EPiC Series in Computing

Volume 59, 2019, Pages 94–110

Proceedings of Pragmatics of SAT 2015 and 2018

Two Flavors of DRAT

Adrián Rebola-Pardo[1] and Armin Biere[2]

[1] TU Wien
[2] JKU Linz

Abstract

DRAT proofs have become the de facto standard for certifying SAT solvers' results. State-of-the-art DRAT checkers are able to efficiently establish the unsatisfiability of a formula. However, DRAT checking requires unit propagation, and so it is computationally non-trivial. Due to design decisions in the development of early DRAT checkers, the class of proofs accepted by state-of-the-art DRAT checkers differs from the class of proofs accepted by the original definition. In this paper, we formalize the operational definition of DRAT proofs, and discuss practical implications of this difference for generating as well as checking DRAT proofs. We also show that these theoretical differences have the potential to affect whether some proofs generated in practice by SAT solvers are correct or not.

1 Prelude

DRAT proofs are sequences of clause additions and clause deletions. Given a formula F, some DRAT proofs are called *correct refutations of* F; for now we can spare ourselves the details on what correct refutations are, but let us just require that in that case F is unsatisfiable. Further consider an arbitrary transformation Φ on DRAT proofs (not necessarily correctness-preserving). Then the following theorem trivially holds:

Theorem 1. *Let F be a CNF formula, and π a DRAT proof. If $\Phi(\pi)$ is a correct refutation of F, then F is unsatisfiable.*

Observe that whether π itself is correct or not is irrelevant. An interesting consequence of this result is that, if we implement a proof checking method that first performs some modifications on the input proof, and then checks the resulting proof to be correct, such a procedure will be sound: whenever an input π is accepted, then some correct refutation of F exists (namely the transform of π). However, it is possible that the class of accepted proofs by this procedure is different from the class of correct DRAT refutations. One can construe this procedure as implicitly defining a *different* proof system.

2 Introduction

The spectacular leap in SAT solvers' efficiency over the last two decades came with an increase in the code complexity of solvers. These programs use many techniques, which can interact in

unexpected ways; and convoluted data structures, which rely on mutability and so are difficult to write in a correct-by-construction way. The main approach to ensure reliability of solving results is to check individually each result. Wrong satisfiability results are easy to detect: SAT solvers provide a model of the formula, and this can be independently verified in linear time.

Wrong unsatisfiability results are harder to recognize. Since 2003, several proof systems of increasing expressive power have been developed [8, 29, 11, 11, 12, 28, 17]. Each of them is based on the same idea: providing a small set of inferences which can be efficiently checked, and simulating the execution of the solver with these rules, logging them into an unsatisfiability proof. A proof checker would then independently verify that each individual inference is correct [14, 19].

Delete Resolution Asymmetric Tautology (DRAT) is the current standard format for unsatisfiability proofs [14], although an extension called Delete Propagation Redundant (DPR) has been proposed [17]. Many SAT solvers are able to produce DRAT proofs, to the extent that this is now a requirement in SAT Competitions [3]. Due to their expressive power, generating a DRAT proof during the execution of a CDCL solver is relatively easy: it suffices to log the clauses infered or removed by the solver [8]. When satisfiability-preserving techniques are used to introduce or remove clauses, DRAT allows to express these clause introductions and deletions [23, 21, 26, 13]. Furthermore, inferences allowed in DRAT are based on unit propagation, which allows very efficient proof checking by using sophisticated data structures adapted from SAT solving [22]. Since some proof search is necessary for unit propagation, deletion information greatly helps in proof checking; otherwise, lemmas that are no longer necessary would accumulate, and too much time would be spent in proof search [28].

However, deletion information introduces an issue in unit propagation: when a clause is deleted, the set of literals implied by unit propagation from the current formula may change. Invariants maintained by lazy data structures [22] often used in DRAT checking are broken if these literals are forgotten.

The approach taken by state-of-the-art DRAT checkers is to ignore deletion instructions for clauses that can potentially change the set of literals implied by unit propagation, called *unit clauses* [14]. However, inferences in DRAT are only satisfiability-preserving, thus non-monotonic [24]. In particular, this means that correct proofs can be rejected by DRAT checkers and vice versa; this issue has been documented and acknowledged in the literature [14, 7].

In this paper, we analyze this issue of unit deletion in detail. Our goal is threefold. First, we clarify why this method for DRAT checking is *sound*, in the sense that no refutation for a satisfiable instance would ever be accepted (modulo implementation errors). Second, we formalize what exactly the class of proofs accepted by state-of-the-art DRAT checkers is, and provide easily checkable evidence that it is incomparable to the class of DRAT proofs as defined originally. With this formalization, we can refer to two *flavors* of DRAT: one corresponding to the definition from previous literature, and another one accounting for the pragmatics of DRAT checking. Last, we aim to spark a discussion on whether one flavor or the other should be taken as a standard. We believe that efficiently checking DRAT proofs according to the original definition is possible. We discuss possible problems arising from the use of either flavor, and we would like to gauge the SAT community's interest on this issue.

3 Preliminaries

Given a variable x, we denote its *complement* by $\overline{x}$. A *literal* is a variable or its complement. For a set of literals L, we denote by $\overline{L}$ the set of its complements. A *clause* is a disjunction of literals; we denote clauses by juxtaposition, i.e. $xy\overline{z}$ is the clause usually denoted by $x \vee y \vee \overline{z}$. We

assume that clauses do not contain complementary literals. The *unsatisfiable* or *empty* clause is the clause containing no literals, which we denote by $\square$. A *CNF formula* is a conjunction of clauses; we construe CNF formulas as finite sets of clauses. For a clause C, we denote by $\overline{C}$ the set of clauses containing the size-one clause $\bar{l}$ for each literal $l \in C$. A *partial assignment* is a set of literals I which does not contain complementary literals. For any literal l, we define $I(l)$ as follows: $I(l) = 1$ if $l \in I$; $I(l) = 0$ if $\bar{l} \in I$; and $I(l) = ?$ otherwise. A clause C is called *unit* w.r.t. a partial assignment I whenever there is a literal $l \in C$ with $I(l) = 1$, and for any other literal $k \in C \setminus \{l\}$ we have $I(k) = 0$. We follow the usual definitions of *satisfiability* and *entailment* of CNF formulas or clauses.

