Complete and Efficient DRAT Proof Checking

Adrián Rebola-Pardo
TU Wien
`arebolap@forsyte.com`

Luís Cruz-Filipe
University of Southern Denmark
`lcf@imada.sdu.dk`

***Abstract*—DRAT proofs have become the standard for verifying unsatisfiability proofs emitted by modern SAT solvers. However, recent work showed that the specification of the format differs from its implementation in existing tools due to optimizations necessary for efficiency. Although such differences do not compromise soundness of DRAT checkers, the sets of correct proofs according to the specification and to the implementation are incomparable. We discuss how it is possible to design DRAT checkers faithful to the specification by carefully modifying the standard optimization techniques. We implemented such modifications in a configurable DRAT checker. Our experimental results show negligible overhead due to these modifications, suggesting that efficient verification of the DRAT specification is possible. Furthermore, we show that the differences between specification and implementation of DRAT often arise in practice.**

I. Introduction

Recent years have seen SAT solvers become increasingly popular, with many sucess stories in their application to several open problems, e.g. the recent computation of the Schur number five [11]. Popularity has also brought about the question of reliability: how much can we trust an answer provided by a SAT solver? A satisfiability result can be easily checked, since SAT solvers output a satisfying assignment. In the case of unsatisfiability results, several formats have been developed aimed at representing proofs of unsatisfiability in a way that is both compact and efficient to check. In this paper we focus on the DRAT format [10], [16], which has been widely adopted in SAT competitions and can represent most inferences done by SAT solvers. DRAT proofs can be checked both by efficient, untrusted programs such as `DRAT-trim`, and by certified, slower programs that work on extended formats such as LRAT [2] and GRAT [12].

A mismatch between the definition of DRAT proofs and the results of state-of-the-art proof checkers has been recently exposed [15]. The class of correct DRAT proofs and that of proofs accepted by modern checkers are incomparable: simple proofs which are correct but rejected, or incorrect but accepted, exist. This is not as catastrophic as it may sound, since it can be shown that whenever checkers accept a DRAT refutation of a formula, the latter is indeed unsatisfiable. Hence, one may consider state-of-the-art checkers as *implicitly* defining a proof system of their own. These two notions of correct DRAT refutations have been refered to as *flavors*: the original definition of a DRAT proof corresponds to the *specified* flavor, whereas the one defined by the results of DRAT checkers is the *operational* flavor. The fundamental difference between them is that in the operational flavor specific clause deletion instructions, called *unit deletions*, are ignored.

While this issue attracted some interest within the SAT solving community, a discussion on the convenience of either flavor is hindered by the absence of specified-DRAT checkers. The reason for this unavailability lies deep down at the heart of how DRAT checkers work. Deleting unit clauses breaks invariants required by some lazy data structures for unit propagation, which are necessary for the huge efficiency of checkers. Without specified-DRAT checkers, it is virtually impossible to assess how often discrepancies between the two flavors occur in proofs produced by SAT solvers in practice.

In this paper, we explain how an efficient specified-DRAT checker can be implemented. By carefully repairing the involved data structures, the invariants necessary for effective unit propagation can be restored. Extensively applying these repairs would be extremely expensive; we identify restrictions that greatly curb the induced overhead. To measure the reparation overhead in specified-DRAT checking, we implemented our method in a configurable checker, which can be run to check proofs on either flavor. To the best of our knowledge, this is the first specified-DRAT checker available. Experimental data suggests that the overhead of checking specified-DRAT proofs over checking operational-DRAT proofs is negligible. Furthermore, we find that discrepancies between both flavors occur relatively often in practice, and are not just an artifact of carefully handcrafted proofs.

Related work: There is extensive literature on clausal proof generation and checking for SAT solvers [5], [6], [8], [10], [16]. Several methods to validate correctness results of DRAT checkers through certified means have been proposed [2], [7], [12], although none of them covers incorrectness results. The incompleteness of state-of-the-art DRAT checkers and its relation with unit clause deletion has been observed and acknowledged [4], [10], [15].

II. Preliminaries

Given a variable x, we denote its *complement* by $\overline{x}$. A *literal* is a variable or its complement. A *clause* is a disjunction of literals; we denote clauses by juxtaposition, i.e. $x \vee y \vee \overline{z}$ is denoted by $xy\overline{z}$. We assume that clauses do not contain complementary literals. The *unsatisfiable* or *empty* clause is denoted by $\square$. A *CNF formula* is a conjunction of clauses. We follow the usual definitions of *satisfiability* and *entailment*. We construe CNF formulas as clause sets and clauses as literal sets. 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 finite, complement-free set of literals I. 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 say that a CNF formula F *implies* a literal l *by unit propagation* whenever there is a finite sequence $l_1, \ldots, l_n$ of literals such that $l_n = l$, and we can find a clause $C_i \in F$ with $l_i \in C_i$ and $C_i \setminus \{l_i\} \subseteq \{\overline{l_1}, \ldots, \overline{l_{i-1}}\}$ for $1 \leq i \leq n$. Furthermore, we say that F *implies a conflict by unit propagation* 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 $F \cup \overline{C}$ implies a conflict by unit propagation. Moreover, 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 all clauses $D \in F$ with $\bar{l} \in D$. We assume that clauses contain at least two literals. In practice, the empty clause is never introduced in the data structures, but size-one clauses are. For simplicity, we assume that a new literal $\top$ is made true by all partial assignments. Then, we replace size-one clauses l by the size-two clause $l\,\overline{\top}$.

Modern SAT solvers are able to generate unsatisfiability certificates called *DRAT proofs*. A DRAT proof is a string of instructions $i_1, \ldots, i_n$; every instruction is either a *clause introduction* $\mathbf{i}\colon C$ or a *clause deletion* $\mathbf{d}\colon C$, for a clause C. Given a DRAT proof π and a CNF formula F, the *accumulated formula* $F[\pi]$ by F through π is recursively defined as follows:

$$\begin{aligned} F[\epsilon] &= F \\ F[\mathbf{i}\colon C,\ \pi] &= (F \cup \{C\})[\pi] \\ F[\mathbf{d}\colon C,\ \pi] &= (F \setminus \{C\})[\pi] \end{aligned}$$

The set of literals implied by unit propagation from the formula accumulated by F through π is called the *accumulated partial assignment*. In [15], the accumulated partial assignment was characterized as the minimal UP-model of $F[\pi]$.

