* over a CNF formula F as follows:

- $\mathrm{acc}_F() = F$
- $\mathrm{acc}_F(\pi,\ \mathbf{i}: C) = \mathrm{acc}_F(\pi) \cup \{C\}$
- $\mathrm{acc}_F(\pi,\ \mathbf{d}: C) = \mathrm{acc}_F(\pi) \setminus \{C\}$

Intuitively, the accumulated formula at any point in the proof is computed by inserting every clause introduction and removing every clause deletion before that point in the proof. A DRAT proof π of a CNF formula F is correct whenever two conditions hold:

1. For each clause introduction where $\pi = \pi',\ \mathbf{i}: C,\ \pi''$, the clause C is either a RUP or a RAT in $\mathrm{acc}_F(\pi')$.
2. The CNF formula $\mathrm{acc}_F(\pi)$ is UP-unsatisfiable under $\emptyset$, or equivalently, whenever $\square$ is a RUP in $\mathrm{acc}_F(\pi)$.

In other words, a clause can only be introduced in a correct DRAT proof if it is either a RUP or a RAT in the accumulated formula before that clause introduction. Hence, clause introduction in a correct DRAT proof preserves satisfiability. Furthermore, there are no constraints for clause deletion, since deleting a clause from a formula is always satisfiability-preserving. The second condition above implies that the accumulated formula at the end of the proof is unsatisfiable, and since every instruction preserves satisfiability, then the initial formula F is unsatisfiable too.

DRAT proofs have become the standard in SAT solving due to several advantages over other proof formats. First and foremost, they are easy to generate for a CDCL SAT solver. Every learnt clause is automatically a RUP clause [1, 6], and no justification is required for deletions, so arbitrarily complex methods can be used for clause elimination without a proof generation overhead [7, 11]. Furthermore, no lower bounds are known for the proof complexity of the DRAT proof system, which in fact is as powerful as extended resolution [16]. Thanks to the RAT introduction criterion, DRAT proofs can express several inprocessing techniques that would lead to an exponential blow-up if expressed using only RUP clauses, such as Gaussian elimination and symmetry breaking [12, 19, 26, 27]. Last, DRAT checking can be performed efficiently, with a caveat we explain in Section 2.5. Since both RUP and RAT checks are based on unit propagation, DRAT checkers can be implemented efficiently with the same methods as in SAT solving.

2.4 DRAT Proof Checking

DRAT checkers implement several optimizations to improve the efficiency of the checking process. In practice, DRAT proofs can be very large and take even longer to check than to solve: for the Schur Number Five problem, solving took just over 14 CPU years whereas running the DRAT checker on the resulting proof took 20.5 CPU years [14]. Like SAT solvers, DRAT checkers implement a trail and the two-watched literal schema. We now note a few implementation optimizations that are relevant to our work.

A first point to observe is that, in practice, DRAT checkers classify proofs as correct under slightly less constringent criteria than provided in Section 2.3. In particular, DRAT checkers will consider a proof correct if they find a fragment of the proof that is correct; and consider a proof incorrect if it is found not to satisfy the requirements from Section 2.3. This means that depending on implementation details a proof might be considered both correct and incorrect. As counter-intuitive as this may sound, this is a feature rather than a bug: DRAT checkers avoid checking some unnecessary clause introductions for efficiency reasons. This can happen in two ways. First, if the accumulated formula at some point in the proof becomes UP-unsatisfiable under $\emptyset$, then one can skip checking the subsequent part of the proof. Second, if some introduced clauses are not transitively needed to derive $\square$ as a RUP, then they can be safely removed from the proof instead of spending time in checking them [9].

