Compositional Propositional Proofs *

Marijn J.H. Heule[1] and Armin Biere[2]

[1] Department of Computer Science, The University of Texas at Austin, USA
`marijn@cs.utexas.edu`
[2] Institute for Formal Models and Verification, JKU Linz, Austria `biere@jku.at`

Abstract. Many hard-combinatorial problems have only be solved by SAT solvers in a massively parallel setting. This reduces the trust one has in the final result as errors might occur during parallel SAT solving or during partitioning of the original problem. We present a new framework to produce clausal proofs for cube-and-conquer, arguably the most effective parallel SAT solving paradigm for hard-combinatorial problems. The framework also provides an elegant approach to parallelize the validation of clausal proofs efficiently, both in terms of run time and memory usage. We evaluate the presented approach on some hard-combinatorial problems and validate constructed clausal proofs in parallel.

1 Introduction

Several long-standing open problems have recently been solved with SAT solvers, including the Erdős discrepancy conjecture [1], van der Waerden numbers [2,3], and optimal sorting networks [4]. These problems have been open for decades and only SAT techniques were able to make progress. Ever since the four-color theorem was solved using heavy computer assistance [5], there have been doubts about the correctness of such results as it is impossible for humans to verify the proof [6]. For most impressive applications of SAT technology, proofs are not provided, since their size would be enormous and due to the absence of validation tools. We present a method and tools to generate and validate *compositional propositional proofs* to increase confidence in the results for such problems.

Unsatisfiability proofs (or refutations) are traditionally expressed as either *resolution proofs* [7] or *clausal proofs* [8]. A proof is a sequence of *lemmas*, i.e., redundant clauses, which when added to the formula preserve satisfiability. Resolution proofs explicitly state which clauses should be resolved to derive a lemma, making them too verbose for hard problems. This detailed information is absent in clausal proofs, leaving it up to the clausal proof checker to determine why a lemma is redundant. Practically all top-tier SAT solvers support clausal proof logging in the DRAT format [9], which was used to check the SAT Competition 2014 results. This paper focuses on how to make compositional DRAT proofs.

Classical propositional proof systems, such as resolution, are of course compositional, in the sense that concatenating two proofs derives the union of the conclusions of both proofs. However, clausal proofs also support clause deletion to realize efficient validation [10] and expressing techniques that do not preserve logical equivalence — in contrast to resolution proofs. Thus, we must come up with a compositional proof system for clausal proofs that *includes deletion information* and operations that do not preserve logical equivalence.

One of the major obstacles for checking proofs obtained from parallel solvers, such as the proofs from portfolio solvers [11], is the huge gap between the time to solve a problem and time to validate the corresponding proof — even with deletion information. One reason for this gap is that the solver runs on all cores of a machine, while a checker uses only one. We address this problem by partitioning proofs in such a way that the validation can be performed in parallel.

The recent SAT result on the Erdős discrepancy conjecture [1] produced a 13 gigabyte clausal proof —comparable to the compressed size of all English text on Wikipedia— and validated it using the `DRATtrim` checker [9]. The ability to verify that result significantly increased the confidence with regards to its correctness. However, for most hard-combinatorial problems that have been solved with SAT solvers no such proof exists: e.g., van der Waerden number $W(2,6)$ [2] and the optimality result of sorting networks with nine wires [4]. These problems require enormous SAT solving time resulting in proofs that are terabytes in size.

One of the leading parallel SAT solving paradigms is *cube-and-conquer* [12], which uses a lookahead solver to generate millions of *cubes* for a conflict-driven clause learning (CDCL) solver. Cube-and-conquer is particularly effective on hard-combinatorial problems where it can heavily outperform both lookahead and CDCL solvers, even on a single core machine. We present a method which allows to produce proofs for problems solved by parallel cube-and-conquer.

Our paper proceeds by presenting some preliminaries in Section 2. Section 3 introduces rules regarding compositional propositional proofs. We provide in Section 4 a method to validate clausal proofs in parallel. In Section 5, we show how to log proofs in parallel for cube-and-conquer solvers. Our tools are presented in Section 6. We give an evaluation in Section 7, and we conclude in Section 8.

2 Preliminaries

CNF Satisfiability. For a Boolean variable x, there are two *literals*, the positive literal x and the negative literal $\bar{x}$. A *clause* is a disjunction of literals and a CNF formula a conjunction of clauses. A clause can be seen as a finite set of literals and a CNF formula as a finite set of clauses. A truth assignment is a function τ that maps literals to $\{\mathbf{f}, \mathbf{t}\}$ under the assumption $\tau(x) = v$ if and only if $\tau(\bar{x}) = \neg v$. A clause C is satisfied by τ if $\tau(l) = \mathbf{t}$ for some literal $l \in C$. An assignment τ satisfies F if it satisfies every clause in F. Two formulas are *logically-equivalent* if they are satisfied by exactly the same set of assignments, and *satisfiability-equivalent* if both formulas are satisfiable or both unsatisfiable.

Resolution and Extended Resolution. The resolution rule states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We say C is the *resolvent* of C_1 and C_2. C is logically implied by any formula containing C_1 and C_2. For a given CNF formula F, the *extension rule* [13] allows one to iteratively add definitions of the form $x := a \wedge b$ by adding the *extended resolution clauses* $(x \vee \bar{a} \vee \bar{b}) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee b)$ to F, where x is a new variable and a and b are literals in the current formula.