Let F be a CNF formula. We say that a literal l is implied by unit propagation over F whenever there is a finite sequence $l_1, \ldots, l_n$ of literals such that $l_n = l$, and for each $1 \leq i \leq n$ we can find a clause $C_i \in F$ with $l_i \in C_i$ and $C_i \setminus \{l_i\} \subseteq \{\overline{l_1}, \ldots, \overline{l_{i-1}}\}$. Furthermore, we say that unit propagation over F leads to contradiction whenever there are two complementary literals l and $\bar{l}$ implied by unit propagation over F. A clause C is a *reverse unit propagation* (RUP) clause in F whenever unit propagation over $F \cup \overline{C}$ leads to contradiction [8]. Furthermore, C is called a *resolution asymmetric tautology* (RAT) in F upon a literal $l \in C$ whenever the clause $C \vee (D \setminus \{\bar{l}\})$ is a RUP in F, for every clause $D \in F$ with $\bar{l} \in D$ [10].

It has been shown that if C is a RUP in F, then C is a semantic consequence of F [4]. Furthermore, if C is a RAT in F upon l, then F is satisfiability-equivalent to $F \cup \{C\}$, although C needs *not* be a consequence of F [10]. Whereas the property of being a RUP is monotonic (i.e. if C is a RUP in F, then C is also a RUP in $F \cup G$), this is not the case for RATs. For example, the clause x is a RAT in the empty formula, but not in the formula $\{\bar{x}\}$; for more details on non-monotonicity of RATs we refer the reader to [24].

Modern SAT solvers are able to generate unsatisfiability certificates called *DRAT proofs*, which record the changes made to the input formula by the solver in an incremental way [14]. A DRAT proof π is a string of instructions $i_1, \ldots, i_n$; every instruction is either a *clause introduction instruction* $\mathbf{i}\colon C$ or a *clause deletion instruction* $\mathbf{d}\colon C$, where C is a clause. Given a DRAT proof π and a CNF formula F, we recursively define the *accumulated formula* $F[\pi]$ by F through π as follows:

$$F[\epsilon] = F \qquad F[\pi\,, \mathbf{i}\colon C] = F[\pi] \cup \{C\} \qquad F[\pi\,, \mathbf{d}\colon C] = F[\pi] \setminus \{C\}$$

Given a CNF formula F, a DRAT proof $i_1, \ldots, i_n$ is called a *correct DRAT proof* from F whenever either of the following holds for every $1 \leq j \leq n$:

- i_j is a deletion instruction $\mathbf{d}\colon C$.
- i_j is an introduction instruction $\mathbf{i}\colon C$, and C is either a RUP in $F[i_1, \ldots, i_{j-1}]$, or a RAT in the same formula upon the first literal l in C.

Furthermore, it is called a correct *refutation* of F if additionally $i_m = \mathbf{i}\colon \square$ for some $1 \leq m \leq n$. Since clause deletion and RUP introduction are truth-preserving, and RAT introduction is satisfiability-preserving, we can only derive the empty clause by a correct DRAT proof from F whenever F is unsatisfiable. In other words: the DRAT proof system is sound, insofar as correct refutations of F only exist for unsatisfiable formulas F.

Example 1. Consider the CNF formula and DRAT proofs:

$$F = \{xyz,\ \bar{x}yz,\ x\bar{y}z,\ \bar{x}\bar{y}z,\ xy\bar{z},\ \bar{x}y\bar{z},\ x\bar{y}\bar{z},\ \bar{x}\bar{y}\bar{z}\}$$
$$\pi = \mathbf{i}\colon xy,\ \mathbf{i}\colon x,\ \mathbf{i}\colon w\bar{x},\ \mathbf{d}\colon w\bar{x},\ \mathbf{i}\colon \bar{w}\bar{x},\ \mathbf{i}\colon wx,\ \mathbf{i}\colon yw,\ \mathbf{i}\colon \square$$
$$\sigma = \mathbf{i}\colon xy,\ \mathbf{i}\colon x,\ \mathbf{d}\colon x,\ \mathbf{i}\colon w\bar{y},\ \mathbf{i}\colon \bar{w}\bar{y},\ \mathbf{i}\colon w,\ \mathbf{i}\colon \square$$

The DRAT proof π is a correct refutation of F; Figure 1 contains a justification for each of the inferences. In contrast, σ is an incorrect DRAT refutation of F. Consider the instruction $\mathbf{i}\colon \bar{w}\bar{y}$. The CNF formula accumulated before this instruction is $F' = F \cup \{xy,\, w\bar{y}\}$. The clause $\bar{w}\bar{y}$ is not a RUP in F', since no unit propagation is available in the formula $F' \cup \{w,\, y\}$. Furthermore, it is not a RAT in F' upon its first literal, namely $\bar{w}$: the only clause containing w in F' is $w\bar{y}$, and the clause $(\bar{w}\bar{y}) \vee (w\bar{y}) \setminus \{w\} = \bar{w}\bar{y}$ is not a RUP in F', for the same reason as above.

instruction	*previous accumulated formula*	*justification*
$\mathbf{i}\colon xy$	F	RUP: $\bar{x}\bar{y} \xrightarrow{xyz} z \xrightarrow{xy\bar{z}} \bot$
$\mathbf{i}\colon x$	$F \cup \{xy\}$	RUP: $\bar{x} \xrightarrow{xy} y \xrightarrow{x\bar{y}z} z \xrightarrow{x\bar{y}\bar{z}} \bot$
$\mathbf{i}\colon w\bar{x}$	$F \cup \{xy,\, x\}$	RAT upon w: no clauses with $\bar{w}$ in the accumulated formula
$\mathbf{d}\colon w\bar{x}$	$F \cup \{xy,\, x,\, w\bar{x}\}$	
$\mathbf{i}\colon \bar{w}\bar{x}$	$F \cup \{xy,\, x\}$	RAT upon $\bar{w}$: no clauses with $\bar{w}$ in the accumulated formula
$\mathbf{i}\colon wx$	$F \cup \{xy,\, x,\, \bar{w}\bar{x}\}$	RAT upon w: clause $(wx) \vee (\bar{w}\bar{x}) \setminus \{\bar{w}\} = (wx\bar{x})$ is a tautology, hence a RUP.
$\mathbf{i}\colon yw$	$F \cup \{xy,\, x,\, \bar{w}\bar{x},\, wx\}$	RUP: $\bar{y}\bar{w} \xrightarrow{x} x \xrightarrow{\bar{x}yz} z \xrightarrow{\bar{x}y\bar{z}} \bot$
$\mathbf{i}\colon \square$	$F \cup \{xy,\, x,\, \bar{w}\bar{x},\, wx,\, yw\}$	RUP: $\top \xrightarrow{x} x \xrightarrow{\bar{w}\bar{x}} \bar{w} \xrightarrow{yw} y \xrightarrow{\bar{x}\bar{y}z} z \xrightarrow{\bar{x}\bar{y}\bar{z}} \bot$

Figure 1: Each inference in the proof π from Example 1 is justified above. Arrows indicate unit propagation, and clauses above the arrows indicate the clause triggering each propagation.