In order to check a DRAT proof, we need to check for every clause introduction $\mathbf{i}: C$, where F' is the accumulated formula before that instruction, whether C is a RUP or a RAT clause in F'. An implication of the previous paragraph is that F' is UP-satisfiable under $\emptyset$. Now consider the following result:

Theorem 2.2. *Consider a CNF formula G, and let us assume it is UP-satisfiable under $\emptyset$. Let us call $P = \mathcal{M}_\emptyset(G)$. Let Q be any partial interpretation. Then, G is UP-unsatisfiable under Q if either of the following hold:*

- *There is a literal l such that $l \in P$ and $\bar{l} \in Q$.*
- *G is UP-unsatisfiable under $P \cup Q$.*

DRAT checkers implicitly use this result to perform RUP and RAT checks, which in any case boil down to checking whether unit propagation over F' and a partial interpretation

trail state throughout π = **i**: x_5, **i**: x_9, **i**: $\square$					
start		after **i**: x_5		after **i**: x_9	
x_1:	x_1	x_1:	x_1	x_1:	x_1
x_2:	$\overline{x_1}x_2$	x_2:	$\overline{x_1}x_2$	x_2:	$\overline{x_1}x_2$
x_3:	$\overline{x_1x_2}x_3$	x_3:	$\overline{x_1x_2}x_3$	x_3:	$\overline{x_1x_2}x_3$
x_4:	$\overline{x_1x_3}x_4$	x_4:	$\overline{x_1x_3}x_4$	x_4:	$\overline{x_1x_3}x_4$
		x_5:	x_5	x_5:	x_5
		x_6:	$\overline{x_1x_5}x_6$	x_6:	$\overline{x_1x_5}x_6$
		x_7:	$\overline{x_2x_5}x_7$	x_7:	$\overline{x_2x_5}x_7$
		x_8:	$\overline{x_3x_6}x_8$	x_8:	$\overline{x_3x_6}x_8$
				x_9:	x_9
				x_{10}:	$\overline{x_4x_9}x_{10}$
				$x_{\overline{10}}$:	$\overline{x_7x_8x_9x_{10}}$

Figure 1. Trail evolution throughout the proof π from Example 2.3. Reason clauses for each propagated literal are indicated.

P fails. Instead, they check whether unit propagation over F' and $\mathcal{M}_\emptyset(F') \cup P$ fails. The gain is that instead of needing to propagate every literal in $\mathcal{M}_\emptyset(F') \cup P$, which can take a significant computation, they cache the propagated literals from $\mathcal{M}_\emptyset(F')$ in the trail, and simply perform the propagations needed after adding the literals in P as assumptions. When switching from one instruction to the next, the trail and the watchlists are updated accordingly to reflect $\mathcal{M}_\emptyset(F')$ in the new accumulated formula.

Example 2.3. Consider the CNF formula F containing the following clauses:

x_1	x_5x_6	$\overline{x_3x_6}x_8$	$\overline{x_4x_9}x_{10}$
$\overline{x_1}x_2$	$\overline{x_2x_5}x_7$	$\overline{x_6x_4}x_3$	$\overline{x_{10}}x_9$
$\overline{x_1x_2}x_3$	$\overline{x_1x_5}x_6$	$\overline{x_8}x_5$	$\overline{x_9}x_7$
$\overline{x_1x_3}x_4$	$\overline{x_5x_6}x_4$	$\overline{x_3}x_9x_{10}$	$\overline{x_7x_8x_9x_{10}}$

Let π be the correct DRAT proof **i**: x_5, **i**: x_9, **i**: $\square$. Figure 1 shows the shared-UP model, contained in the checker trail, throughout the proof, including the reason clauses for the corresponding propagations. In order to check if the instruction **i**: x_5 is a RUP, unit propagation can be performed over the accumulated formula and the partial intepretation $\{x_1, x_2, x_3, x_4, \overline{x_5}\}$ instead of simply over $\{\overline{x_5}\}$; the end result is the same, but as shown in Figure 1, this way most of the trail can be reused in subsequent checks. $\square$

State-of-the-art checkers perform two sweeps through the proof. The first sweep is called *incremental prepropagation* and the second sweep is called *backwards checking* [9]. For an in-detail discussion of these methods, we refer the reader to [22].