Given a CNF formula F, a DRAT proof $i_1, \ldots, i_n$ is called a *correct* DRAT proof 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 a either a RUP or a RAT in $F[i_1, \ldots, i_{j-1}]$.

Example 1. Throughout this paper we use the following running example. We consider a CNF formula F containing the following clauses:

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

Furthermore, we consider the following two DRAT proofs:

$$\pi = \mathbf{i}\colon x_5,\ \mathbf{d}\colon \overline{x_1}x_2,\ \mathbf{i}\colon x_9,\ \mathbf{i}\colon \square \qquad \pi' = \mathbf{i}\colon x_5,\ \mathbf{i}\colon x_9,\ \mathbf{i}\colon \square$$

Both π and π' are correct DRAT proofs. Let us check that the instruction $\mathbf{i}\colon x_9$ in π is correct. The accumulated formula at that point is $F' = (F \setminus \{\overline{x_1}x_2\}) \cup \{x_5\}$. $F' \cup \{\overline{x_9}\}$ implies both x_9 and $\overline{x_9}$ by unit propagation, so x_9 is a RUP in F'.

The proofs π and π' do not contain any RAT introduction instruction. As an example, clause $\overline{x_5}$ is not a RUP in F, but it is a RAT in F. The formula $F \cup \{x_5\}$ implies by unit propagation exactly the literals $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$, so $\overline{x_5}$ is not a RUP in F. To show that it is a RAT in F upon $\overline{x_5}$, we check that $\overline{x_5}x_6 = \overline{x_5} \vee (x_5x_6 \setminus \{x_5\})$ and $\overline{x_5}x_8 = \overline{x_5} \vee (x_5x_8 \setminus \{x_5\})$ are RUPs in F. This holds, for $F \cup \{x_5\overline{x_8}\}$ (resp. $F \cup \{x_5\overline{x_6}\}$) implies by unit propagation x_8 and $\overline{x_8}$ (resp. x_6 and $\overline{x_6}$). ■

Our definition of a DRAT proof, reflecting the original from [9], [10], is central to this paper. *DRAT checkers* are programs that determine whether a DRAT proof is correct or not. DRAT checking is computationally challenging, due to the sheer size of proofs and the need for unit propagation to check introduction instructions. Several DRAT checkers are available. `DRAT-trim`[1] is the *de facto* standard checker, and is used in SAT Competitions to certify unsatisfiability results [1], [10]. Some data structure improvements have been shown to induce notable improvements over `DRAT-trim` [12].

However, recent work exposed critical differences between the way DRAT proofs are defined and the way DRAT proofs are checked [15]. DRAT checkers ignore deletion instructions removing clauses that are unit w.r.t. the accumulated assignment. Hence, whereas the notion of correctness stays the same, DRAT checkers compute the accumulated formula differently: $F[\mathbf{d}\colon C,\ \pi]$ is defined as $F[\pi]$ if C is a unit clause w.r.t. the acumulated assignment for F; and $(F \setminus \{C\})[\pi]$ as usual otherwise. Proofs that are correct but rejected by DRAT checkers exist, and vice versa. We refer to the original definition as the *specified* flavor of DRAT, whereas the *operational* flavor uses the modified definition for accumulated formula.

A. Data structures for DRAT checking

Modern DRAT checkers are relatively complex programs. Efficient unit propagation is required to check the correctness of RUP and RAT introductions. This is achieved through the same *two-watched literal* schema CDCL SAT solvers are based upon, where each clause is *watched* on two distinct literals, and the clauses watched on literal l are stored in the *watchlist* for l [13]. Also as in SAT solvers, a *trace* of the assigned literals is kept as a stack. The trace stores the accumulated assignment (i.e. the literals implied by unit propagation by the accumulated formula), together with information about the order on which they were assigned and the reason clause that triggered that propagation. Moreover, watchlists keep track of clauses that are candidate to trigger future unit propagations. Both data structures maintain invariants throughout the execution of the DRAT checker, which are required so that all available unit propagations are appropriately detected.

[1] https://github.com/marijnheule/drat-trim

At a given stage during checking, the j-th instruction is considered. The trace then contains the accumulated assignment I_j for the accumulated formula F_j. Remarkably, literals in the trace occur in the same order as they were assigned. In fact, they are *staged*: the trace behaves like a stack that grows monotonically throughout the proof, so it can be divided in sections such that the first j' sections correspond to the accumulated assignment $I_{j'}$. Furthermore, every clause is watched in such a way that that the following invariant holds:

Invariant 1. *If a clause is watched on literals l and k, and the current trace I_j falsifies l, then I_j satisfies k.*

A DRAT checker can decide whether a CNF formula together with some *assumed literals* implies a conflict by unit propagation using a well-known procedure [13]. After assigning each assumed literal l, the watchlist for $\bar{l}$ is traversed. By Invariant 1, clauses that trigger new propagations must be watched on $\bar{l}$, so they are all eventually encountered. The checker tries to relocate the watches in each clause so that Invariant 1 is satisfied. Two conditions may prevent this. In one case, the trace falsifies all literals, hence a conflict is reported. In the other case, all literals are falsified but for one unassigned literal k. In this case, k is implied by unit propagation, so it can be assigned to true. In turn, this triggers new propagations, which are detected when the watchlist for $\bar{k}$ is traversed. If no further watchlists for previously assigned literals remain to be processed, and a conflict has not been reached, the checker can conclude there is no conflict by unit propagation. Preparing the data structures to check if a new set of assumed literals implies a conflict by unit propagation only requires to unassign the literals in the trace: any watch choice satisfies Invariant 1 correct afterwards.