3.1 DRAT proof checking

State-of-the-art DRAT proof checkers [14, 19] are complex programs, albeit significantly simpler than SAT solvers (e.g. the commonly used DRAT checker `DRAT-trim` currently contains 1600 lines of C code). Much of this complexity is due to the implementation of efficient unit propagation through the two-watched literal schema [22]. Unfortunately, the latter intensively relies on mutable data structures and pointer arithmetics, and these are hard to simulate with immutable data structures typical from correct-by-construction implementations. In addition to fast unit propagation, two improvements are crucial to efficient DRAT checking, namely incremental prepropagation and backwards checking [11].

Incremental prepropagation By adapting ideas from CDCL SAT solvers [22], a DRAT checker can save a good amount of work when computing if a clause is a RAT in the accumulated formula. Literals that follow from the accumulated formula F at the i-th instruction are stored in a stack, much in the way as zero-decision level literals are stored in SAT solvers. To test whether a clause C is a RUP in F, the literals from $\bar{C}$ are added to the stack as assigned but unprocessed literals. Then, unit propagation on yet unprocessed literals is performed. The literals from $\bar{C}$ are analogous to decisions in CDCL SAT solvers, although DRAT checkers

introduce these assumptions at once and only later perform unit propagation; the end result is essentially equivalent.

At the end of this process the stack has a prefix of prepropagated literals, followed by the literals in $\bar{C}$, and by a suffix of additional propagated literals. Now, discarding the whole stack would be wasteful: the prefix can still be used for later RUP checks. Instead, DRAT checkers only pop the two latter parts of the stack, and the prefix of prepropagated literals remains in the stack. Analogously to backtracking in CDCL, this does not harm the invariants preserved by the two-watched literal schema, as long as all literals that could be prepropagated from F were in the first part of the stack.

In order to continue checking the DRAT proof, the clause C needs to be added to the clause database, which may make further literals available for propagation. The checker is able to find those literals, which are then pushed into the prepropagated stack, which is then ready to check the next instruction.

Backwards checking An interesting observation is that, because of the "blind" way on which SAT solvers perform search, many derived clauses are not involved in the derivation of the empty clause. Checking these inferences is useless, and skipping them improves the efficiency of the proof checker. To do this, the proof can be checked *backwards* [11]. Starting from the empty clause at the end of the proof, every inference is checked. A DRAT checker is able to detect which premise clauses are necessary to derive the goal clause; these clauses are scheduled for later checking. If the checker reads a clause in the proof which is not scheduled, it is simply skipped.

Incremental prepropagation and backwards checking may seem incompatible: the former proceeds forward, whereas the latter does so backwards. However, one can take advantage of the monotonic way in which incremental propagation pushes literals in the stack to pop them back in the same order during backwards propagation. Proof checking proceeds in two sweeps through the proof. The proof is firstly sweeped forward performing incremental prepropragation, and nothing is checked at this stage. Then, the proof is checked backwards, skippping any unscheduled instruction. During backwards checking, the prepropragated stack needs to be updated, but since literals in the prepropagation stack are only stacked and never popped during incremental prepropagation, updating is as simple as popping a segment of the stack.

Example 2. Consider the unsatisfiable CNF formula F containing clauses:

$$\begin{array}{lllllll} x_1 & \overline{x_1}\,\overline{x_2}x_3 & x_5x_6 & \overline{x_1}\,\overline{x_5}x_6 & \overline{x_3}\,\overline{x_6}x_8 & \overline{x_8}x_5 & \overline{x_4}\,\overline{x_9}x_{10} & \overline{x_9}x_7 \\ \overline{x_1}x_2 & \overline{x_1}\,\overline{x_3}x_4 & \overline{x_2}\,\overline{x_5}x_7 & \overline{x_5}\,\overline{x_6}x_4 & \overline{x_6}\,\overline{x_4}x_3 & \overline{x_3}x_9x_{10} & \overline{x_{10}}x_9 & \overline{x_7}\,\overline{x_8}\,\overline{x_9}\,\overline{x_{10}} \end{array}$$

and the correct DRAT refutation of F given by $\tau = \mathbf{i}\colon x_5,\ \mathbf{i}\colon x_4,\ \mathbf{i}\colon x_9,\ \mathbf{i}\colon \square$. We now recreate the checking process in a DRAT checker. The first sweep proceeds forwards, computing the prepropagated stack by unit propagation:

- Before the $\mathbf{i}\colon x_5$ instruction, the prepropagated stack is x_1, x_2, x_3, x_4, which are propagated using clauses x_1, $\overline{x_1}x_2$, $\overline{x_1}\,\overline{x_2}x_3$, and $\overline{x_1}\,\overline{x_3}x_4$, respectively.
- Before the $\mathbf{i}\colon x_4$ instruction, the clause x_5 is introduced in the formula. The prepropagated stack is x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, where the extra literals are propagated using clauses x_5, $\overline{x_1}\,\overline{x_5}x_6$, $\overline{x_2}\,\overline{x_5}x_7$, and $\overline{x_3}\,\overline{x_6}x_8$, respectively.
- Before the $\mathbf{i}\colon x_9$ instruction, the clause x_4 is introduced in the formula. This clause leads to no new propagations, so the prepropagated stack remains as in the last step.

- Before the $\mathbf{i}\colon \square$ instruction, the clause x_9 is introduced in the formula. The prepropagated stack is x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, $\overline{x_{10}}$, where the extra literals are propagated using clauses x_9, $\overline{x_4}x_9x_{10}$, $\overline{x_7}x_8x_9\overline{x_{10}}$ respectively.

At this point, a conflict in the stack is detected due to complementary literals x_{10} and $\overline{x_{10}}$, and backwards checking starts. Originally, the only clause scheduled for verification is $\square$.

- The clause $\square$ is a RUP in $F \cup \{x_5, x_4, x_9\}$ as shown by the last prepropagated stack. The non-premise clauses used to derive a conflict are x_5 and x_9, so the instructions $\mathbf{i}\colon x_5$ and $\mathbf{i}\colon x_9$ are scheduled for verification.