During incremental prepropagation, proof instructions are processed one by one in order to cache the propagated literals, i.e. the shared UP-model under the accumulated formula under $\emptyset$. This continues until a conflict is detected in the trail, which we call the *final contradiction*; this means that condition 2 for DRAT proof correctness holds for the proof fragment so far, and the subsequent instructions in the proof are discarded.

At this point backwards checking starts, proceeding towards the start of the proof. DRAT checkers are able to detect which clauses were required to derive a conflict in the trail, i.e. to find UP-unsatisfiability under $\emptyset$ of the accumulated formula. Such formulas are *marked* during RUP and RAT checks as well as in the final contradiction; unmarked clause introductions are not checked, since this means they are not required for the correctness of a subproof.

State-of-the-art DRAT checkers grow their trails *monotonically* throughout the proof, so it suffices a single trail to cache the corresponding propagated literals from $\mathcal{M}_\emptyset(F')$. For example, the trail in Example 2.3 only grows throughout the proof. More details on this issue are available in [21, 22].

2.5 Two Flavors of DRAT

In [22] a discrepancy between how the DRAT proof system is defined and how DRAT checkers work was found. DRAT checkers ignore a specific kind of clause deletions, called *unit deletions*. The result is not strictly incorrect: as shown in that paper, whenever DRAT checkers verify a proof of a CNF formula, that formula is guaranteed to be unsatisfiable. In fact, DRAT checkers can be formalized as operating under a different proof system. We will refer from now on to the previously presented proof system as *specified DRAT*. We now introduce the proof system that is actually checked by state-of-the-art checkers, called *operational DRAT*.

A clause C is called a *unit clause* under a partial interpretation P whenever $P(l) = 1$ for some literal $l \in C$, and $P(k) = 0$ for all $k \in C \setminus \{l\}$. By an abuse of notation, we say that C is a unit clause on a CNF formula F whenever either F is UP-unsatisfiable under $\emptyset$, or otherwise C is a unit clause under $\mathcal{M}_\emptyset(F)$. The operational DRAT proof system works exactly like specified DRAT, with the single exception that the accumulated formula is computed in an alternative way. In particular, $\mathrm{acc}_F(\pi,\ \mathbf{d}{:}\,C) = \mathrm{acc}_F(\pi)$ whenever C is a unit clause on $\mathrm{acc}_F(\pi)$; and $\mathrm{acc}_F(\pi,\ \mathbf{d}{:}\,C) = \mathrm{acc}_F(\pi) \setminus \{C\}$ as usual otherwise. In other words, unit deletions have no effect whatsoever in the proof.

Specified DRAT and operational DRAT are both sound and complete proof systems, but the classes of accepted proofs for each are incomparable. In [22], short proofs for each flavor are presented that are correct for one of the systems but incorrect for the other. Furthermore, such discrepancies occur with enormous frequency in practice: in [21] it was estimated that about half of the generated proofs during SAT solving competitions show a discrepancy.

The reasons why unit deletions are ignored are unknown to the authors, but as explained in [21], performing such deletions carelessly violates the watchlist invariants that allow for efficient unit propagation. In Section 2.4, we mention

that in state-of-the-art checkers, the trail grows monotonically throughout the proof. However, if a reason clause for a literal in the trail is deleted, the trail needs to shrink in order to satisfy the invariant that the trail reflects the shared UP-model of the accumulated formula. This would compromise the soundness of the checker, so instead checkers simply ignore unit deletions.

Recently, Rebola-Pardo and Cruz-Filipe presented an algorithm for specified DRAT checking [21]. While their implementation `rupee` showed a negligible overhead in specified mode over operational mode, it was not overall competitive, performing about an order of magnitude slower than state-of-the-art checkers such as `drat-trim`.