B. Double-sweep DRAT checking

The described procedure can already check DRAT proofs: to check if C is a RUP in F, it suffices to assume $\bar{C}$ and perform unit propagation, and RAT checking can be done via several RUP checks. There is however much room for improvement. DRAT checkers implement a number of techniques to speed checking up, e.g. resolution candidate caching [12] and core-first propagation [8]. Two techniques are especially relevant to our work: an undocumented technique we call *incremental prepropagation*, and *backwards checking* [8]. DRAT checkers perform two sweeps through the proof. In the first sweep, incremental prepropagation traverses the proof forwards, caching propagation information that will be used in the second sweep. Incremental prepropagation performs no proper checking. Instead, the second sweep called backwards checking performs RUP or RAT checks for introduction instructions, traversing the proof backwards. Backwards checking allows to skip irrelevant parts of the proof by performing conflict analysis.

Incremental prepropagation: The description of the unit propagation algorithm above implicitly assumes that the trace starts empty. This is unnecessary: as long as the watches satisfy Invariant 1, the initial trace may contain literals. Invariant 1 also implies that the trace contains all literals implied by unit propagation. DRAT checkers exploit this by preserving the anterior part of the trace stack between instructions during the first sweep, in such a way that the trace grows monotonically.

Incremental prepropagation traverses the CNF instance and the DRAT proof forwards. Every premise or introduction instruction adds a clause C to the clause database; deletion instructions are discussed later in this section. After a clause is introduced, the trace and watchlists are updated. New literals implied by unit propagation are *incrementally* added to the trace stack. Hence, the trace has the form $I_0 I_1 \dots I_m$, and the substack $I_0 \dots I_j$ is the accumulated assignment after the j-th instruction. The data structures can be updated in three ways:

- If watches for C respecting Invariant 1 exist, no further literals are propagated. C is added to the relevant watchlists, and the checker moves on to the next instruction.
- If C is falsified by the trace, then C is a RUP in F, and moreover $\Box$ is a RUP in $F \cup \{C\}$. This can be treated as the end of the proof, and backwards checking starts.
- Otherwise, C only contains falsified literals except for one unassigned literal l. In this case, C is watched in l and in some other literal, and l follows by unit propagation. Hence, l is pushed into the trace stack, and the propagation procedure is called to derive new literals.

As observed above, the stack structure of the trace is monotonic with respect to the proof: to recover the trace computed before introducing C, if C was the reason to propagate l, it suffices to drop the latter part of the stack starting with l. When doing so, watches need not be modified, although this is not so obvious; again, we defer this discussion to Section III-C, when we will have the tools to explain the reason for this.

Example 2. Let us reconsider the proofs from Example 1:

$$\pi = \mathbf{i}\colon \overline{\top} x_5,\ \mathbf{d}\colon \overline{x_1} x_2,\ \mathbf{i}\colon \overline{\top} x_9,\ \mathbf{i}\colon \Box \qquad \pi' = \mathbf{i}\colon \overline{\top} x_5,\ \mathbf{i}\colon \overline{\top} x_9,\ \mathbf{i}\colon \Box$$

where we have introduced the literal $\overline{\top}$ to prevent size-one clauses. Figure 1 shows the evolution of the trace throughout incremental prepropagation. Observe that the trace evolution for π is non-monotonic, since some literals are removed from the trace, whereas the one for π' is monotonic. The reason for this difference is the deletion of reason clause $\overline{x_1} x_2$ in π. State-of-the-art checkers would ignore this deletion instruction in π because $\overline{x_1} x_2$ is a unit clause w.r.t. the trace before the deletion, thus *implicitly* checking proof π'. Therefore, checking π and π' is equivalent under the operational flavor. Observe that the procedure described above to restore previous traces works well in all cases except for recovering the trace "after $\mathbf{i}\colon \overline{\top} x_5$" from "$\mathbf{d}\colon \overline{x_1} x_2$" in π. As we will see later, this is the reason why unit clause deletions are ignored. ■

Backwards checking: Once a conflict in the accumulated assignment is reached, the second sweep starts. Backwards checking traverses the proof from the conflict point towards the beginning of the proof. Introduction instructions are checked for RUP or RAT by restoring the trace to its state before that instruction during incremental inprocessing. RUP checks for a clause C are performed by assumming $\overline{C}$ and propagating; RAT checks can be reduced to a number of RUP checks.

trace preprocessing for $\pi = \mathbf{i}{:}\,\overline{\top} x_5,\ \mathbf{d}{:}\,\overline{x_1} x_2,\ \mathbf{i}{:}\,\overline{\top} x_9,\ \mathbf{i}{:}\,\Box$							
start		after $\mathbf{i}{:}\,\overline{\top} x_5$		after $\mathbf{d}{:}\,\overline{x_1} x_2$		after $\mathbf{i}{:}\,\overline{\top} x_9$	
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:	$\overline{x_1 x_2} x_3$	x_3:	$\overline{x_1 x_2} x_3$	x_6:	$\overline{x_1 x_5} x_6$	x_6:	$\overline{x_1 x_5} x_6$
x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_5 x_6} x_4$	x_4:	$\overline{x_5 x_6} x_4$
		x_5:	$\overline{\top} x_5$	x_3:	$\overline{x_6 x_4} x_3$	x_3:	$\overline{x_6 x_4} x_3$
		x_6:	$\overline{x_1 x_5} x_6$	x_8:	$\overline{x_3 x_6} x_8$	x_8:	$\overline{x_3 x_6} x_8$
		x_7:	$\overline{x_2 x_5} x_7$			x_9:	$\overline{\top}\overline{x_9}$
		x_8:	$\overline{x_3 x_6} x_8$			x_7:	$\overline{x_9} x_7$
						x_{10}:	$\overline{x_4 x_9} x_{10}$
						$\overline{x_{10}}$:	$\overline{x_7 x_8 x_9 x_{10}}$

trace preprocessing for $\pi' = \mathbf{i}{:}\,\overline{\top} x_5,\ \mathbf{i}{:}\,\overline{\top} x_9,\ \mathbf{i}{:}\,\Box$					
start		after $\mathbf{i}{:}\,\overline{\top} x_5$		after $\mathbf{i}{:}\,\overline{\top} x_9$	
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_2:	$\overline{x_1} x_2$
x_3:	$\overline{x_1 x_2} x_3$	x_3:	$\overline{x_1 x_2} x_3$	x_3:	$\overline{x_1 x_2} x_3$
x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_1 x_3} x_4$
		x_5:	$\overline{\top} x_5$	x_5:	$\overline{\top} x_5$
		x_6:	$\overline{x_1 x_5} x_6$	x_6:	$\overline{x_1 x_5} x_6$
		x_7:	$\overline{x_2 x_5} x_7$	x_7:	$\overline{x_2 x_5} x_7$
		x_8:	$\overline{x_3 x_6} x_8$	x_8:	$\overline{x_3 x_6} x_8$
				x_9:	$\overline{\top} x_9$
				x_{10}:	$\overline{x_4 x_9} x_{10}$
				$x_{\overline{10}}$:	$\overline{x_7 x_8 x_9 x_{10}}$

Fig. 1. Trace evolution throughout incremental prepropagation for proofs π and π' from Example 1. Reason clauses for each propagated literal are indicated.