Unit Propagation. For a CNF formula F, *unit propagation* simplifies F based on unit clauses ; that is, it repeats the following until fixpoint: if there is a unit clause $(l) \in F$, remove all clauses that contain the literal l from the set $F \setminus \{(l)\}$ and remove the literal $\bar{l}$ from all clauses in F. If unit propagation on formula F produces complementary units (l) and $(\bar{l})$, we say that unit propagation *derives a conflict* and write $F \vdash_1 \epsilon$ with ϵ referring to the (unsatisfiable) empty clause.

Example 1. Consider the formula $F = (a) \wedge (\bar{a} \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$. We have $(a) \in F$, so unit propagation removes literal $\bar{a}$, resulting in the new unit clause (b). After removal of the literals $\bar{b}$, two complementary unit clauses (c) and $(\bar{c})$ are created. From these two units the empty clause can be derived: $F \vdash_1 \epsilon$.

Clause Redundancy. A clause C is called *redundant* with respect to a formula F if $F \wedge \{C\}$ is satisfiability equivalent to F. A *tautology* is a redundant clause that contains literals x and $\bar{x}$ for some variable x. A clause $C \in F$ is also redundant if there exists a clause $D \in F$ such that $D \subseteq C$, i.e., D *subsumes* C.

Asymmetric tautologies, also known as *reverse unit propagation* (RUP) clauses, are the most common redundant (learned) clauses in CDCL SAT solvers. Let $\overline{C}$ denote the conjunction of unit clauses that falsify all literals in C. A clause C is an asymmetric tautology with respect to a CNF formula F if $F \wedge \overline{C} \vdash_1 \epsilon$. *Resolution asymmetric tautologies* (or RAT clauses) [14] are a generalization of both asymmetric tautologies and extended resolution clauses. A clause C has RAT on $l \in C$ (referred to as the *pivot* literal) with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$, it holds that $F \wedge \overline{C} \wedge (\overline{D} \setminus \{(l)\}) \vdash_1 \epsilon$.

Not only can RAT be computed in polynomial time, but all preprocessing, inprocessing, and solving techniques in state-of-the-art SAT solvers can be expressed in terms of addition and removal of RAT clauses [14].

Clausal Proofs. A *proof of unsatisfiability* (also called a *refutation*) is a sequence of redundant clauses, called *lemmas*, containing the empty clause. There are two prevalent types of unsatisfiability proofs: *resolution proofs* and *clausal proofs*. Several formats have been designed for resolution proofs [7,15,16], but they all share the same disadvantages. Resolution proofs are often huge, and it is hard to express important techniques, such as conflict clause minimization, with resolution steps. Other techniques, such as bounded variable addition [17], cannot be polynomially-simulated by resolution. Clausal proof formats [9,18,19] are syntactically similar; they involve a sequence of clauses that are claimed to be

redundant with respect to a given formula. It is important that the redundancy property of clauses can be checked in polynomial time.

A *DRUP proof*, short for *Deletion Reverse Unit Propagation*, is a sequence of addition and deletion steps of RUP clauses. A *DRAT proof*, short for *Deletion Resolution Asymmetric Tautology*, is a sequence of addition and deletion steps of RAT clauses. A DRAT *refutation* is a DRAT proof that contains the empty clause. Figure 1 shows an example DRAT refutation.

Example 2. Consider the CNF formula $F = (a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (b \vee c \vee \bar{d}) \wedge (\bar{b} \vee \bar{c} \vee d) \wedge (a \vee c \vee d) \wedge (\bar{a} \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee b \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$, shown in DIMACS format in Fig. 1 (left), where 1 represents a, 2 is b, 3 is c, 4 is d, and negative numbers represent negation. The first clause in the proof, $(\bar{a})$, is a RAT clause with respect to F because all possible resolvents are asymmetric tautologies:

$$F \wedge (a) \wedge (\bar{b}) \wedge (c) \vdash_1 \epsilon \quad \text{using} \quad (a \vee b \vee \bar{c})$$
$$F \wedge (a) \wedge (\bar{c}) \wedge (\bar{d}) \vdash_1 \epsilon \quad \text{using} \quad (a \vee c \vee d)$$
$$F \wedge (a) \wedge (b) \wedge (d) \vdash_1 \epsilon \quad \text{using} \quad (a \vee \bar{b} \vee \bar{d})$$

3 Rules

In this section, we introduce rules for composing propositional proofs. We first establish a notation to describe operations, or *derivations* on formulas and proofs, and then continue with basic rules for addition or deletion of a clause/lemma. Finally, we propose compositional rules that address merging proofs produced in parallel and validating proofs that have been produced in parallel.

Throughout this section, we will use $\bigcirc$ to express a CNF formula and $\triangle$ to express a proof composed of a sequence of proof steps. Note that a formula is a multi-set of clauses as elements may be duplicated by some of the operations below. Furthermore, a formula may be treated as a part of a proof by treating each clause as an added lemma. Concatenation of proofs is simply the concatenation of the sequences of their proof steps. The union of two formulas is interpreted as multi-set union. Both operations are denoted by juxtaposition.