Example 2.4. Let us elaborate on Example 2.3. This time we consider a slightly different proof π' for the same proof F containing a unit deletion:

$$\mathbf{i}\colon x_5,\ \mathbf{d}\colon \overline{x_1}x_2,\ \mathbf{i}\colon x_9,\ \mathbf{i}\colon \square$$

Under operational DRAT, this proof is essentially the same as the proof π from Example 2.3. Under specified DRAT this proof is still correct, but the trail does not monotonically grow anymore, as shown in Figure 2. This violates many assumptions state-of-the-art checkers make in order to maintain their watchlists. □

trace preprocessing for $\pi = \mathbf{i}\colon \overline{\top} x_5,\ \mathbf{d}\colon \overline{x_1}x_2,\ \mathbf{i}\colon \overline{\top} x_9,\ \mathbf{i}\colon \square$

start		after $\mathbf{i}\colon \overline{\top} x_5$		after $\mathbf{d}\colon \overline{x_1}x_2$		after $\mathbf{i}\colon \overline{\top} x_9$	
x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$
x_2:	$\overline{x_1}x_2$	x_2:	$\overline{x_1}x_2$	x_5:	$\overline{\top} x_5$	x_5:	$\overline{\top} x_5$
x_3:	$\overline{x_1}x_2x_3$	x_3:	$\overline{x_1}x_2x_3$	x_6:	$\overline{x_1}\overline{x_5}x_6$	x_6:	$\overline{x_1}\overline{x_5}x_6$
x_4:	$\overline{x_1}x_3x_4$	x_4:	$\overline{x_1}x_3x_4$	x_4:	$\overline{x_5}\overline{x_6}x_4$	x_4:	$\overline{x_5}\overline{x_6}x_4$
		x_5:	$\overline{\top} x_5$	x_3:	$\overline{x_6}\overline{x_4}x_3$	x_3:	$\overline{x_6}\overline{x_4}x_3$
		x_6:	$\overline{x_1}\overline{x_5}x_6$	x_8:	$\overline{x_3}\overline{x_6}x_8$	x_8:	$\overline{x_3}\overline{x_6}x_8$
		x_7:	$\overline{x_2}\overline{x_5}x_7$			x_9:	$\overline{\top}\,\overline{x_9}$
		x_8:	$\overline{x_3}\overline{x_6}x_8$			x_7:	$\overline{x_9}x_7$
						x_{10}:	$\overline{x_4}\overline{x_9}x_{10}$
						$\overline{x_{10}}$:	$\overline{x_7}\overline{x_8}\overline{x_9}\overline{x_{10}}$

Figure 2. Trail evolution throughout the proof π' from Example 2.4. Reason clauses for each propagated literal are indicated.

3 Generating Proofs without Spurious Deletions

As explained in Section 2.5, many SAT solvers generate proofs that show different correctness status for each DRAT flavor. This is due to the solver inserting unit deletions that are later ignored by the checker. However, the discrepancy happens in a somewhat unexpected way: discrepant proofs are correct under operational DRAT, but incorrect under specified DRAT. In other words, some SAT solvers insert unit deletions in the proof that were never intended to be applied, and proof checking succeeds because unit deletions are ignored in state-of-the-art checkers. Our first goal is to understand how this happens, and to bring proof generation methods back to the overlapping fragment of both flavors, namely by preventing solvers from introducing spurious unit deletions.

To the best of our knowledge, all solvers affected by this issue are based on `MiniSat` [4] or `CryptoMiniSat` [24]. This issue traces back to the `DRUPMiniSat` patch[8], which was the first method to generate DRAT proofs in CDCL SAT solvers. A simplification technique in these solvers in the method `Solver::removeSatisfied` removes clauses C that are satisfied by the trail, and emits deletion instructions $\mathbf{d}\colon C$. While this is sound, it may delete clauses that are reason clauses for a literal in the trail, and these are always unit clauses; the literal is however kept in the trail, so this can be construed as implicitly keeping the clause in the formula but removing it from memory.