Done naïvely, restoring the trace would mean storing the trace for each instruction in the proof, and then retrieving the appropriate trace for every instruction. Watches would then need to be relocated too, incurring in large costs. Fortunately, as explained above, the checker can restore a previous trace can be recovered by simply removing the latter part of the trace stack. Also, this makes watch relocation unnecessary.

This does not justify checking the proof backwards: the same effect can be obtained by checking introductions during the first sweep. However, by performing conflict analysis on each conflict similarly to CDCL [13], the checker can determine which clauses were involved in the conflict. These clauses get *marked*; unmarked clauses are skipped during backwards checking, since they are unnecessary to derive □.

Ignoring unit clause deletions: We had let aside the issue of deletion instructions in incremental prepropagation. Clauses that were not involved in trace propagation can be safely removed from the clause database and watchlists. Otherwise, C triggered the propagation of a literal l in the trace; we refer to C as a *reason clause* for l. Removing a reason clauses is cumbersome. For one, the propagated literal l may be used to propagate later literals in the trace. For another, l (or any of the subsequently propagated literals) may *still* be implied by unit propagation, just through a different propagation sequence.

The solution adopted by state-of-the-art checkers is rather pragmatic: ignore such deletions. If the checker only ignored reason clauses, the results would be unpredictable, for reason clauses depend on arbitrarities like the order of clauses in the formula or the order of literals within clauses. Instead, a more semantic criterion is used: a deletion instruction for C is ignored whenever C is a unit w.r.t. the accumulated assignment, which is stored in the trace. This is a necessary condition for being a reason clause, albeit not a sufficient one.

Example 3. Consider the instruction $\mathbf{d}{:}\,\overline{x_1} x_2$ in proof π in our running example. At this point, the trace is storing the accumulated assignment $\{x_1, x_2, x_3, x_4, x_5, x_7, x_6, x_8\}$, and the clause $\overline{x_1} x_2$ is a unit w.r.t. this assignment. Therefore this deletion instruction is simply ignored by DRAT checkers. ■

This criterion makes the results of DRAT checkers stable, i.e. equivalent representations of proofs yield the same correctness result. However, ignoring unit clause deletions changes the class of accepted proofs: DRAT checkers are checking *something else* instead. The implicitly defined proof system is sound, i.e. it can only prove unsatisfiable formulas. However, its class of correct proofs is incomparable to that of correct DRAT proofs. The implicit proof system has been formalized and named *operational-DRAT*, in contrast to the originally defined *specified-DRAT* proof system. A comparison between the two flavors and a discussion on the need for specified-DRAT checkers can be found in [15].

III. (Naïvely) checking specified-DRAT proofs

Due to the problems discussed in Section II-B, no DRAT checkers for the specified flavor are available: the invariants broken by unit clause deletion are precisely those that make DRAT checking efficient. In this section, we describe how to restore broken invariants after unit clause deletion. The operations described in this section are expensive, but the optimizations in Section IV vastly curb this overhead.

Our first goal is to construct the trace after a reason clause deletion during incremental propagation, such as the trace "after $\mathbf{d}{:}\,\overline{x_1} x_2$" in Example 2. A very inefficient way to do that would be simply to discard the trace and the watches and reconstruct them from scratch. We aim to improve over this by reusing the trace before the deletion as much as possible.

We construct the trace after deleting the reason clause C for literal l in two stages. First, we identify which literals in the trace used l to be derived by unit propagation; we call these literals the *propagation cone* of l. After removing the propagation cone from the trace, the second stage restores into the trace the removed literals that are still implied by unit propagation. These two stages are illustrated in Example 4.

A. Computing the propagation cone

Intuitively, the propagation cone $P(l)$ for literal l with respect to a trace is determined inductively by two rules:

- The literal l is in the propagation cone.
- A literal k from the trace with reason clause D is in the propagation cone if D contains a (necessarily falsified) literal $m \neq k$ where $\overline{m}$ is in the propagation cone.

after $\mathbf{i}$: $\overline{\top}x_5$		after cone removal		after reinsertion		after propagation	
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_5:	$\overline{\top}x_5$	x_5:	$\overline{\top}x_5$	x_5:	$\overline{\top}x_5$
x_3:	$\overline{x_1x_2}x_3$	x_6:	$\overline{x_1x_5}x_6$	x_6:	$\overline{x_1x_5}x_6$	x_6:	$\overline{x_1x_5}x_6$
x_4:	$\overline{x_1x_3}x_4$			x_4:	$\overline{x_5x_6}x_4$	x_4:	$\overline{x_5x_6}x_4$
x_5:	$\overline{\top}x_5$					x_3:	$\overline{x_6x_4}x_3$
x_6:	$\overline{x_1x_5}x_6$					x_8:	$\overline{x_3x_6}x_8$
x_7:	$\overline{x_2x_5}x_7$						
x_8:	$\overline{x_3x_6}x_8$						

Fig. 2. Constructing the trace "after $\mathbf{d}$: $\overline{x_1}x_2$" from π in Example 2.