3.1 The Base Rules

Each element of a derivation is either the addition of a clause C, denoted by $a(C)$ or the deletion of a clause C, denoted by $d(C)$. Given a formula $\bigcirc_i$, a clause C and a modification $m \in \{a, d\}$, a proof step is denoted as

$$\bigcirc_i \xrightarrow{m(C)} \bigcirc_{i+1}$$

We introduce two base rules ADD and DEL which produce atomic proof steps.

$$\text{ADD:} \quad \frac{}{\bigcirc \xrightarrow{a(C)} \bigcirc C} \quad \text{where } C \text{ has RAT on } l \in C \text{ w.r.t. } \bigcirc$$

$$\text{DEL:} \quad \frac{}{\bigcirc C \xrightarrow{d(C)} \bigcirc} \quad \text{(no side condition)}$$

The ADD rule has the precondition that there exists a literal $l \in C$ such that C had RAT on l with respect to the formula $\bigcirc$. The correctness of the ADD rule follows from the observation that the addition of RAT clauses preserves satisfiability. Practically all techniques used in modern CDCL SAT solvers can be simulated by these rules bas they can be expressed as a RAT derivation[3] [14].

The DEL rule has no precondition, and the removal of clauses from a formula is always allowed. We are only interested in proofs of unsatisfiability and deletion of a clause trivially preserves satisfiability. For proofs of satisfiability, the situation is reversed: the (same) precondition is required for the DEL rule, while the ADD rule has no precondition. The most important function of the DEL rule is to facilitate fast validation of proofs. Without clause deletion, validation costs can be two orders of magnitude larger on reasonable-sized proofs. The achievable speed-up factor increases for larger proofs.

A *DRAT derivation* is a sequence of proof steps that consists for each step i of a clause C_i and a modification $m_i \in \{a, d\}$. Applying a DRAT derivation of n steps to a CNF formula $\bigcirc_0$ results in $\bigcirc_n$ by applying each step in the order in which they occur in the derivation. A DRAT derivation of n steps is *valid* for a given formula $\bigcirc_0$ if for all steps $i \in \{1..n\}$ holds that C_i has RAT on a $l \in C_i$ w.r.t. $\bigcirc_{i-1}$ if $m_i = a$ and C_i occurs in $\bigcirc_{i-1}$ if $m_i = d$. Consider the proof:

$$\bigcirc_0 \xrightarrow{m_1(C_1)} \bigcirc_1 \xrightarrow{m_2(C_2)} \bigcirc_2 \ldots \bigcirc_{n-1} \xrightarrow{m_n(C_n)} \bigcirc_n$$

We say, the proof $\triangle = m_1(C_1)m_2(C_2)\cdots m_n(C_n)$ gives a *derivation* from $\bigcirc_0$ to $\bigcirc_n$, or in symbols

$$\bigcirc_0 \xrightarrow{\overbrace{m_1(C_1)m_2(C_2)\ldots m_n(C_n)}^{\triangle}} \bigcirc_n \qquad \text{or} \qquad \bigcirc_0 \xrightarrow{\triangle} \bigcirc_n$$

We also represent rules as a triple containing: a *pre-CNF* $\bigcirc_{\text{pre}}$, a proof $\triangle$, and a *post-CNF* $\bigcirc_{\text{post}}$, denoting that proof $\triangle$ is a derivation from $\bigcirc_{\text{pre}}$ to $\bigcirc_{\text{post}}$.

---

[3] All solver techniques can be expressed as a RAT derivation. For some techniques, such as symmetry-breaking, the construction of a RAT derivation is complex [20].

Definition 1. $(\bigcirc_{\mathrm{pre}}, \triangle, \bigcirc_{\mathrm{post}})$ *is* valid *iff* $\bigcirc_{\mathrm{pre}} \xrightarrow{\triangle} \bigcirc_{\mathrm{post}}$ *is a derivation.*

The addition of RAT clauses preserves satisfiability [14], as does the deletion of any clause. Thus, we get the following soundness result for valid compositional triples and derivations respectively.

Proposition 1. *Given a valid composition triple* $(\bigcirc_{\mathrm{pre}}, \triangle, \bigcirc_{\mathrm{post}})$, *if* $\bigcirc_{\mathrm{pre}}$ *is satisfiable then* $\bigcirc_{\mathrm{post}}$ *is satisfiable as well.*

In practice, we focus on the contrapositive, e.g., if $\bigcirc_{\mathrm{post}}$ contains the empty clause then $\bigcirc_{\mathrm{pre}}$ is unsatisfiable, and we consider $\triangle$ to be a proof (refutation) for the unsatisfiability of $\bigcirc_{\mathrm{pre}}$.

3.2 The Composition Rules

In this section, the notion of a satisfiability-preserving derivation, as defined in the previous section, will be lifted to the compositional case.

In addition to the two base rules ADD and DEL, we propose two composition rules which combine two compositional triples into one. The first rule SEQ, short for "sequential", combines two compositional triples for which the post-CNF of one triple equals the pre-CNF of the other triple. The second rule PAR, short for "parallel", combines two compositional triples for which the two pre-CNFs are equal. Visualizations of the SEQ rule PAR rule can be found in Fig. 2.