- The checker must now show that x_9 is a RUP in $F \cup \{x_5, x_4\}$. Since x_9 is removed from the formula, the literals it led to propagate must be removed from the prepropagation stack. This is easy, since they were the last to be introduced: the prepropagated stack at this point should be x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, and this can be obtained by popping all literals until x_9. Now, to check that x_9 is a RUP in $F \cup \{x_5, x_4\}$, the checker pushes $\overline{x_9}$ in the stack and performs unit propagation. This leads to the conflicting stack x_1, x_2, x_3, x_4, x_5, x_6, x_7, $x_8, \overline{x_9}, \overline{x_{10}}, \overline{x_3}$, where clauses $x_9\overline{x_{10}}$ and $\overline{x_3}x_9x_{10}$ are used in propagation. Hence, the instruction $\mathbf{i}\colon x_9$ is verified, and no new clauses are scheduled for verification.

- The instruction $\mathbf{i}\colon x_4$ is not scheduled for verification, so it can be skipped. Furthermore, it led to no prepropagation, so there is no need to perform changes over the prepropagation stack.

- The checker must finally check that x_5 is a RUP in F. By the same procedure as above, the prepropagated stack x_1, x_2, x_3, x_4 is restored. The literal $\overline{x_5}$ is pushed in the stack, and unit propagation yields the stack x_1, x_2, x_3, x_4, $\overline{x_5}$, x_6, $\overline{x_8}$, x_8, where clauses x_5x_6, $x_5\overline{x_8}$, $\overline{x_3}\overline{x_6}x_8$ are used for propagation. The instruction $\mathbf{i}\colon x_5$ is then verified, and no new clauses are scheduled for verification.

At this point, the checker reaches the start of the proof, and it is reported as correct.

Unit clause deletion The paragraphs above incur into a small simplification: deleting a clause can provoke that some literals which were implied by unit propagation stop being so. This is problematic, because propagation of other literals may depend on that one, and it is not clear how would one update the stack when a deletion instruction is processed. As discussed above, the invariants for backwards checking depend on the monotonicity of the stack, and so do the invariants for the two-watched literal schema [22]. Furthermore, simply discarding the whole stack and recompute it from scratch would defeat the purpose of incremental prepropagation.

Fortunately, there is an easily checked necessary condition for a clause deletion to affect the stack in this way, namely that the clause is a unit w.r.t. the partial assignment defined by the stack. State-of-the-art checkers take a pragmatic approach: unit clause deletion instructions are ignored [14].

Example 3. Consider the correct DRAT refutation $\tau' = \mathbf{i}\colon x_5$, $\mathbf{d}\colon \overline{x_1}x_2$, $\mathbf{i}\colon x_9$, $\mathbf{i}\colon \square$ of the formula F from Example 2. If one were to take the unit clause deletion $\mathbf{d}\colon \overline{x_1}x_2$ into account, several problems arise during checking. For one, the prepropagated stack before this instruction is x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, and after the instruction is x_1, x_5, x_6, x_4, x_3, x_8; it is not clear at all how to obtain the latter from the former. Even worse, during backwards checking one would need to restore the former stack from the latter: the previously helpful method of popping

the last elements in the stack does not work anymore, and we would even need to introduce literals x_2 and x_7.

Furthermore, there is another issue regarding the two-watched literal schema, which maintains two *watched* literals in each clause. For its lazy data structures to work, it is necessary that if a watched literal l in a clause is falsified by the current stack, the other watched literal k must be satisfied. Consider now the clause $\overline{x_2}\,\overline{x_5}x_7$, and assume literals $\overline{x_5}$ and x_7 are watched before applying the deletion $\mathbf{d}\colon \overline{x_1}x_2$. This is a correct watch choice before the deletion, but after it the literal $\overline{x_5}$ is falsified and x_7 is not satisfied by the corresponding stack. Violating this condition leads the unit propagation engine to "miss" propagations, and so some RUPs will not be detected as such.

4 What DRAT checkers are really checking

The DRAT proof system was designed so that it would be tightly adapted to the operation of SAT solvers. Several desired features were:

- Soundness, of course. Whenever an input DRAT proof π is reported a correct refutation of a CNF formula F, the latter should be unsatisfiable. In principle, this is ensured by the fact that the three inferences (clause deletion, RUP introduction, and RAT introduction) are satisfiability-preserving [10, 18]. Hence, whenever the empty clause is derived, the input CNF formula is unsatisfiable. Later in this section we will see the whole picture is a bit more complex, but even then soundness holds.

- Proofs should be easily generated by SAT solvers. For CDCL solvers, it is very simple to generate a DRAT proof: every clause learned by CDCL is a RUP [4]. Hence, it suffices to log the list of learned clauses to produce a DRAT proof [8]. Clause elimination is unproblematic too, since there is no restriction on which clauses can be deleted. Many inprocessing techniques, including bounded variable addition [20], self-subsuming resolution [9], and variable elimination [6], can be easily simulated in DRAT. However, for a few techniques, notably parity reasoning [27] and symmetry breaking [1], it is notably more complex to generate a DRAT proof [23, 13]. In this case, whole proof fragments are generated *ad hoc* by hard-coded procedures to derive the introduced clauses with minimal reasoning effort.

- Proofs should be efficiently checkable with significantly simpler programs than SAT solvers. Although DRAT checkers are not that simple, they are just optimized unit propagation engines. Furthermore, DRAT checkers are able to enrich the proof so that it can be validated with certified software in reasonable time [5, 19].

In complying with these constraints, state-of-the-art DRAT checkers slightly deviate from the original definition of correct DRAT refutations. Formulas are regarded by DRAT checkers as multisets of clauses; we show in a moment that there are compelling reasons for this. Furthermore, deletion instructions of unit clauses w.r.t. the current stack are ignored, as explained in Section 3.1. The rest of the paper is devoted to discuss this issue in depth.

Formulas as multisets of clauses Let us assume that a solver introduces clause C in the CNF formula F by some complex inprocessing technique. A proof generation procedure is then called, which generates a (hopefully correct) DRAT proof fragment π with the property that $F[\pi] = F \cup \{C\}$. It is important that the accumulated formula after π corresponds to

the formula in the SAT solver: RAT introduction is non-monotonic, and so it also depends on the absence of clauses. Having different clause sets can make future RAT introductions incorrect. In some circumstances (e.g. parity reasoning [23], and symmetry breaking [13]) the proof π is rather long and introduces many intermediate clauses that are necessary to derive C. These clauses do not occur in $F \cup \{C\}$, and so they need to be deleted later in π to avoid the aforementioned problems; this is an issue addressed in work on proofs for parity reasoning [25].