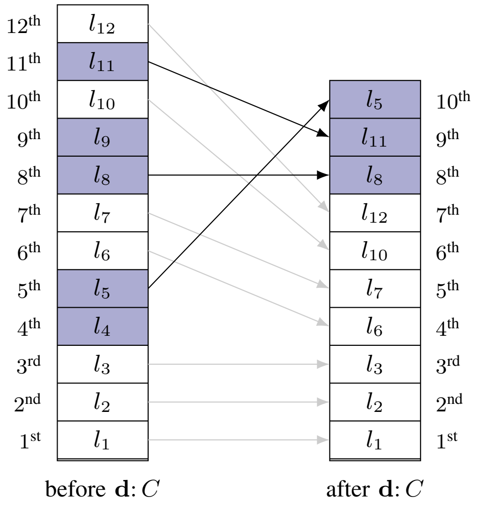

Fig. 3. Stack structure of trace reconstruction after a unit deletion. In this case, the deleted clause C is the reason clause for l_4. The propagation cone $P(l_4)$ is shaded on the left; as explained in Section III-A these literals are removed, and the unshaded part in the stack on the right is obtained. Some literals from $P(l_4)$ may be reinserted as presented in Section III-B; these are the shaded literals on the right, which need not preserve the order on the left.

To compute the propagation cone $P(l)$ w.r.t. a trace inducing the partial interpretation I, let $P_0(l) = \{l\}$, and

$$P_{n+1}(l) = P_n(l) \cup \{k \in I \mid \exists m \in R_k \setminus \{k\},\, \overline{m} \in P_n(l)\}$$

for each $n \geq 0$, where we denote by R_k the reason clause for literal k in the trace. The propagation cone is then the fixpoint $P(l) = \bigcup_{n \geq 0} P_n(l)$, which exists and is reachable because the sequence $(P_n(l))_{n \in \mathbb{N}}$ is increasing and $P(l)$ is finite. Because the reason clauses for trace literals are stored for conflict analysis purposes, all information needed for computing the propagation cone is available. The cone $P(l)$ is then removed from the trace, keeping the order of remaining literals.

B. Reintroducing literals implied by unit propagation

The fact that a literal k is in the propagation cone of l only means that l was used to derive k by unit propagation in the original trace; but k might still be implied through a different propagation sequence. Such literals must be restored into the trace; to find them, we exploit that unit propagation only requires Invariant 1 to discover all propagations. To satisfy it, we can relocate the watches; calling unit propagation would then do the heavy work. Again, the simple way is to relocate watches for each clause; again, we can outperform this.

Let I and J be the partial assignments defined by the traces before and after the removal of the propagation cone. Invariant 1 is satisfied by I, but possibly violated by J. This only happens for clauses D with watched literals k and m such that $J(k) = 0$ and $J(m) \neq 1$. Removing literals from I can only unassign literals; in particular, we infer that $I(k) = 0$. By Invariant 1 we conclude that $I(m) = 1$, and so m got unassigned by the removal of the propagation cone. Hence, m was in the propagation cone.

This means that the only clauses whose watches may need to be relocated are watched in a literal from the propagation cone. In order to enforce Invariant 1, one can traverse the watchlist for every literal m in the propagation cone $P(l)$ and relocate watches. When this cannot be done, then Invariant 1 is enforced by assigning literal m back into the trace. Furthermore, in the latter case, all subsequent clauses watched in m have correct watches, so we can move on to the next propagation cone literal. This procedure may reassign some literals, which may in turn lead to new propagations. Since Invariant 1 is satisfied afterwards, we can simply perform unit propagation to find them out. Our procedure always reintroduces these literals in the latter part of the stack; this will become very relevant in Section III-C. An overview of the procedure is depicted in Figure 3.

Example 4. Let us consider the traces for π from Example 2. Starting from the trace "after $\mathbf{i}$: $\overline{\top}x_5$", we construct the trace "after $\mathbf{d}$: $\overline{x_1}x_2$". Let us assume the following watch choices (shown as dots and only for clauses of size larger than 2):

$$\overline{x_1\dot{x}_2}\dot{x}_3 \quad \overline{x_2\dot{x}_5}\dot{x}_7 \quad \overline{x_5\dot{x}_6}\dot{x}_4 \quad \overline{\dot{x}_6x_4}\dot{x}_3 \quad \overline{x_4\dot{x}_9}\dot{x}_{10}$$
$$\overline{x_1\dot{x}_3}\dot{x}_4 \quad \overline{x_1\dot{x}_5}\dot{x}_6 \quad \overline{x_3\dot{x}_6}\dot{x}_8 \quad \overline{x_3}\dot{x}_9\dot{x}_{10} \quad \overline{x_7x_8\dot{x}_9\dot{x}_{10}}$$

Clause $\overline{x_1}x_2$ is the reason for literal x_2 in the trace "after $\mathbf{i}$: $\overline{\top}x_5$". The propagation cone $P(x_2)$ contains the literals x_2, x_3, x_4, x_7, x_8. By removing those literals from the trace, we obtain the trace "after cone removal" in Figure 2. The procedure above can be applied to the watchlists for literals in $P(x_2)$. We perform the following changes:

- Watchlist for literal x_3: clause $\overline{\dot{x}_6x_4}\dot{x}_3$ becomes $\overline{x_6\dot{x}_4}\dot{x}_3$.
- Watchlist for literal x_4: clause $\overline{x_5\dot{x}_6}\dot{x}_4$ causes literal x_4 to be reinserted in the trace.
- Watchlist for literal x_7: clause $\overline{x_2\dot{x}_5}\dot{x}_7$ becomes $\overline{\dot{x}_2x_5}\dot{x}_7$.
- Watchlist for literal x_8: clause $\overline{x_3\dot{x}_6}\dot{x}_8$ becomes $\overline{\dot{x}_3x_6}\dot{x}_8$.

This yields the trace "after reinsertion". Unit propagation then finds clause $\overline{x_6\dot{x}_4}\dot{x}_3$ in the watchlist for $\overline{x_4}$, propagating x_3, and clause $\overline{\dot{x}_3x_6}\dot{x}_8$ in the watchlist of $\overline{x_3}$, propagating x_8. We obtain the trace "after propagation", which corresponds to the trace "after $\mathbf{d}$: $\overline{x_1}x_2$". ■

C. Restoring trace and watches in backwards checking

The methods explained above apply to the incremental prepropagation sweep. It nevertheless remains unclear how would this work during backwards checking. One problem is recovering the trace before the deletion: removing the latter part of the trace as in Section II-B does not work anymore: fter reverting a clause deletion, some unassigned literals may become assigned. In terms of Example 2, what we need to do is to recover the trace "after $\mathbf{i}\colon \overline{\top} x_5$" from "$\mathbf{d}\colon \overline{x_1} x_2$" for π.