$$\text{SEQ:} \quad \frac{\bigcirc_0 \xrightarrow{\triangle_1} \bigcirc_1 \quad \bigcirc_1 \xrightarrow{\triangle_2} \bigcirc_2}{\bigcirc_0 \xrightarrow{\triangle_1 \triangle_2} \bigcirc_2} \qquad\qquad \text{PAR:} \quad \frac{\bigcirc_0 \xrightarrow{\triangle_1} \bigcirc_1 \quad \bigcirc_0 \xrightarrow{\triangle_2} \bigcirc_2}{\bigcirc_0 \xrightarrow{\bigcirc_0 \triangle_1 \triangle_2} \bigcirc_1 \bigcirc_2}$$
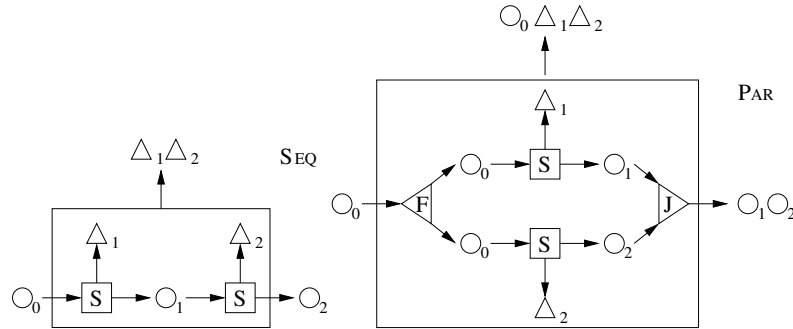
Fig. 2. Visualization of SEQ and PAR rules (S = Solve, F = Fork, J = Join). A solver (S) takes a formula as input and produces a modified formula as well as a derivation that describes the modifications. Forking (F) can be used to let two solvers work on the same formula. Internally this means that the formula is duplicated. If all added clauses preserve logical equivalence (DRUP), the resulting formulas can be joined (J).

The SEQ rule has no preconditions and can be used for any two valid compositional triples for which one pre-CNF is equal to the other post-CNF. We will

use this rule to develop a method to validate DRAT derivations in parallel. The soundness result for SEQ follows directly from the definition of how proofs are concatenated and formulas are joined.

Proposition 2. *Given two valid compositional triples as antecedents, then the* SEQ *rule produces a valid compositional triple as consequent.*

Note that validity of a compositional triple is still defined in terms of basic derivations which are sequences of addition and deletion steps. Thus these compositional rules allow one to generate a basic derivation from a "compositional proof", which in turn is sound.

The PAR rule expresses how to merge DRUP (*not* DRAT) derivations which are obtained by running multiple solvers running on the same pre-CNF in parallel. Notice that the merged derivation by the PAR rule start with a copy of the pre-CNF. The pre-CNF is included because both derivations may delete the same clause from the original formula.

Notice that the DRUP proof $\bigcirc_0 \triangle_1 \triangle_2$ in the conclusion of the PAR rule cannot be replaced by $\triangle_1 \bigcirc_0 \triangle_2$, because $\triangle_1$ may have eliminated clauses from $\bigcirc_0$ in such a way that a clause $C \in \bigcirc_0$ no longer has DRUP w.r.t. $\bigcirc_1$. For example consider an unsatisfiable formula $\bigcirc_0$, let $\bigcirc_1$ be the empty formula, and let $\triangle_1$ simply remove all clauses from $\bigcirc_0$ (without adding anything). Clearly, $\triangle_1 \bigcirc_1$ is a valid DRUP proof for $\bigcirc_0$ as it contains only deletion information. However, it is not possible to create a valid DRUP proof by appending $\bigcirc_0$ to $\triangle_1 \bigcirc_1$ because $\bigcirc_1$ is satisfiable and $\bigcirc_0$ unsatisfiable.

Proposition 3. *Given two valid compositional triples as antecedents with DRUP proofs, then the* PAR *rule produces a valid compositional triple with DRUP proof as consequent.*

Proof. (sketch) All the added $\bigcirc_0$ clauses in the combined proof are valid DRUP clauses (since they occur in the pre-CNF $\bigcirc_0$ and are even subsumed). Further note, that DRUP is monotonic, in the sense, that if a clause has the DRUP property w.r.t. $\bigcirc$ it will also have DRUP w.r.t. all $\bigcirc'$ with $\bigcirc \subseteq \bigcirc'$ (as multi-sets). Thus adding $\bigcirc_0$ in front of $\triangle_1$ does not destroy the property of $\triangle_1$ to be a derivation. Because we use a multi-set interpretation for formulas all the clauses in $\triangle_1$ are still in the intermediate formula reached after the sub-proof $\bigcirc_0 \triangle_1$ and $\triangle_2$ just works as before, also keeping all the derived $\bigcirc_1$ clauses in the post-CNF in addition to deriving all its own $\bigcirc_2$ clauses. $\qquad \square$

The above argument does not hold for DRAT proofs (instead of DRUP), because DRAT is not monotonic: A clause C can have RAT w.r.t. a formula F, but not with respect to $F \wedge G$ for some formula G. Hence, $\triangle_1$ may add a clause which breaks the RAT property of a clause addition step in $\triangle_2$.

As an optimization, to avoid the duplication of the original clauses in the PAR rule, one can consider a modified rule, which has a side condition that neither $\triangle_1$ nor $\triangle_2$ eliminate clauses from $\bigcirc_0$.

4 Parallel Proof Checking

Existing tools to validate clausal proofs, such as `DRATtrim` [9] and our new proof checker `DRABT`, can check proofs of reasonable size (dozens of gigabytes) efficiently (within in a day). Yet existing tools are not well-equipped to deal with huge proofs because they keep the full proof in memory and validation is done on a single core. In this section, we present a method to validate DRAT proofs in parallel effectively with only a few changes to existing proof-checking tools.

4.1 Proofs Checking Optimizations

There are several optimizations that make the efficient, serial validation of clausal proofs possible. The most significant gains can be realized by exploiting deletion information in proofs. Ignoring deletion information can increase the validation costs by two orders of magnitude on reasonable-sized proofs of say several gigabytes [10,21]. We will provide an example of the impact of deletion information on the validation costs in the introduction of Section 7.