Example 3.1. Let us assume that the formula contains the clauses x and xy. In this case, the trail contains the literal x. During the simplification phase, both clauses will be removed by the solver, but the trail will still contain the literal x. Deletion instructions $\mathbf{d}\colon x$ and $\mathbf{d}\colon xy$ are then emitted to the proof, but the former is a unit deletion. Because the literal x is still on the trail, subsequent techniques applied by the solver will be performed as though the clause x remained in the formula. Hence, if the checker applies the deletion $\mathbf{d}\colon x$, it may not succeed in checking the proof instructions for those subsequent techniques; by ignoring that deletion, the checker matches the internal semantics in the solver. □

In this case, a fix is relatively simple: it suffices to prevent the emission of spurious unit deletions when calling the `Solver::removeSatisfied` method. We provide small and unintrusive patches to avoid said deletions for `MiniSat`[9] [10] as well as `MapleLCMDistChronoBT`, the winner of the main track of the 2018 SAT competition[11] [12]. After applying either of those patches, the generated proofs will be correct under both specified DRAT and operational DRAT, since they do not contain unit deletions. The patches can easily be adapted to other `MiniSat`-based solvers.

4 Checking Specified DRAT Proofs

The patches provided in Section 3 solve the issue of discrepancies in proofs generated by CDCL SAT solvers. By ensuring that generated proofs remain in the common fragment

[8] https://www.cs.utexas.edu/~marijn/drup/
[9] https://github.com/krobelus/minisat/commit/keep-locked-clauses
[10] https://github.com/krobelus/minisat/commit/add-unit-before-deleting-locked-clause
[11] https://github.com/krobelus/MapleLCMDistChronoBT/commit/keep-locked-clauses
[12] https://github.com/krobelus/MapleLCMDistChronoBT/commit/add-unit-before-deleting-locked-clause

of both DRAT flavors, one can use a DRAT checker for any flavor to check the proofs. Unfortunately, this might prove insufficient in the future. One of the most prominent advances in state-of-the-art SAT solving is the use of inprocessing techniques, as explained in Section 2.2. Proof generation for inprocessing techniques is substantially different from proof generation for CDCL solving.

Ad hoc proof generation methods are used for each different inprocessing technique, which generally work as follows. When the inprocessing technique comes up with a clause C that can be inserted in the formula F in a satisfiability-equivalent way, a proof generation procedure is called, which emits an essentially hardcoded proof fragment. The accumulated formula by the end of such proof fragment should be precisely $F \cup \{C\}$; the non-monotonic properties of RAT mentioned in Section 2.3 require that no extra clauses remain in the accumulated formula. However, proofs for inprocessing techniques routinely requires the derivation of many intermediate clauses, so the inserted proof fragment may look as follows:

$$\mathbf{i}\colon D_1,\ \mathbf{i}\colon D_2,\ \mathbf{i}\colon D_3,\ \mathbf{d}\colon D_1,\ \mathbf{i}\colon D_4,\ \mathbf{i}\colon C,\ \mathbf{d}\colon D_2,\ \mathbf{d}\colon D_3,\ \mathbf{d}\colon D_4$$

Observe that, under specified semantics, the total effect of this proof fragment is simply introducing the clause C. Now let us assume the clause D_1 is a unit clause. Then, under operational semantics the unit deletion $\mathbf{d}\colon D_1$ will never be applied, and so the clause D_1 will be spuriously introduced too. In model-preserving proof systems this would not be an issue, but satisfiability-preserving proof systems are non-monotonic. This means that the presence of some clauses may *disable* the proof system from deriving some clauses. In other words, if the proof later introduces a clause $\mathbf{i}\colon D'$ as a RAT clause, and this depends on the *absence* of D_1 in the accumulated formula, this proof would be incorrect under the operational semantics.

One possible solution to this issue would be to ensure that *ad hoc* proof fragments do not contain unit clauses. However, this would involve a substantial amount of work and processing. Instead, we consider that it is simpler to just modify checkers to competitively check specified DRAT. Our goal is to find out whether specified DRAT proofs can be checked with comparable efficiency to operational DRAT proofs.