With the definition of DRAT proofs as given originally, this subtly leads to unnecessary work at proof generation time. Assume that the proof π above introduces an intermediate clause D, which then needs to be deleted. The proof thus looks as follows:

$$\pi = \ldots, \mathbf{i}\colon D, \ldots, \mathbf{d}\colon D, \ldots, \mathbf{i}\colon C, \ldots$$

where we can assume that D and C do not occur anywhere else in π. Observe now that, if $D \in F$ already, then first $\mathbf{i}\colon D$ is redundant, but much worse we have $D \notin F[\pi]$ because D gets deleted by the instruction $\mathbf{d}\colon D$. Accordingly the condition $F[\pi] = F \cup \{C\}$ fails to hold. A naïve way to solve this would be to check at generation time if introduced clauses already occurred in F, and insert the deletion instruction only when they did not. This is inconvenient: doing so efficiently requires additional data structures, which in turn need maintenance every time the clause set is modified, hence introducing undesired solving time overhead.

By treating formulas as multisets of clauses, this issue is resolved. Every deletion removes exactly one copy of a clause, and every introduction inserts exactly one copy. Hence, by matching the number of introductions and deletions in the proof generation procedure, one can ensure that the accumulated formula by the proof corresponds to the working clause set in the solver. Multiset clause introduction of RUP or RAT clauses, as well as multiset clause deletion, are trivially satisfiability-preserving, since the underlying clause set is either preserved by this operations or changed in the traditional way. Using multiset semantics is also important for compositional proofs and parallel proof checking [15].

Ignoring unit clause deletions As explained in Section 3.1, deletion instructions of unit clauses w.r.t. the current stack are ignored. From Theorem 1 we know that arbitrarily modifying a proof cannot yield a correct refutation for a satisfiable formula. In particular, ignoring some clause deletions during checking can be thought of as removing those deletion instructions from the proof, and then checking the resulting proof. Hence, ignoring these deletion instructions of unit clauses is sound.

However, completeness is lost. More specifically: the class of proofs accepted by this method and the class of correct DRAT refutations differ [14]. In fact, these classes are incomparable. Example 4 shows a correct DRAT refutation but rejected once unit clause deletions are ignored; Example 5 presents the converse counterexample. These examples have been verified[1] with the current (as of April 20, 2018) versions of state-of-the-art DRAT checkers `DRAT-trim`[2] and `gratgen`[3].

Example 4 (a correct DRAT refutation rejected by DRAT checkers)**.** Consider the formula F and DRAT proof π from Example 1. There we showed that π is a correct DRAT refutation of F. However, state-of-the-art DRAT checkers reject it. This is an indirect consequence of ignoring the deletion instruction $\mathbf{d}\colon w\bar{x}$. Right before this instruction, the accumulated formula is $F \cup \{xy,\, x,\, w\bar{x}\}$. This formula implies exactly the literals x and w by unit propagation,

[1] https://github.com/arpj-rebola/pos2018/
[2] https://github.com/marijnheule/drat-trim/
[3] https://www21.in.tum.de/~lammich/grat/

and so the stack will contain only these literals. The clause $w\bar{x}$ is then a unit clause w.r.t. the stack, and so the deletion instruction $\mathbf{d}\colon w\bar{x}$ will be ignored. A state-of-the-art DRAT checker is actually checking the proof

$$\pi' = \mathbf{i}\colon xy,\ \mathbf{i}\colon x,\ \mathbf{i}\colon w\bar{x},\ \mathbf{i}\colon \bar{w}\bar{x},\ \mathbf{i}\colon wx,\ \mathbf{i}\colon yw,\ \mathbf{i}\colon \square$$

which is not a correct DRAT refutation of F. Let us walk through the steps that backwards checking performs over π', finally rejecting the proof.

- The empty clause $\square$ is checked for RUP in the formula $F \cup \{xy,\ x,\ w\bar{x},\ \bar{w}\bar{x},\ wx,\ yw\}$. This checks succeeds using clauses x, $\bar{w}\bar{x}$, and $w\bar{x}$ for propagation; these three clauses are scheduled for verification.
- The clauses yw and wx are not scheduled for verification, so they are skipped.
- The clause $\bar{w}\bar{x}$ is first checked for RUP in the formula $F \cup \{xy,\ x,\ w\bar{x}\}$, and this RUP check fails. Then the checker tries to find out if $\bar{w}\bar{x}$ is a RAT in the formula above upon literal $\bar{w}$. This check fails as well, since the clause $(\bar{w}\bar{x}) \vee (w\bar{x}) \setminus \{w\} = \bar{w}\bar{x}$ is not a RUP in the formula above.

Example 5 (an incorrect DRAT refutation accepted by DRAT checkers)**.** Now consider the formula F and DRAT proof σ from Example 1, which was an incorrect DRAT refutation of F. A state-of-the-art DRAT checker will accept it, again because of unit deletion instructions being ignored, namely $\mathbf{d}\colon x$. The checked proof is actually

$$\sigma' = \mathbf{i}\colon xy,\ \mathbf{i}\colon x,\ \mathbf{i}\colon w\bar{y},\ \mathbf{i}\colon \bar{w}\bar{y},\ \mathbf{i}\colon w,\ \mathbf{i}\colon \square$$

It is easy to check that each introduction in σ' is a RUP in the previously accumulated formula, so state-of-the-art checkers accept σ.

The caption "an incorrect DRAT refutation accepted by DRAT checkers" might sound dramatic, but let us reassure: we can do this *only* because the input CNF formula was unsatisfiable, as we know from Theorem 1. To the best of our knowledge, there is no way to make state-of-the-art proof checkers accept a DRAT proof of a satisfiable instance (barring overflow errors and other low level issues), and so this should not be interpreted as a soundness risk in DRAT checkers. In Section 6 we will however discuss how these issues may affect the pragmatics of SAT solving.

Thus, from a theoretical perspective, there is no problem on defining the class of DRAT proofs accepted by DRAT checkers as a proof system itself. Throughout the rest of this paper, we refer to the proof system defined by correct DRAT refutations as *specified DRAT*; and to the proof system defined by DRAT proofs accepted by DRAT checkers as *operational DRAT*.

5 Formalizing operational DRAT

Defining a proof system through the properties *of a transformation* of the proof is unconventional, albeit theoretically admissible. We now formalize the operational DRAT proof system as a set of inference rules and associated accumulated formulas, in the same fashion as the specified DRAT system. The criterion to decide whether or not a deletion instruction is accounted for depends on the literals occurring in the prepropagated stack, so we need to formalize this; once this is formalized, operational DRAT is straightforward to define. In a nutshell, the prepropagated stack contains a literal if and only if it can be obtained from the accumulated formula by unit propagation. However, there exists some non-determinism: many propagation choices can be chosen to decide this. We propose instead a semantic definition of the literals in the stack.