One, so far unpublished, optimization in `DRATtrim` and `DRABT` is ignoring deletion information of unit clauses or *pseudo-unit clauses*, i.e., clauses that have become unit under the top-level assignment. For example, (a) is a unit clause in formula $F := (a) \wedge (\bar{a} \vee b)$, while $(\bar{a} \vee b)$ is a pseudo-unit clause. Deleting (pseudo-)unit clauses during proof checking can be very costly as the checker has to unassign all variables and compute a new top-level assignment. When a proof claims to show unsatisfiability, the deletion of unit clauses is not useful.

Enhancing a clausal proof with deletion information can be somewhat tricky. While working on this paper, we discovered that there is a bug in the proof logging of several CDCL SAT solvers. The bug is caused by deleting pseudo-unit clauses without first adding the corresponding unit clauses to the proof. Due to this bug, many clausal proofs produced by these solvers are invalid, which would have been reported by the checker if it did not ignore the deletion of (pseudo-)unit clauses. We even observed cases where the intermediate formula becomes satisfiable after the invalid deletion of a pseudo-unit clause. Appendix A offers details and a fix for this bug for the SAT solver `Glucose`.

4.2 Backward Checking of Derivations

The validation of a clausal proof for a given formula requires checking the validity of each clause addition step, i.e., the precondition of the ADD rule. This can be implemented using *forward checking*: go over the proof from the start to end, modify the formula at each step, and check the validity of addition steps.

A refutation can also be validated using *backward checking* [8]: First, mark the empty clause as a *core* lemma, i.e., a lemma that needs to be validated. Second, process the proof in reverse order and only validate the addition of core lemmas, assuming that all added lemmas occurring earlier in the proof — and that are not deleted prior to the checked core lemma — can be validated. Checking a core lemma may mark other lemmas occurring earlier in the proof as

core. Successful backward checking does not imply that the original proof was valid, but that a new valid proof was obtained that consists of the sequence of added and deleted core lemmas. The order of the lemmas in the new proof will match the order of the lemmas in the original proof. Backward checking enables optimizating deletion information, i.e., the clause deletion steps in the proof [21].

Backward checking can be generalized for arbitrary derivations to check the validness of compositional triples efficiently. Instead of marking the empty clause as core, initially all lemmas occurring in the derivation that are not deleted will be marked as core. Furthermore, it is allowed to unmark a marked lemma if it is subsumed by another marked lemma or by a clause in the pre-CNF which will not be deleted. This can be computed efficiently using backward subsumption [22]. Notice that when this restriction is applied, backward checking for refutations is unaffected, because the empty clause subsumes all other lemmas. Recall, successful backward checking does not guarantee that the original derivation is valid, but only that a new valid derivation was obtained.

4.3 Parallel Proof Checking via Seq Rule

The Seq rule provides an elegant method for validating DRAT proofs in parallel: given a CNF formula $\bigcirc_0$ and DRAT refutation $\triangle$, partition $\triangle$ into k derivations such that $\triangle_1 \triangle_2 \ldots \triangle_k = \triangle$. Second, compute the pre- and post-CNFs $\bigcirc_i$, where $\bigcirc_i$ denotes the result of applying derivation $\triangle_i$ to formula $\bigcirc_{i-1}$. Notice that this cannot be done in parallel because the computation of $\bigcirc_{i+1}$ depends on the existence of $\bigcirc_i$. Finally, check that all $(\bigcirc_{i-1}, \triangle_i, \bigcirc_i)$ with $i \in \{1..k\}$ are valid compositional triples and that $\epsilon \in \bigcirc_k$. When all checks are successful, the Seq rule states that $\triangle$ is a valid refutation for $\bigcirc_0$. Below it is shown in symbols how to deduce the validness of refutation $\triangle$ by applying the Seq rule $k-1$ times:

$$\frac{\bigcirc_0 \xrightarrow{\triangle_1} \bigcirc_1 \quad \bigcirc_1 \xrightarrow{\triangle_2} \bigcirc_2 \quad \ldots \quad \bigcirc_{k-1} \xrightarrow{\triangle_k} \epsilon}{\bigcirc_0 \xrightarrow{\triangle_1 \triangle_2 \ldots \triangle_k} \epsilon}$$

4.4 Validating the Post-CNF

One method to check that $(\bigcirc_{\mathrm{pre}}, \triangle, \bigcirc_{\mathrm{post}})$ is a valid compositional triple is to check that $\triangle$ is a valid derivation for $\bigcirc_{\mathrm{pre}}$ and assumes that the computation of the $\bigcirc_{\mathrm{post}}$ was done correctly. As partitioning problems can easily result in errors, confidence in the correctness of the complete proof checking chain can be improved by fully validating compositional triples. One can explicitly check that $\bigcirc_{\mathrm{post}}$ is derived from $\bigcirc_{\mathrm{pre}}$ by applying $\triangle$. This is implemented in our checker **DRABT** by hashing, which requires $\bigcirc_{\mathrm{post}}$ to be provided as a third input file.