We reimplemented the ideas from [21] in a novel DRAT checker `rate`[13] ("Rate Ain't Trustworthy Either"). We implement similar methods for specified DRAT checking to `rupee`. Furthermore, we use core-first propagation ideas from `drat-trim` and `gratgen` [17]. Our checker is able to check both specified and operational DRAT with state-of-the-art performance, as presented in Section 5. `rate` is also able to generate LRAT and GRAT correctness witnesses that can be verified by certified checkers [3, 17]. The details of specified DRAT checking algorithms are complex and fall beyond the scope of this work; we refer the reader to [21] for an overview.

5 Experimental Results

Here we present a performance evaluation of our checker. We analyze four checkers:

1. `rate`
2. `rate-d` (the flag `-d` means *"skip unit deletions"*)
3. `drat-trim` [29]
4. `gratgen` [17]

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. 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 in preliminary runs that the LRAT proof is verified by the formally verified checker `lrat-4` [14] [8]. 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 the final evaluation we also disabled assertions used to check invariants in the checker and logging in `rate` and `rate-d` which seems to give small speedups.

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 [25] 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[15] 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 pairs of solvers and 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

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

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

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

All benchmarks	3653
where the solver does not time out	3605
from which `rate` rejects the proof	2762
from which `rate` reports an error [17]	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

Figure 3. Split overview of benchmarks and results

benchmarking checker performance to include proofs that are rejected under specified DRAT.

We use as benchmarks all combinations of solvers and proofs benchmarks where the solver successfully produced a proof of unsatisfiability during the 2018 competition [16]. Figure 3 shows how many benchmarks were ignored for the performance evaluation using the criteria described above.

5.1 Comparison of Checkers

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 4 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 `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.

5.2 Unit Deletion Overhead

Handling unit deletions may require extra time and memory. `rate` works in such a way that the only unit deletions that create an overhead are deletions of clauses that occur as a reason clause for a literal in the trail. Figure 5 shows the number of reason deletions and the overhead of `rate` compared to `rate-d` — among our benchmarks runtime at most doubles.

6 Verifying Incorrectness Results

There remains a methodological issue with the experimental results from Section 5. From our benchmarks, only 843 out of 3605 proofs have not been rejected by `rate`. Given that `rate` is the first competitive specified DRAT checker, and that many proofs are rejected by `rate`, the possibility our checker contains bugs and proofs are incorrectly rejected

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

[17] `rate` and other DRAT checkers fail to verify this proof because it contains a number that exceeds the bounds of 32 bit integers

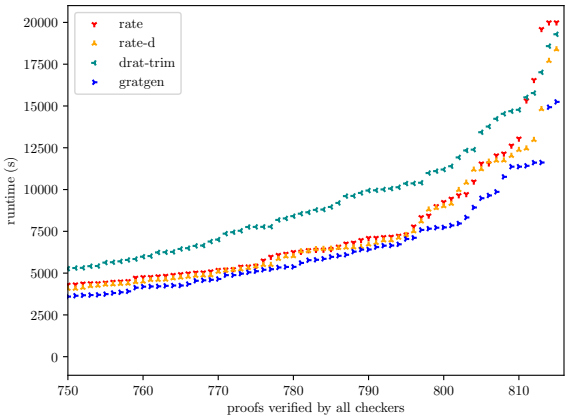

Figure 4. Cactus plot showing the distribution of checkers' runtime.

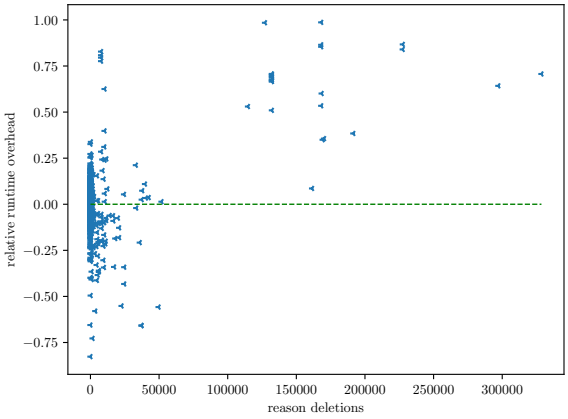

Figure 5. The number of reason deletions compared to the runtime overhead of checking specified DRAT over operational DRAT with `rate`.

by `rate` must be seriously considered. In order to guarantee that this is not the case, `rate` generates also *incorrectness certificates* that can be used to validate an incorrectness result by the checker.