5.1 Shared UP-models

Consider a CNF formula F, and a partial assignment I. We say that I is a *unit propagation-model* (UP-model) of F whenever, for every clause $C \in F$, either of the following holds:

1. there is some literal $l \in C$ with $I(l) = 1$.
2. there are at least two distinct literals $l, k \in C$ such that $I(l) = I(k) =?$.

Intuitively, a unit propagation-model is stable under unit propagation: assuming the literals assigned by I, no further literals can be deduced from F by unit propagation. We say that a CNF formula F is *UP-satisfiable* whenever there exists an UP-model of F.

If F is UP-satisfiable, then we can define its *shared UP-model* $\mathcal{S}_F$ as the following mapping from literals to the set $\{0, ?, 1\}$:

$$\mathcal{S}_F(l) = \begin{cases} p & \text{if for all UP-models } I \text{ of } F \text{ we have } I(l) = p \\ ? & \text{otherwise} \end{cases}$$

In other words: whenever all UP-models I of F agree in the value of $I(l)$, then $\mathcal{S}_F$ maps l to that value; and otherwise it maps l to ?.

Theorem 2. *If F is UP-satisfiable, then $\mathcal{S}_F$ is an UP-model of F.*

Proof. Observe that since all UP-models of F are partial assignments, so is $\mathcal{S}_F$: whenever $\mathcal{S}_F(l) = 1$ for some literal l, then $\mathcal{S}_F(\bar{l}) = 0$ and vice versa, since in these cases all UP-models of F agree on the values of l and $\bar{l}$.

We now show that $\mathcal{S}_F$ is an UP-model of F by reduction to absurd. Assume it is not. Then, there is a clause C such that, firstly, $\mathcal{S}_F(l) \neq 1$ for every $l \in C$; and secondly, there is at most one literal $k \in C$ such that $\mathcal{S}_F(k) =?$. We discuss two cases:

- No such k exists. Then, for every literal $l \in C$, we find $\mathcal{S}_F(l) = 0$. Therefore, for every UP-model I of F, and every literal $l \in C$, we have $I(l) = 0$. This is a contradiction, since it implies that I is not an UP-model of F, and we know that some UP-model exists.
- Exactly one such k exists. Then, we know that for some UP-model I of F we have $I(k) \neq 1$. Since I is an UP-model of F, there must be two distinct literals k_1 and k_2 in C such that $I(k_1) = I(k_2) =?$. This implies $\mathcal{S}_F(k_1) = \mathcal{S}_F(k_2) =?$, whether all models agree on the value ? for k_1 and k_2 or whether they do not. But then we have $k_1 = k_2 = k$, which contradicts that k_1 and k_2 are distinct.

In both cases we reach a contradiction, so $\mathcal{S}_F$ is an UP-model of F. □

The following is the main result relating the shared UP-model to the prepropagated stack in a DRAT checker. The latter holds all literals that are implied by the current formula by unit propagation. Furthermore, once two complementary literals are found to be implied by unit propagation during stack prepropagation, DRAT checkers move on to backwards checking, regardless that the empty clause has been reached in the proof: at this stage, the empty clause can be derived as a RUP, since the formula itself leads to contradiction by unit propagation.

Theorem 3. *Let F be a CNF formula. The following hold:*

1. *If F is UP-satisfiable, then a literal l is implied by unit propagation over F if and only if $\mathcal{S}_F(l) = 1$.*

2. *If F is UP-unsatisfiable, then either F contains the empty clause, or two complementary literals are implied by unit propagation over F.*

Proof. To show this theorem, we will first prove two lemmas. We first show that, if a literal l is implied by unit propagation from F and I is an UP-model of F, then $I(l) = 1$. In this case, there must be literals $l_1, \ldots, l_n$ and clauses $C_1, \ldots, C_n$ in F such that $l_n = l$, and furthermore for every $1 \leq i \leq n$, we have $l_i \in C_i$ and $C_i \setminus \{l_i\} \subseteq \{\overline{l_1}, \ldots, \overline{l_{i-1}}\}$. We show that $I(l_i) = 1$ for all $1 \leq i \leq n$ by induction on i. The claim then follows straightforwardly because $l_n = l$.

If $i = 1$, then the clause C_1 only contains the literal l_1. Since C_1 is in F and I is an UP-model of F, we conclude $I(l_1) = 1$. Now assume that $I(l_1) = \cdots = I(l_i) = 1$ for some $i < n$; we show that $I(l_{i+1}) = 1$. From $l_{i+1} \in C_{i+1}$ and $C_{i+1} \setminus \{l_{i+1}\} \subseteq \{\overline{l_1}, \ldots, \overline{l_i}\}$ we conclude that the only literal k in C_{i+1} with $I(k) \neq 0$ is l_{i+1}. But since I is an UP-model of F, this implies that $I(l_{i+1}) = 1$. This concludes the proof by induction.

We now show the second lemma. Consider the set L of literals implied by unit propagation over F. If no two complementary literals occur in L, one can construct the a partial assignment J given by:

$$J(l) = 1 \text{ if } l \in L \qquad J(l) = 0 \text{ if } \bar{l} \in L \qquad J(l) =? \text{ if } l, \bar{l} \notin L$$

We show that, if no two complementary literals are implied by unit propagation over F, and F does not contain the empty clause, then J is an UP-model of F.

Let C be an arbitrary clause in F. Every literal $l \in C$ with $J(l) = 0$ satisfies that $\bar{l}$ is implied by unit propagation over F. Hence, we can find a finite sequence of literals $l_1, \ldots, l_n$ and clauses $D_1, \ldots, D_n$ in F with the following properties: firstly, for every $1 \leq i \leq n$, we have $l_i \in D_i$ and $D_i \setminus \{l_i\} \subseteq \{\overline{l_1}, \ldots, \overline{l_{i-1}}\}$; and secondly, for every literal $l \in C$ with $J(l) = 0$, the literal l is one of the literals $\overline{l_1}, \ldots, \overline{l_n}$.

We now reason by reduction to absurd. Assume J does not UP-satisfy C, i.e. there is at most one literal $l \in C$ that satisfies $J(l) =?$, and for all literals $l \in C$ we have $J(l) \neq 1$. In particular, this implies that all literals in C except at most one are mapped to 0 by J. Furthermore, C is nonempty, because $\square \notin F$. Hence, we can choose l to be the only literal in C with $J(l) \neq 0$, if such a literal exists; and otherwise an arbitrary literal in C. Since every other literal in C is one of $\overline{l_1}, \ldots, \overline{l_n}$, we have $C \setminus \{l_{n+1}\} \subseteq \{\overline{l_1}, \ldots, \overline{l_n}\}$. This shows that l_{n+1} is implied by unit propagation over F. In particular, $J(l_{n+1}) = 1$, which contradicts that no literals in C are mapped to 1 by J. By reduction to absurd, we conclude that J is an UP-model F.