Alternatively, one can increase confidence in the tool chain by checking that $\triangle\bigcirc_{\mathrm{post}}$ is a valid derivation for $\bigcirc_{\mathrm{pre}}$. This validates that there exists a valid compositional triple $(\bigcirc_{\mathrm{pre}}, \triangle', \bigcirc_{\mathrm{post}})$ and the checker should able to produce $\triangle'$. In practice, appending $\triangle$ with $\bigcirc_{\mathrm{post}}$ can significantly increase the costs of validating proofs as many clauses in post-CNF $\bigcirc_{\mathrm{post}}$ occur also in pre-CNF

$\bigcirc_{\mathrm{pre}}$. Validating such clauses will mark the equivalent clauses in the pre-CNF as core, which will obstruct the core-first optimization of proof checking [21]. If the computation of the post-CNFs was done correctly, all clauses in the post-CNFs will be unmarked and hence not be validated.

5 Parallel Proof Generation

Traditionally, proof generation has only been supported by non-parallel SAT solvers. A recent study [11] presented an approach to construct clausal proofs from clause-sharing portfolio parallel SAT solvers. The proofs constructed with that method were very costly to validate. In this section, we present a method to construct clausal proofs from parallel SAT solvers based on the cube-and-conquer paradigm, such as `march_cc+iLingeling`. The experimental evaluation shows that these proofs can be validated in parallel efficiently.

In short, cube-and-conquer solvers consist of two parts: a lookahead (or cube) solver and a CDCL (or conquer) solver. First, the cube solver partitions the problem into many subproblems, frequently millions. Each of the subproblems is represented by a cube, i.e., a conjunction of literals. In the second phase, one or more CDCL solvers will use these cubes to guide their search. Clauses learned while solving a cube are typically not useful for solving other cubes. One can solve cubes massively in parallel and obtain almost a linear time speed-up with the number of solvers — assuming that there are as many cores as solvers.

We now show how to construct a DRUP refutation for cube-and-conquer solvers. First, the cube solver computes cubes for the input formula. guide the conquer solvers. Assume that we have k conquer solvers S_i with $i \in \{1..k\}$. Each solver S_i gets a set of cubes $\overline{\bigcirc_i}$. After solver S_i refutes all of its cubes, it generates a DRUP proof $\triangle_i$ that expresses how to produce all clauses $\bigcirc_i$ from the original formula $\bigcirc_0$ and deletes all the other learned and original clauses. Then, a refutation is computed for the conjunction of all cubes $\bigcirc_1 \bigcirc_2 \ldots \bigcirc_k$, the conquer proof $\triangle_c$.

The composition rules explain how to merge these derivations in a refutation for the input formula. First, all $\triangle_i$ derivations are merged using the PAR rule, by starting with the $k-1$ copies of the input formula and adding the concatenation of the derivations. Second, the merged derivation is combined with the conquer proof using the SEQ rule.

$$
\cfrac{
\cfrac{\bigcirc_0 \xrightarrow{\triangle_1} \bigcirc_1 \quad \bigcirc_0 \xrightarrow{\triangle_2} \bigcirc_2 \quad \ldots \quad \bigcirc_0 \xrightarrow{\triangle_k} \bigcirc_k}
{\bigcirc_0 \xrightarrow{\bigcirc_0 \ldots \bigcirc_0 \triangle_1 \triangle_2 \ldots \triangle_k} \bigcirc_1 \bigcirc_2 \ldots \bigcirc_k \qquad \bigcirc_1 \bigcirc_2 \ldots \bigcirc_k \xrightarrow{\triangle_c} \epsilon}
}
{\bigcirc_0 \xrightarrow{\bigcirc_0 \ldots \bigcirc_0 \triangle_1 \triangle_2 \ldots \triangle_k \triangle_c} \epsilon}
$$

6 Tools

We have implemented several tools to support compositional proof generation and validation, which are available at `www.cs.utexas.edu/~marijn/cpp` and at

`http://fmv.jku.at/drabt`. The `DRATtrim` proof checking tool [9] was enhanced to support backward checking [8] for arbitrary DRAT derivations. The previous version only supported backward checking of refutations. We improved the speed of validating DRAT derivations by unmarking all lemmas that are subsumed by other marked lemmas or undeleted clauses in the pre-CNF, see Section 4.2.

We also added a new feature, called *proof application*, to `DRATtrim`: Given an input formula (the pre-CNF) and a DRAT proof , the tool computes the post-CNF formula that would be the result of applying the proof to pre-CNF. In other words, the post-CNF contains all clauses in pre-CNF that are not deleted in the proof together with all lemmas in proof that are added (and not deleted). Proof application facilitates parallel proof checking via the SEQ rule, see Section 4.3.

To further increase confidence in the results, the second author independently implemented a new clausal proof checker, called `DRABT`. The current version of `DRABT` supports forward checking of DRUP proofs and implements checking validity of compositional triples natively in contrast to `DRATtrim` which checks it implicitly by appending the post-CNF to the proof. The `DRABT` tool puts also much more focus on proper error messages as well as improved diagnostic capabilities if an error occurs. It is, however, missing core generating features.

The SAT solver `march_cc` [12] can be used in a cube-and-conquer setting to produce cubes to guide a conquer solver. We had to slightly change `march_cc` in order to use it for compositional propositional proofs. The change consists of extending the cube output with all the branches that `march_cc` was able to refute using lookahead techniques. Without those cubes, the cube output does not cover the entire search space — which would cause the proof checker to fail. We observed that the cubes which can be refuted by lookahead techniques are also easy for a CDCL solver to refute. Consequently, adding these cubes to the cube output hardly increases the overall performance.