The certificates are based on the idea that a DRAT proof is incorrect if any clause introduction is not a RUP or a RAT inference. To show that a clause introduction $\mathbf{i}: C$ is incorrect, i.e. not a RUP or a RAT in the accumulated formula F, we need to show the following conditions:

- C is not a RUP in F, or equivalently, the formula F is UP-unsatisfiable under $\overline{C}$.
- For each literal $l \in C$ there is some clause $D \in F$ such that $\overline{l} \in F$ and the clause $C \cup D \setminus \{\overline{l}\}$ is not a RUP in F, or equivalently, F is UP-unsatisfiable under $\overline{C} \cup \overline{D} \setminus \{l\}$.

Hence, certifying the incorrectness of a DRAT proof boils down to showing that F is UP-unsatisfiable under some partial interpretations. The properties of UP semantics presented in Section 2.1 make this relatively simple: it suffices to provide a partial interpretation P that UP-satisfies F, and that subsumes the corresponding partial interpretation. In particular, we need to give one *natural UP-model* P_0, as well as one *failing clause* $D_l \in F$ and one *failing UP-model* P_l for each $l \in C$, such that the following conditions are met:

1. $\overline{C} \subseteq P_0$
2. $P_0 \vDash_{\text{up}} F$
3. $\overline{C} \cup \overline{D_l} \setminus \{l\} \subseteq P_l$
4. $P_l \vDash_{\text{up}} F$

Since a checker already computes the shared UP-model for each RUP check, contained in the trail, generating this information upon failure is straightforward. Furthermore, UP semantics allow checking these properties with simple tools that require no unit propagation.

Example 6.1. Consider the formula $F = \{xy, \overline{x}y, x\overline{y}\}$ and the clause introduction $\mathbf{i}: \overline{x}\overline{y}$, which is neither RUP nor RAT in F. In this case we can take as the natural UP-model $P_0 = \{x, y\}$, as failing clauses $D_{\overline{x}} = x\overline{y}$ and $D_{\overline{y}} = \overline{x}y$, and as failing UP-models $P_{\overline{x}} = P_{\overline{y}} = P_0$. □

We propose the SICK format, which describes an incorrect clause introduction, as well as the natural UP-model, the failing clauses, and the failing UP-models that certify its incorrectness; the grammar for the SICK format is shown in Figure 6, and an example is given in Figure 7.

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 clause introduction in the proof.

- `proof_step` specifies the proof step $\mathbf{i}: C$ that failed (by offset in the proof, starting at one for the first proof step). If `proof_step` is omitted, it means that the proof does not add enough clauses to make the accumulated formula UP-unsatisifiable under $\emptyset$.
- `natural_model` gives the natural UP-model P_0. If `proof_step` is omitted, this is the shared UP-model after applying all proof steps.
- Each `witness` comprises the following elements:
  - `pivot` specifies the literal $l \in C$.
  - `failing-clause` specifies the failing clause D_l.
  - `failing-model` specifies the additional literals that, together with P_0, make up the failing UP-model P_l.

A SICK certificate produced by `rate` rejecting a proof can be used by our tool `sick-check`[18] to again verify incorrectness of the proof. The certificate is tiny compared to the formula, and checking it is very fast to the extent that parsing usually takes up most of `sick-check`'s runtime. We successfully checked the 2762 rejected proofs using `sick-check`.