For the time being, our solution is simple: store the trace every time a unit clause deletion is processed during incremental propagation, and then restore it back when the deletion is reverted during backwards checking. This does not solve all the problems, though. In Section II-B, the trace is restored by removing its latter part. As we mentioned there, Invariant 1 is satisfied after doing so; let us inspect the reasons for this.

Removing *arbitrary* literals from the trace can violate Invariant 1, which is required for exhaustive unit propagation. For example, a clause $x_1 x_2$ satisfies the Invariant 1 for a trace containing x_1 and $\overline{x_2}$, but violates it after x_1 is dropped from the trace. Operational-DRAT checkers must be somehow preventing this situation. It is apparent from Invariant 1 and from the monotonic growth of the trace stack in operational-DRAT checking that, once a watched literal is satisfied by the trace during stack prepropagation, further watch relocation is unnecessary. This is not a only an efficiency hack, but also needed to maintain Invariant 1 during backwards checking too: this ensures that, in the conditions above, if x_1 (resp. $\overline{x_2}$) was added to the trace in the j_1-th (resp. j_2-th) instruction during trace preprocessing, then $j_2 \geq j_1$. Hence, during backwards checking, $\overline{x_2}$ is dropped from the trace before or at the same time as x_1, and so the problematic situation above never arises.

Invariant 2. *Consider a clause F in the current accumulated formula for the c-th instruction F_c that is watched on a literal l satisfied by the current trace I_c. Let $p < c$ the largest index such that I_p does not satisfy l, and k be the other watched literal in D. Then either of the following holds:*

a) *$D \notin F_r$ for some index $p \leq r < c$*
b) *$I_r(k) \neq 0$ for some index $p \leq r \leq c$*

This invariant is preserved by operational-DRAT checkers, and forces Invariant 1 to hold after the removal of the latter part of the trace stack when reverting a clause introduction during backwards checking. Unfortunately, reverting a unit clause deletion by restoring the stored trace violates Invariant 2, and this eventually causes Invariant 1 to be violated.

Example 5. Consider now the clause $\overline{x_2}\dot{\overline{x_5}}\dot{x_7}$ during backwards checking in proof π from Example 1. After instruction $\mathbf{d}\colon \overline{x_1} x_2$, literals $\overline{x_2}$ and x_7 are unassigned, so Invariant 1 holds. However, Invariant 2 is violated with this watch choice: the literal x_7 is last not satisfied in the "start" trace, but this trace falsifies $\overline{x_2}$. Invariant 1 is eventually violated too. In "after $\mathbf{i}\colon \overline{\top} x_5$", literal $\overline{x_2}$ becomes falsified and x_7 becomes satisfied, and so Invariant 1 is still satisfied. Once backwards checking moves on to "start", x_7 is unassigned while $\overline{x_2}$ is still falsified, and this violates Invariant 1. RUP checks may then report false negatives: if literal x_5 is added to the trace, then literal x_7 must be propagated, but since the clause is not watched on literal $\overline{x_5}$ the checker will not inspect this clause. ■

The reason why Invariant 2 is broken in Example 5 lies on the non-monotonic changes that reverting the reason clause deletion $\mathbf{d}\colon \overline{x_1} x_2$ causes in the trace. Restoring Invariant 2 is difficult, since this requires storing the traces after instructions. Instead, we establish an invariant that is strong enough to force Invariant 1 and weak enough to be simple to maintain.

Invariant 3. *Consider a clause D in the current accumulated formula F_c for the c-th instruction that is watched on a literal l satisfied by the current trace I_c. Let $p < c$ the largest index such that I_p does not satisfy l, and k be the other watched literal in D. Then either of the following holds:*

a) *$D \notin F_r$ for some index $p \leq r < c$*
b) *$I_r(k) \neq 0$ for some index $p \leq r \leq c$*
c) *$\overline{k}$ is in the propagation cone from Section III-A at a deletion in some index $p < r \leq c$.*

Together, Invariants 1 and 3 are preserved when reverting an introduction instruction $\mathbf{i}\colon C$ during backwards checking at index c. Assume that they both hold at the c-th instruction. If Invariant 1 was violated at index $c-1$ by some clause $D \in F_{c-1}$, then the value of p would necessarily be $c-1$, and $I_c(k) = I_{c-1}(k) = 0$. Since $\mathbf{i}\colon C$ is an introduction instruction, Invariant 3 would be violated at index $c-1$, which is a contradiction. On the other hand, if Invariant 3 was violated at index $c-1$, then we have $I_{c-1}(l) = I_c(l) = 1$, and furthermore $D \in F_r$ for all $p \leq r < c-1$; $I_r(k) = 0$ for all $p \leq r \leq c$; and $\overline{k}$ is never removed as a part of a propagation cone at an index $p < r \leq c-1$. Because $\mathbf{i}\colon C$ is an introduction instruction $I_c(k) = I_{c-1}(k) = 0$ holds, and $\overline{k}$ is also not removed as a part of a propagation cone at index c. But then Invariant 3 would be violated at index c, which is again a contradiction.

The previous paragraph shows that Invariant 3 is strong enough to guarantee the same good behavior as Invariant 2. However, in the specified-DRAT case we also need to consider reverting deletion instructions $\mathbf{d}\colon C$ during backwards checking at index c, and in general Invariant 3 is not preserved by this operation (although it *almost* is, as we will see in Section IV-C). Instead, we explicitly reestablish the invariant by relocating the watches in every clause D in the accumulated formula F_{c-1} before the deletion. If D is not a unit clause w.r.t. I_{c-1}, we choose as watches any two non-falsified literals. Otherwise, it contains one satisfied literal l, which is chosen as one of the watches. All other literals $k \in D \setminus \{l\}$ are falsified by I_{c-1}. We choose as the second watch the k such that $\overline{k}$ occurs the latest in the trace stack I_{c-1}. Finding k is computationally simple, since the trace is stored as an array in memory, and so it boils down to pointer comparison.