The CDCL solver `iLingeling` [12] is a parallel SAT solver that solves benchmarks in the iCNF format[4], which combines a CNF formula with a sequence of cubes that guide the solver. We extended `iLingeling` with DRUP proof logging support. The `iLingeling` solver runs multiple `Lingeling` solvers in parallel and guides them using the cubes. Each of these `Lingeling` solvers emits its own DRUP proof. Additionally, a separate `Lingeling` solver computes a proof for the cube file, the so-called conquer proof.

7 Evaluation

In this section, we evaluate parallel proof generation based on the PAR rule and parallel proof validation based on the SEQ rule. All experiments were performed on the Stampede cluster of the Texas Advanced Computing Center (TACC) which has two 8-core Xeon E5 processors and 32GB of memory per node.

Before describing the experiments, we want to reiterate the importance of deletion information in clausal proofs: on the smaller proofs discussed in this

---

[4] see `http://www.siert.nl/icnf/` for details

section, ignoring the deletion information would increase the validation costs by a factor of 20. For the large proofs, this increases to two orders of magnitude.

7.1 Parallel Compositional Proof Checking

We evaluated our parallel proof checking method on some existing DRAT proofs focusing on the speed-up in wall-clock time. Our method consists of multiple phases, some of which can be parallelized while other cannot. The first phase is partitioning a given proof $\triangle$ into k derivations: $\triangle_1, \ldots, \triangle_k$. This can simply be realized by the Unix utility `split`, the computational costs of which are practically ignorable. In the second phase, we need to compute the pre- and post-CNFs for proof checking, which is performed by `DRATtrim` using the new "proof application" mode. As described in Section 4.3, this part cannot be done in parallel. However, one could preprocess the derivations in parallel by removing all lemmas that are added and deleted within the same partial proof, because these lemmas will not influence the creation of the pre- and post-CNFs. Since most lemmas are added and deleted in the same proof, such preprocessing could significantly reduce the cost of this phase. This is not yet implemented. The third phase consists of checking that all $(\bigcirc_{i-1}, \triangle_i, \bigcirc_i)$ are valid compositional triples. We checked all proofs running $k = 16$ `DRATtrim` executables in parallel in the default mode, which validates a partial proof using backward checking and checks the post-CNFs implicitly via subsumption.

Table 1 shows the usefulness of parallel proof checking of proofs expressing symmetry-breaking techniques[5]. There are several interesting observations. First, the speed-up of checking derivations in parallel (on a 16-core machine) compared to checking them in serial is about a factor of nine on all instances when ignoring the initialization cost of splitting the proof and computing the pre- and post CNF. Taking these costs into account clearly reduces the speed-up on the smaller proofs. However, parallel proof checking is only interesting for large proofs. Second, checking derivations sequentially is more costly than checking the original refutation if the proofs are small. However, for the larger proofs the opposite happens. Third, computing the pre- and post-CNFs is quite costly for small proofs, but becomes relatively cheaper for larger proofs.

7.2 Parallel Proof Generation

For the evaluation of parallel proof generation based on the PAR rule, we used the cube-and-conquer solver `march_cc+iLingeling`. We picked a notoriously hard benchmark `eq.atree.braun.12.unsat.cnf` which has been used in several SAT competitions. This formula is a miter (a circuit equivalence-checking benchmark) which cannot be solved by sequential SAT solvers in hours and by very few parallel SAT solvers.

Figure 3 shows the results of the experiments, which were performed on a 16-core cluster node using 1, 2, 4, 8, or 16 cores. The solving process time is

---

[5] available on `http://www.cs.utexas.edu/~marijn/sbp/`

Table 1. Sequential versus parallel proof checking of DRAT proofs expressing symmetry-breaking techniques. The first column shows the benchmark name. The second and third column shows size of the original proof (in MB) and the `DRATtrim` checking time (in seconds). The fourth and fifth column show the time to split the proofs and to compute the pre- and post-CNFs, respectively. The last four columns show the costs to validate the derivations, sequentially, in parallel, and the speed-up with and without initialization costs on a 16-core machine.

benchmark	size	`DRATtrim`	`split`	CNFs	seq-chk	par-chk	$\frac{\text{seq+init}}{\text{par+init}}$	$\frac{\text{seq}}{\text{par}}$
EDP2_1161	2,180.98	3331.73	2.91	85.70	3288.78	455.93	6.20	7.21
R_4_4_18	20.01	2.55	0.04	1.91	4.19	0.43	2.58	9.74
tph6	2.78	0.61	0.01	1.25	2.03	0.22	2.22	9.23
tph7	5.09	1.30	0.02	1.39	2.70	0.29	2.41	9.31
tph8	10.68	2.98	0.03	1.61	4.29	0.46	2.82	9.32
tph9	34.18	6.17	0.04	1.98	7.33	0.83	3.28	8.83
tph10	19.86	11.78	0.06	2.51	12.67	1.32	3.92	9.60
tph11	56.49	22.96	0.09	3.39	22.64	2.85	4.13	7.94
tph12	92.29	39.42	0.15	4.73	39.07	3.89	5.01	10.04

very stable, close to 6,000 seconds. The wall-clock solving time of the conquer phase by `iLingeling` almost scales linearly in the number of cores. `iLingeling` emits a separate proof for each used `Lingeling` solver (one per core). For the experiment with k cores, each core validated one compositional triple consisting of the original formula (as pre-CNF), one of the proof files, and the precomputed post-CNF based on the pre-CNF and the proof. The size of the full proof is the concatenation of all these proofs together with the duplication of the original clauses due to the PAR rule.