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

```
SICK           := Header Witness*
Header         := [ProofStep] NaturalModel
ProofStep      := proof_step = Integer
NaturalModel   := natural_model = ListOfLiterals
Witness        := [[witness]] FailClause FailModel Pivot
FailingClause  := failing_clause = ListOfLiterals
FailingModel   := failing_model = ListOfLiterals
Pivot          := pivot = Literal
ListOfLiterals := [ (Literal ,)* ]
```

Figure 6. The grammar of a SICK certificate

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

Figure 7. Example SICK certificate for an incorrect proof

7 Conclusions

DRAT proofs are the main way to certify SAT solvers' unsatisfiability results. However, there exists a discrepancy between the originally declared DRAT proof system and the operationally used DRAT proof system due to unit deletions. This led to an interdependency issue where SAT solvers generate DRAT proofs containing spurious unit deletions, and DRAT checkers ignore these unit deletions. We propose a two-sided fix to this issue.

On the one hand, in Section 3 we explain how state-of-the-art SAT solvers generate proofs that can only be reasonably verified when their unit deletions are ignored, despite the lack of a reason to do so. We present lightweight patches for affected solvers to avoid unit deletions in all proofs generated so far. These changes makes their proofs correct under either flavor of DRAT, paving the way for a possible adoption of specified DRAT.

On the other hand, unit deletions may legitimately occur in DRAT proofs. As explained in Section 4, advanced inprocessing techniques may emit unit deletions. We implement the first competitive proof checker that can handle unit deletions in proofs efficiently. In Section 5 we present experimental results that demonstrate that our checker is competitive compared to other state-of-the-art DRAT checkers. We also conclude that around three quarters of all proofs

produced by solvers at the 2018 SAT competition are incorrect under specified DRAT. Additionally we give evidence to the claim that checking specified DRAT is as expensive as checking operational DRAT on the average instance, though there may be a significant overhead for proofs with a huge number of unit deletions.

Since many proofs in our benchmarks are rejected by our checker, we were wary of any bugs in our tool. We introduce the SICK certificate format in 6, which describes a small and efficiently checkable witness of the incorrectness of a proof. Such witnesses can be of help when debugging SAT solvers and DRAT checkers. Finally, we provide a simple tool to check SICK certificates, independent of a usually much more complex DRAT checker.

Acknowledgments

This work was funded through the LogiCS doctoral program W1255-N23 of the Austrian Science Fund (FWF), the Vienna Science and Technology Fund (WWTF) through grant VRG11-005, and Microsoft Research through its PhD Scholarship Programme.

References

[1] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22(1):319–351, December 2004.

[2] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning (KR)*, pages 148–159. Morgan Kaufmann, 1996.

[3] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *TACAS*, volume 10205 of *LNCS*, pages 118–135. Springer, 2017.

[4] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, SAT 2003.*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[5] 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.

[6] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 886–891. IEEE, 2003.

[7] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6397 of *LNCS*, pages 357–371. Springer, 2010.

[8] 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 *LNCS*, pages 269–284. Springer, 2017.

[9] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013.

[10] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *CADE*, volume 7898 of *LNCS*, pages 345–359. Springer, 2013.

[11] 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.

[12] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *CADE-25*, pages 591–606, 2015.

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

[14] 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, New Orleans, Louisiana, USA, February 2-7, 2018*. AAAI Press, 2018.

[15] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.

[16] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *LNCS*, pages 516–531. Springer, 2018.

[17] 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*, volume 10491 of *LNCS*, pages 457–463. Springer, 2017.

[18] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535. ACM, 2001.

[19] Tobias Philipp and Adrián Rebola-Pardo. DRAT proofs for XOR reasoning. In *JELIA*, pages 415–429, 2016.

[20] Tobias Philipp and Adrián Rebola-Pardo. Towards a semantics of unsatisfiability proofs with inprocessing. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 65–84. EasyChair, 2017.

[21] Adrián 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.

[22] AdriÃąn Rebola-Pardo and Armin Biere. Two flavors of DRAT. EasyChair Preprint no. 457, EasyChair, 2018.

[23] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[24] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, volume 5584 of *LNCS*, pages 244–257. Springer, 2009.

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

[26] Alasdair Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

[27] Alasdair Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96-97:177–193, 1999.

[28] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.

[29] 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 *LNCS*, pages 422–429. Springer, 2014.