We can now show the theorem. To show the "only if" implication of Claim 1, observe that if l is implied by unit propagation over F, then $I(l) = 1$ holds for every UP-model I of F by the first lemma above. Now, F is UP-satisfiable, so its shared UP-model $\mathcal{S}_F$ is an UP-model of F, hence $\mathcal{S}_F(l) = 1$. We now show the "if" implication. Firstly observe that $\square \notin F$: were this the case, F would be UP-unsatisfiable. Furthermore, no two complementary literals k, $\bar{k}$ are implied by unit propagation over F, for the "only if" implication we have just shown would then yield $1 = \mathcal{S}_F(k) = 0$. Hence, the second lemma shows that the partial assignment J defined above is an UP-model of F, and so it agrees with $\mathcal{S}_F$ in the value of l. We conclude that l is implied by unit propagation over F.

Finally, Claim 2 follows straightforward from the second lemma: if F is UP-unsatisfiable, F does not contain the empty clause and no two complementary literals are implied by unit propagation over F, the lemma shows that the partial assignment J is an UP-model of F, contradicting UP-unsatisfiability. □

5.2 The operational DRAT proof system

Equipped with these theoretical results, we are now ready to give a formal definition of operational DRAT as a proof system. We maintain the same conditions for a correct refutation of a formula F; we instead modify the notion of accumulated formula. We recursively define the *adjusted accumulated formula* $F\langle\pi\rangle$ of a formula F through a DRAT proof π as follows:

$$
\begin{aligned}
F\langle\epsilon\rangle &= F \\
F\langle \mathbf{i}\colon C,\ \pi\rangle &= F\langle\pi\rangle \cup \{C\} \\
F\langle \mathbf{d}\colon C,\ \pi\rangle &= \begin{cases} F\langle\pi\rangle & \text{if } F\langle\pi\rangle \text{ is UP-satisfiable, and } C \text{ is a unit clause w.r.t. } \mathcal{S}_{F\langle\pi\rangle} \\ F\langle\pi\rangle \setminus \{C\} & \text{otherwise} \end{cases}
\end{aligned}
$$

where unions and complementations have been taken as multiset operations. We then say that a DRAT proof $i_1, \ldots, i_n$ is an *operationally correct DRAT refutation* of F if $\square = i_m$ for some $1 \leq m \leq n$, and for every $1 \leq j \leq n$ either of the following holds:

- i_j is a deletion instruction $\mathbf{d}\colon C$.
- i_j is an introduction instruction $\mathbf{i}\colon C$, and C is either a RUP in $F\langle i_1, \ldots, i_{j-1}\rangle$, or a RAT in the same formula upon the first literal l in C.

6 How do you like your DRAT proofs?

Now that we have formal definitions for both flavors of DRAT, and that we have clarified their theoretical differences, we would like to open a discussion in the SAT community about the convenience to use one or another flavor. As explained above, both of them are sound proof systems, insofar as there is no correct DRAT refutation of a satisfiable formula for either flavor.

Hence, the theory does not yield a clear preference; we must consider the pragmatics of current and future SAT solving to decide on this. This section discusses a few issues that we consider interesting to take into account.

Checking specified DRAT proofs The reason why DRAT checkers do not check the specified flavor, deciding instead for the operational DRAT flavor, lies on unit propagation. Efficient unit propagation techniques adapted from SAT solving do not work when arbitrary literals are eliminated from the stack. This has to be done when a clause is removed through a deletion instruction, since otherwise the stack may contain literals that are not implied by unit propagation. One problem is to find out which literals need to be removed from the stack. Furthemore, combining backwards checking with stack prepropagation efficiently requires, with the current techniques, that the order on which literals occur in the stack is compatible with the order on which clauses are added throughout the proof. Otherwise, invariants on the two-watched literal may be violated. Given that the involved data structures originate in SAT solving, these issues may simply be an artifact of their procedence. We believe that efficient specified DRAT proof checking is possible by conveniently adapting the two-watched literal schema.

Practical extent of discrepancies The proofs presented in Examples 1, 4 and 5 are handmade, and very unlikely to be produced by a SAT solver. Unfortunately, the nature of unit propagation makes it very hard to tell whether discrepancies between the two flavors of DRAT occur in proofs produced by SAT solvers. However, we argue that there is a significant potential for these discrepancies to occur. The `DRAT-trim`[2] checker issues warnings whenever a unit

clause is deleted in its verbose mode; each such deletion can lead the prepropagated stack not to precisely reflect the literals that follow by unit propagation over the accumulated formula.

We ran `DRAT-trim` over the proofs generated by all participant solvers in the SAT Competition 2017 for all unsatisfiable benchmarks[1]. We set a time limit for each SAT solver and `DRAT-trim` of 5000s. Out of the 3144 instances, in 1857 instances both tools terminated and the generated DRAT proof was accepted. On these instances, we counted the number of reported ignored unit clause deletions. In 95% of them, `DRAT-trim` ignored some unit clause deletion, and in 44% of them more than a thousand unit clause deletions were ignored, with some cases reaching the order of 10^6 ignored deletion instructions. We consider that these results make it plausible that proofs with discrepant correctness status for each DRAT flavor occur in practice.

Total instances		3144	(100.0%)
SAT solver did not terminate		1059	(33.7%)
SAT solver reported unsatisfiability		2085	(66.3%)
from which	`DRAT-trim` did not terminate	220	(10.6%)
	`DRAT-trim` rejected the proof	8	(0.4%)
	`DRAT-trim` accepted the proof	1857	(89.1%)
	from which `DRAT-trim` ignored deletions	1774	(95.5%)
	`DRAT-trim` ignored more than 100 deletions	1480	(79.7%)
	`DRAT-trim` ignored more than 1000 deletions	814	(43.8%)
	Maximum number of ignored deletions	373710	

Figure 2: Experimental results on the occurrence of ignored deletions by `DRAT-trim`.

Proof generation with minimal overhead One of the goals of DRAT proof generation is to keep the resources dedicated to proof generation at solving time to a minimum. Discrepancies due to unit clause deletion sometimes require additional data structures and computing time. Certain techniques such as shrinking proofs during solving through extended resolution [2] or (inprocessing) bounded variable addition [20] regularly need to introduce and discard new variables and thus benefit from also internally deleting variables completely, before they are reused. This can be done when all occurrences of the reused literal are eliminated from the formula. However, if the literal has been propagated, some clauses would not be deleted when checking the associated DRAT proof. To generate a proof which is accepted by available checkers, one must store a dictionary of literal correspondences with new variables, and generate proof fragments with new variable names when literals are reused.