We validated the proofs with both `DRABT` and `DRATtrim`. Figure 3 reports the `DRATtrim` times. The `Drabt` times, both process and wall-clock, were about twice as long. Notice that both the process and wall-clock time significantly drop when increasing the number of cores. The process time decreases by about a factor of 1.5 when doubling the number of cores. For the wall-clock time, the speed-up is close to a factor of 3 when doubling the number of cores. This indicates a
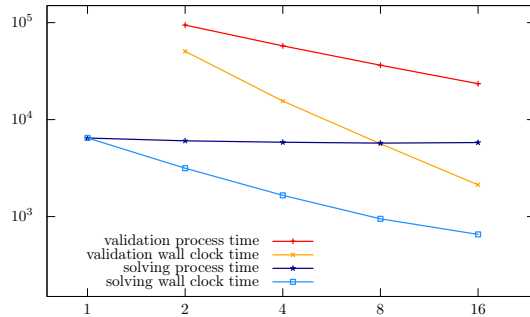
Fig. 3. A log-log plot of the effect of the number of cores (x-axis) on the wall-clock and process time (y-axis in seconds) to solve `eq.atree.braun.12.unsat.cnf` using `march_cc+iLingeling` and validate the emitted proof in parallel using `DRATtrim`. All experiments were performed on a single 16-core cluster node.

super-linear speed-up to validate proofs. Apparently, `DRATtrim` slows down when dealing with larger and larger proofs. This may be caused by an increase in the number of cache misses. Studying the reasons for the super-linear speed-up will be focus of future research.

8 Conclusion

SAT solvers have recently been used to tackle long-standing open problems. These problems are frequently solved in a massively parallel setting without emitting proofs to validate these results. Clausal proofs with deletion information are easy to emit from state-of-the-art, non-parallel SAT solvers, they are relatively compact, and they can be checked in a reasonable amount of time. However, for long-standing open problems, we need to construct clausal proofs of solvers based on arguably the most effective parallel SAT solving paradigm: cube-and-conquer. Additionally, we need tools to validate these proofs in parallel and bridge the gap between the solving and validation costs.

We presented the concept of compositional clausal proofs with deletion information. Following this concept, we developed and implemented an algorithm to validate clausal proofs in parallel effectively. Moreover, we show how to obtain clausal proofs from cube-and-conquer solvers and demonstrate how to validate those proofs in parallel. The experiments show that the speed-up can be super-linear in the number of cores.

References

1. Konev, B., Lisitsa, A.: Computer-aided proof of Erdős discrepancy properties. Artif. Intell. **224** (2015) 103–118
2. Kouril, M., Paul, J.L.: The van der Waerden number W(2, 6) is 1132. Experimental Mathematics **17**(1) (2008) 53–61
3. Kouril, M.: Computing the van der Waerden number $w(3, 4) = 293$. Integers **12** (2011) Paper A46, 13 p., electronic only
4. Codish, M., Cruz-Filipe, L., Frank, M., Schneider-Kamp, P.: Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In: ICTAI 2014, IEEE Computer Society (2014) 186–193
5. Appel, K., Haken, W.: The solution of the four-color-map problem. Sci Am **237**(4) (October 1977) 108–121
6. Aron, J.: Wikipedia-size maths proof too big for humans to check. New Scientist **221**(2957) (2014) 11
7. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE. (2003) 10880–10885

8. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE. (2003) 10886–10891
9. Wetzler, N., Heule, M.J.H., Hunt, Warren A., J.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Sinz, C., Egly, U., eds.: SAT 2014. Volume 8561 of LNCS. Springer (2014) 422–429
10. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.: Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. Software Testing, Verification, and Reliability (STVR) **24**(8) (2014) 593–607
11. Heule, M., Manthey, N., Philipp, T.: Validating unsatisfiability results of clause sharing parallel SAT solvers. In: Pragmatics of SAT. (2014) 12–25
12. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: HVC. Volume 7261 of LNCS., Springer (2011) 50–65
13. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2. Springer (1983) 466–483
14. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In Gramlich, B., Miller, D., Sattler, U., eds.: IJCAR. Volume 7364 of LNCS., Springer (2012) 355–370
15. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518
16. Biere, A.: Picosat essentials. JSAT **4**(2-4) (2008) 75–97
17. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Proceedings of Haifa Verification Conference 2012. (2012)
18. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: ISAIM. (2008)
19. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.: Verifying refutations with extended resolution. In: CADE. Volume 7898 of LNAI., Springer (2013) 345–359
20. Heule, M.J.H., Hunt, Warren A., J., Wetzler, N.: Expressing symmetry breaking in DRAT proofs. In: CADE-25. Volume 9195 of LNCS. Springer (2015) 591–606
21. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design, IEEE (2013) 181–188
22. Eén, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In: SAT 2005. Volume 3569 of LNCS., Springer (2005) 61–75

A Proof-logging Bug in CDCL Solvers

We observed a bug in the clausal proof logging of `Glucose` version 3.0, which actually occurs in all `MiniSAT`-based solvers — which is the majority of state-of-the-art solvers these days. The bug consists of deleting pseudo-unit clauses. This bug can simply be fixed by adding the following lines to `Solver.cc`:

```
if (certifiedUNSAT)
  for (int i = 0; i < c.size(); i++)
    if (reason(var(c[i])) == cr && level(var(c[i])) == 0)
      return;
```

just below the beginning of the removeClause procedure

```
void Solver::removeClause(CRef cr) {
  Clause& c = ca[cr];
```