# Validating Unsatisfiability Results of Clause Sharing Parallel SAT Solvers

Marijn J. H. Heule[1]*and Norbert Manthey[2] and Tobias Philipp[2]

[1] The University of Texas at Austin, United States
[2] Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany

**Abstract**

As satisfiability (SAT) solver performance has improved, so has their complexity, which make it more likely that SAT solvers contain bugs. One important source of increased complexity is clause sharing in parallel SAT solvers. SAT solvers can emit a proof of unsatisfiability to gain confidence that their results are correct. Such proofs must contain deletion information in order to check them efficiently. Computing deletion information is easy and cheap for parallel solvers without clause sharing, but tricky for parallel solvers with clause sharing.

We present a method to generate unsatisfiability proofs from clause sharing parallel SAT solvers. We show that the overhead of our method is small and that the produced proofs can be validated in a time similar to the solving (CPU) time. However, proofs produced by parallel solvers without clause sharing can be checked in a time similar to the solving (wall-clock) time. This raises the question whether our method can be improved such that the checking time of proofs from parallel solvers without clause sharing is comparable to the time to check proofs from parallel solver with clause sharing.

## 1 Introduction

Satisfiability (SAT) solvers have become more complex in recent years due to inprocessing techniques [21] and parallel computing [14, 18, 22]. This raises the question whether to results of these solvers can be trusted. To gain confidence in the correctness of the results, SAT solvers can emit *unsatisfiability proofs* that can be validated using a checker. The main challenge to check unsatisfiability results of SAT solvers is obtaining a relatively compact proof that can be validated in a reasonable time. We present a first approach in this direction for parallel SAT solvers with clause sharing [1, 11–13].

Initially, checking unsatisfiability results of SAT solvers was based on constructing a resolution proof [7, 27]. It was relatively easy to obtain a resolution proof of SAT solvers a decade ago, but due to the increased complexity it is harder for contemporary SAT solvers. Hardly any the state-of-the-art SAT solver can produce resolution proofs. Another major disadvantage of resolution proofs is their size. Resolution proofs of hard application benchmarks can be hundreds of gigabytes large. Additionally, emitting resolution proofs significantly increases the memory consumption. For sequential solvers memory usage can increase by two orders of magnitude [15]. For parallel SAT solvers, the space and memory requirements to emit resolution proofs will scale linearly in the number of solver incarnations. So the disadvantages of resolution proofs are even more severe for parallel SAT solvers compared to sequential SAT solvers.

*Supported by DARPA contract number N66001-10-2-4087, and the National Science Foundation under grant number CCF-1153558.

Alternatively, one can check unsatisfiability results via *clausal proofs* [10], also known as *reverse unit propagation* (RUP) [28]. For example, Beame et. al showed that learned clauses can be checked using RUP [4]. Clausal proofs are easy to emit and much smaller than resolution proofs. Also, emitting clausal proofs hardly increases the memory consumption of solvers. Consequently, this alternative is a more viable option to validate results of parallel SAT solvers. However, clausal proof are costly to check. In order to validate clausal proofs reasonably efficiently, proofs must contain deletion information, resulting in so-called DRUP proofs [15]. Some techniques cannot be checked with resolution and RUP. Such techniques can be checked using *resolution asymmetric tautology checks* [21, 29].

In this paper, we propose a method to produce DRUP proofs from parallel SAT solvers with clause sharing. Clause sharing makes it complicated to add deletion information to clausal proofs: a clause should only be marked as deleted when it is removed from all solver incarnations. Our ethod uses a counting mechanism to track how many solver incarnations use a certain clause. As soon as a counter is reduced to zero, then the corresponding clause is added as deletion information to the proof. We implemented our method in the portfolio solver `Priss` and evaluate it on application benchmarks from SAT Competition 2013.

The paper is structured as follows: in Section 2 we give the notation that is used in the paper and present the proof format DRUP. Next, we present how to generate DRUP proofs for portfolio solvers in Section 3, and prove that the presented approach is sound in Section 4. Afterwards, we analyze the presented approaches empirically in Section 5 and present a conclusion in Section 6.

# 2 Preliminaries

## 2.1 The Satisfiability Problem

We assume a fixed infinite set $\mathcal{V}$ of boolean *variables*. A *literal* is a variable $x$ (*positive literal*) or a negated variable $\overline{v}$ (*negative literal*). The *complement* $\overline{x}$ of a positive (negative, resp.) literal $x$ is the negative (positive, resp.) literal with the same variable as $x$. For a set of literals $S$, the complement of $S$, denoted with $\overline{S}$ is defined as $\overline{S} = \{\overline{x} \mid x \in S\}$. In SAT, we deal with finite clause sets, called *formulas*. Each clause $C$ is a finite set of literals. We write a clause $\{x_1, \ldots, x_n\}$ also as disjunction $(x_1 \vee \ldots \vee x_n)$ and a formula $\{C_1, \ldots, C_n\}$ as a conjunction $(C_1 \wedge \ldots \wedge C_n)$. The empty clause is denoted with $\bot$.