This watch choice trivially satisfies Invariant 1; we show that Invariant 3 is attained too. The former case is straightforward; we explain the case when D is a unit w.r.t. I_{c-1}. Assume D violates Invariant 3. Then we have $I_{c-1}(l) = 1$, and furthermore $D \in F_r$ for all $p \leq r < c-1$; $I_r(k) = 0$ for all $p \leq r \leq c$; and $\overline{k}$ is never removed as a part of a propagation cone at an index $p < r \leq c-1$; where p is defined as in Invariant 3. The trace I_p at the p-th instruction is saturated under unit propagation, so $I_p(l) \neq 1$ implies that there is some $m \in D \setminus \{l\}$ such that $I_p(m) \neq 0$. Our choice of watch k implies that $\overline{m}$ occurs strictly earlier in I_{c-1} than $\overline{k}$. Now consider the instruction at the $(c-1)$-th index.

- If it is an introduction, then I_{c-1} is obtained from I_{c-2} by appending literals in the later part of the stack. Because $I_{c-2}(k) = 0$, $\overline{k}$ is not one of the appended literals; and $\overline{m}$ occurs strictly earlier than $\overline{k}$ in I_{c-1}, so neither is $\overline{m}$. We conclude that $\overline{m}$ occurs strictly earlier than $\overline{k}$ in I_{c-2}.
- If it is a deletion, I_{c-1} is obtained from I_{c-2} by removing a propagation cone P, and reinserting some literals from P into the result. We know that $\overline{k} \notin P$; in particular $\overline{k}$ is not reintroduced. As observed at the end of Section III-B, literals are reintroduced at the later part of the stack; so if $\overline{m} \in P$ held true, $\overline{m}$ would occur later than $\overline{k}$ in I_{c-2}, but we have the opposite case. Thus, $\overline{m} \notin P$, and so $\overline{m}$ occurs strictly earlier than $\overline{k}$ also in I_{c-2}.

Iterating this argument shows that $\overline{m}$ occurs strictly earlier than $\overline{k}$ in I_{p+1}. Now, $I_p(l) \neq 1 = I_{p+1}(l)$, so the instruction at index p must be an introduction. Then, I_p is obtained from I_{p+1} by removing literals in the later part of the stack. Now, $I_p(k) = I_{p+1}(k) = 0$, so $\overline{k}$ is not removed; and $\overline{m}$ occurs earlier than $\overline{k}$, so neither is $\overline{m}$. But then $I_p(m) = I_{p+1}(m) = 0$ contradicts our choice of m. Therefore, Invariant 3 is fulfilled.

This completes our method for checking specified-DRAT proofs with incremental preprocessing and backwards checking. To summarize, we give a method that behaves essentially like operational-DRAT checkers, the only difference being the treatment of unit clause deletion instructions. During incremental preprocessing, our method is able to construct a trace reflecting the accumulated assignment after the deletion, and relocate watches in a suitable way. By storing this assignment to memory, we are able to restore it when the same unit clause deletion is encountered during backwards checking; at that point, watches for all clauses must be relocated.

IV. Optimizing unit clause deletion

The methods from Section III are computationally expensive, and in practice they make specified-DRAT checking much less efficient than operational-DRAT checking. This overhead is mainly due to three causes. First, the fixpoint computation for the propagation cone involves traversing the trace quadratically many times. Second, storing each trace before a deletion instruction may have a notable impact in memory even if the changes in the trace are minimal. Last, the watch relocation method in Section III-C involves relocating the watches for every clause in the formula. We now explain optimizations that greatly reduce the clause deletion-induced overhead in specified-DRAT checking.

A. Linearly computing propagation cones

In order to efficiently compute propagation cones, yet another invariant maintained by traces can be exploited:

Invariant 4. *Let l be a literal in the trace with reason clause R_l. Then, every literal $k \in R_l \setminus \{l\}$ is falsified by the trace, and $\overline{k}$ either is $\top$, or occurs earlier than l in the trace stack.*

```
$P(l) \coloneqq \{l\}$
for $k$, trace literal after $l$ do
    if there is a literal $m \in R_k$ with $\overline{m} \in P(l)$ then
        $P(l) \coloneqq P(l) \cup \{k\}$
    end if
end for
```

Fig. 4. Algorithm to linearly compute the implication cone

literal	position index	reason
x_2	3rd	$\overline{x_1}x_2$
x_3	4th	$\overline{x_1}\,\overline{x_2}x_3$
x_4	5th	$\overline{x_1}\,\overline{x_3}x_4$
x_7	8th	$\overline{x_2}\,\overline{x_5}x_7$
x_8	9th	$\overline{x_3}\,\overline{x_6}x_8$

Fig. 5. Information stored to reconstruct trace “after **i**: $\overline{\top}x_5$” from trace “after **d**: $\overline{x_1}x_2$” in Example 1.

The algorithm in Figure 4 exploits Invariant 4 to compute the implication cone in a single pass through the trace[2].

B. Storing deleted traces as permutations

Rather than storing each trace before a reason clause deletion during incremental prepropagation and restoring it during backwards checking, we can store the permutation that the trace undergoes. By deleting a clause, no literal is derived: some literals are removed from the trace, and some others are moved to the latter part of the trace stack. From Figure 3 it is apparent that storing the original reasons and positions within the trace for propagation cone literals is enough to restore the trace before deletion from the trace after deletion. Following Example 1, we store the information in Figure 5 to reconstruct the trace “after **i**: $\overline{\top}x_5$” from the trace “after **d**: $\overline{x_1}x_2$”.

C. On-demand watch relocation

Our previous analysis required the relocation of watches during backwards checking for all clauses in the accumulated formula. This is immensely wasteful: our preliminary experiments showed that doing so takes up to 85% of the checking runtime. This can however be vastly improved, reducing the runtime share spent on this sort of watch relocation negligible.

Consider a clause deletion $\mathbf{d}{:}\,C$ at the c-th index, which removed the propagation cone P from the trace I_{c-1}, reintroducing afterwards a set $R \subseteq P$ of literals to obtain I_c. Let D be a clause in F_c watched on l and k, and assume it satisfies Invariants 1 and 3 at the c-th instruction. If $\overline{k}$ and $\overline{l}$ do not occur in P, it is easy to check that both invariants also hold at the $(c-1)$-th instruction. In other words: the watch relocation explained in Section III-C is only needed for clauses in the watchlist of $\overline{l}$ for every literal l in the propagation cone.