Another instance of unnecessary overhead provoked by unit deletion is the aforementioned deletion of intermediate clauses in proof generation for inprocessing techniques. In order for the accumulated formula in the proof to correspond to the clause database in the solver, it is necessary to know at solving time which deletion instructions will be ignored and which will be applied. Now, intermediate clauses are generated by hard-coded methods; in order to identify unit clause deletions, unit propagation must be performed over the intermediate clauses, which involves unnecessary reasoning effort. To the best of our knowledge, the current approach is to simply generate the proof as though the clause would be deleted, taking the risk that the accumulated formula and the clause database do not coincide. This becomes problematic once non-monotonicity of RAT introduction is taken into consideration.

Last, the DPR proof system [17], which supersedes DRAT, has a great potential to generate proof fragments for inprocessing techniques without using new variables [17]. This increases the

risk of propagating a literal due to an intermediate clause that will be subsequently deleted in the proof fragment, since the isolation of effects provided by new variables is lost. The recently proposed translation of PR into DRAT proofs [16] uses exactly one variable which is introduced and deleted for every PR proof steps and thus leads to the same risk.

On the other hand, since there are no checkers for specified DRAT, we are not aware of any efforts to generate proofs that comply with this standard. As a consequence, the extent of potential problems arising from switching to specified DRAT proofs is unknown. We also do not know how frequently, if at all, solvers produce proofs intended to be checked as specified DRAT proofs but rejected by DRAT checkers.

DRAT proofs for debugging If developers want to use DRAT checkers to debug their solvers, several problems arise. The most obvious is that since there was so far no theoretical description of the two flavors of DRAT, the developer may code a proof generation procedure that sticks to one flavor, but the proof is then checked by the other flavor, potentially leading to unexpected results and use of unnecessary resources for debugging. We think this is a compelling reason for the SAT community to decide for one of the two flavors.

A more subtle problem is proof rejection. When a DRAT checkers accept a proof, a correctness witness is generated and can be checked with certified tools [5, 19]. However, when the proof is rejected, there is currently no way to confirm this result. In particular, a developer would have a hard time figuring out whether an isolated incorrect proof is due to flavor discrepancy, to a bug in the SAT solver, or to a bug in the DRAT checker. Given that DRAT checkers are not that simple, and bugs have been found in the past, we consider that an *incorrectness* witness that can be validated by certified means would be desirable.

7 Conclusion

The original definition of the DRAT proof system in [28], which we call specified DRAT, contains significant differences with respect to the class of refutations accepted by DRAT checkers, referred to as operational DRAT. One aspect on which they differ is that operational DRAT treats formulas as multisets of clauses, whereas specified DRAT deals with sets of clauses. However, we consider this a mere matter of simplification in presentation, since proof generation becomes much simpler with multiset semantics, and this translates into shorter solving runtimes.

The second aspect on which they differ is the treatment of deletion of unit clauses. DRAT checkers use efficient techniques that depend on convoluted and fragile invariants. Limitations in the adaption of the two-watched literal schema have led DRAT checkers to ignore unit clause deletion instructions. The specified and operational flavors, where unit deletion instructions are performed and ignored respectively, result in incomparable classes of accepted refutations.

Despite non-monotonicity of RAT introduction, the operational DRAT proof system is sound: we have argued that ignoring unit clause deletion instructions does not lead to accept proofs for satisfiable formulas as refutations — in fact, ignoring arbitrary instructions has this property. We have then formalized the operational DRAT proof system in a more traditional way, with inferences and criteria for them. We do this by introducing the notion of shared unit propagation-models, which formalize the set of all literals that follow by unit propagation.

Currently there exists no efficient checker for the specified DRAT, although we identify some issues one such checker would need to tackle. Without a specified DRAT checker, we can only speculate about whether discrepant results ever occur in practice. Experimental results suggest that unit clause deletions, which is a necessary condition for discrepancies, occur overwhelmingly often. In any case, we consider this a reasonable motivation to work on efficient specified DRAT

checkers. Finally, we discuss the relevance of this difference for the pragmatics of SAT solving. We consider implications for the proof generation overhead at solving time, as well as the use of DRAT proofs as a tool for developers; in particular, we believe it is important that methods to certify incorrectness results reported by DRAT checkers are developed.

Acknowledgments This work was supported by the Austrian National Research Network S11403-N23 (RiSE), 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] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[2] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning SAT solvers. In Maria Fox and David Poole, editors, *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*. AAAI Press, 2010.

[3] Tomás Balyo, Marijn J. H. Heule, and Matti Järvisalo. SAT competition 2016: Recent developments. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 5061–5063. AAAI Press, 2017.

[4] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.

[5] 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.

[6] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[7] Walter Forkel, Tobias Philipp, Adrian Rebola-Pardo, and Elias Werner. Fuzzing and verifying RAT refutations with deletion information. In Vasile Rus and Zdravko Markov, editors, *Proceedings of the Thirtieth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2017, Marco Island, Florida, USA, May 22-24, 2017.*, pages 190–193. AAAI Press, 2017.

[8] 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.

[9] HyoJung Han and Fabio Somenzi. On-the-fly clause improvement. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 209–222. Springer, 2009.

[10] Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53:127–168, 2015.

[11] 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.

[12] 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.

[13] 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.

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

[15] Marijn J. H. Heule and Armin Biere. Compositional propositional proofs. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 444–459. Springer, 2015.

[16] Marijn J. H. Heule and Armin Biere. What a difference a variable makes. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2018.

[17] 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.

[18] 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.

[19] 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.

[20] Norbert Manthey, Marijn Heule, and Armin Biere. Automated reencoding of boolean formulas. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012.

[21] Norbert Manthey and Tobias Philipp. Formula simplifications as DRAT derivations. In Carsten Lutz and Michael Thielscher, editors, *KI 2014: Advances in Artificial Intelligence - 37th Annual German Conference on AI, Stuttgart, Germany, September 22-26, 2014. Proceedings*, volume 8736 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2014.

[22] 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.

[23] 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.

[24] 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.

[25] Adrián Rebola-Pardo. Unsatisfiability proofs in SAT solving with parity reasoning. Master's thesis, TU Dresden, 2015.

[26] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science - Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.

[27] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[28] 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.

[29] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10880–10885. IEEE Computer Society, 2003.