[2] An anonymous reviewer pointed out that `MiniSAT` contains a similar algorithm in its `analyzeFinal` function [3].

V. Experimental evaluation

The ideas described in this paper were implemented in a proof-of-concept DRAT checker rupee. Our DRAT checker can be run in operational or specified modes; the operational mode is designed to be as close as possible to a standard DRAT checker, whereas the specified mode includes the unit deletion processing methods described in this paper. Being a proof-of-concept implementation, this checker lacks of many optimizations, including efficient proof parsing, exploitation of CPU cache, core-first propagation, and resolution candidate caching. We thus expect worse performance than state-of-the-art checkers. However, our goal is to measure the overhead induced by specified-DRAT checking compared to operational-DRAT checking, and for this we needed a system that we completely understood to minimally change the behavior between the two modes. To the best of our knowledge, there is no reason to think that the aforementioned optimizations are incompatible with our methods for specified-DRAT checking.

An LRAT certificate [2] can be generated for instances that rupee reports as correct. For instances reported as incorrect, rupee reports information on the state of the trace at the end of RUP and RAT checks on failing instructions. To the best of our knowledge, rupee reports the right result in both modes.

We selected 11 benchmarks which were solved fast by solvers in the SAT Competition 2017. DRAT proofs for these benchmarks were generated by 4 participant solvers: COMiniSatPS_Pulsar_drup, glucose-4.1, Maple_LCM_Dist, and cadical-sc17-proof. The 44 resulting proofs were checked with rupee in both modes, as well as with the state-of-the-art DRAT-trim as a baseline[3].

DRAT-trim and rupee in operational mode agree on all instances, as expected; rupee in specified mode only agrees on 18 instances, rejecting all remaining instances. Hence, discrepancies between specified-DRAT and operational-DRAT occur rather frequently. Despite the semantic complexity of the interaction between RAT introduction and clause deletion [14], [15], this is not the cause of discrepancies: none of the discrepant proofs contains RAT clauses. The distribution of the discrepancies gives some insight in this regard: cadical-sc17-proof produced no discrepancies; for the other three solvers 8 out of 11 proofs were discrepant. We conjecture that the cause of discrepancies may be in the MiniSAT patch which most checkers use for proof generation in the CDCL loop, since cadical-sc17-proof implements its own method.

Figure 6 shows runtime results. We only compared results on instances where all three checkers accepted the proof; comparing discrepant instances would be meaningless, since execution stops as soon as an instruction is declared incorrect. DRAT-trim performs about one order of magnitude better than rupee; this is expectable due to the lack of optimizations in our tool. However, the runtimes of rupee in both its modes are comparable, with the specified mode outperforming the operational mode in hard instances. We conclude that the overhead of checking specified-DRAT proofs as compared to operational-DRAT proofs can be made negligible. Further research is required to verify the observed speed-up; one possible explanation would be that, by deleting more clauses in the specified mode, less resolution candidates are available for RAT checks, and so less RUP check calls need to be made.

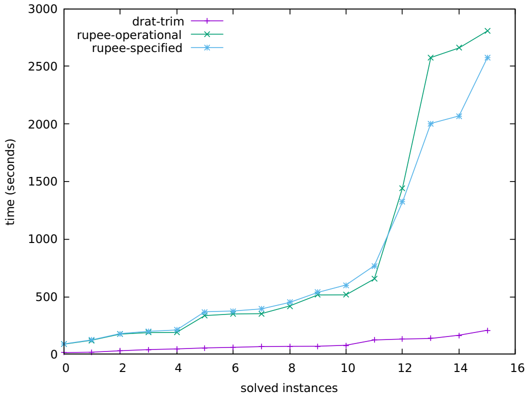

Fig. 6. Performance of DRAT-trim compared to both versions of rupee.

VI. Conclusion

The notion of a correct DRAT proof in the specification differs from the used in the implementation of DRAT checkers. We discussed the practical reasons for this, which lie on data structure invariants that are broken if the original definition of DRAT were to be respected. We proposed several changes in DRAT checkers' data structures and algorithms to check DRAT proofs according to the specification in an efficient way. In particular, we explained how to maintain slightly more intricate invariants so that unit clause deletions can be applied, and explored ways to vastly reduce the induced overhead.

We implemented these enhanced algorithms in a tool rupee, and used it to verify DRAT proofs produced by modern SAT solvers. Our results show that the discrepancy between the DRAT definition and the operational notion of correctness arises relatively often in practice. Our tool has a negligible overhead over checking with respect to the operational semantics, although further efforts in optimization must be done in order to attain similar performance to state-of-the-art DRAT checkers. Our data also suggests that discrepancies might have their root cause in an anomalous behavior of the CDCL proof logging method underlying many solvers. This suggests that future work should be directed towards efficient, specification-complying proof generation.

Acknowledgments: We would like to acknowledge anonymous reviewers who pointed out several relevant details. 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.

[3] https://github.com/arpj-rebola/fmcad2018

References

[1] Tomas Balyo, Marijn J. H. Heule, and Matti Järvisalo. SAT competition 2016: Recent developments. In *AAAI Conference on Artificial Intelligence*, pages 5061–5063, 2017.

[2] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *CADE*, volume 10395 of *LNCS*, pages 220–236. Springer, 2017.

[3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919, pages 502–518. Springer, 2004.

[4] Walter Forkel, Tobias Philipp, Adrian Rebola-Pardo, and Elias Werner. Fuzzing and verifying RAT refutations with deletion information. In *Florida Artificial Intelligence Research Society Conference*, pages 190–193. AAAI Press, 2017.

[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, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving*, volume 10499 of *LNCS*, pages 269–284. Springer, 2017.

[8] 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.

[9] 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.

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

[11] 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.

[12] Peter Lammich. Efficient verified (UN)SAT certificate checking. In *CADE*, volume 10395 of *LNCS*, pages 237–254. Springer, 2017.

[13] 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.

[14] 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.

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

[16] 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.