Asymmetric literal addition [21], $ALA(F, C)$, is the unique clause resulting from repeating the following until fixpoint: if $x_1, \ldots, x_n \in C$, and there is a clause $(x_1 \vee \ldots \vee x_n \vee x) \in F \setminus \{C\}$ for some literal $x$, let $C := C \cup \{\overline{x}\}$. We consider the redundancy criteria *asymmetric tautology* (AT) [21]. A *clause $C$ has AT* if $ALA(C, F)$ is a tautology.

A *DRUP derivation* of a formula $F$ is a finite sequence of proof instructions $(\alpha_i \mid 0 \leq i \leq n)$, where a proof instruction $\alpha$ is either of the form $C^{\mathsf{del}}$ or $C^{\mathsf{rup}}$. The *empty* derivation is denoted with $\varepsilon$. A *RUP* derivation is a DRUP derivation in which no $C^{\mathsf{del}}$ occurs. For a proof derivation $(\alpha_i \mid 1 \leq i \leq n)$ in the formula $F$, we assign a *view* of the formula $F$ and the DRUP derivation $\alpha$, in symbols, $\mathsf{view}(\alpha, F)$, as follows $\mathsf{view}(\varepsilon, F) = F$, $\mathsf{view}(\alpha\, C^{\mathsf{rup}}, F) = \mathsf{view}(\alpha, F) \wedge C$, $\mathsf{view}(\alpha\, C^{\mathsf{del}}, F) = \mathsf{view}(\alpha, F) \setminus \{C\}$. A proof derivation $(\alpha_i \mid 1 \leq i \leq n)$ in the formula $F$ is *correct* if and only if for all $1 \leq i \leq n$ with $\alpha_i = C^{\mathsf{rup}}$ it holds that the clause $C$ has AT in $\mathsf{view}(\alpha_1 \ldots \alpha_{i-1}, F)$. A *refutation for the formula $F$* is a proof derivation $\alpha$ in $F$ such that we find that $\bot \in \mathsf{view}(\alpha, F)$.

### 2.2 Parallel SAT Solving

*Parallelization* is a promising approach to solve *hard instances*. We identify two approaches in parallel SAT solving (see [16, 25, 26] for an overview):

The *Parallel Portfolio* approach [14] made the breakthrough in 2008, and exploits different search strategies by running multiple SAT solvers on the same input formula. This means that these SAT solvers are competing in answering the satisfiability question. This solving approach is the most widely used approach, due to its simplicity, as well as due to its robustness.

The *Search Space Decomposition* approach [30] was introduced in 1996. The *search space*, i.e. the set of interpretations, is decomposed into different spaces by *guiding paths*, that are then explored in parallel by modern sequential SAT solvers. This means that the solvers are competing in finding a model of the formula, and cooperating in proving its unsatisfiability. The solvers `PCASSO` [19] and `Treengeling` [5] have shown that the decomposition approach is competitive with their recent impressive performance, showing that the approach can compete with state-of-the art parallel SAT solvers.

For both the portfolio and the decomposition approach, *clause sharing* has been discussed: learned clauses from one solver incarnation are distributed to other incarnations. For plain portfolio solvers without inprocessing clause sharing is straightforward, because all learned clauses are entailed by all formulas, many clause sharing schemes and heuristics have been analyzed [1, 11, 13]. As soon as inprocessing is used, shared clauses have to be selected carefully to ensure the soundness of the portfolio solver [24]. For decomposition solvers clause sharing is even more complicated. Nevertheless, clause sharing is also possible for this solving approach [17, 22].

In the following, we consider the parallel portfolio approach.

## 3 DRUP Derivations in Parallel Portfolios

This paper investigates the question how *compact* DRUP proofs can be *easily* constructed in the parallel portfolio setting. In the parallel portfolio setting, the input formula $F_0$ is spread among the solver incarnations. Each solver incarnation then tries to find a model of the input formula or to prove the unsatisfiability of the input formula. Additionally, the solvers can share learned clauses.

### 3.1 No Clause Sharing

If no clause sharing is applied, we can easily construct DRUP proofs: each solver incarnation $Solver_i$ constructs a DRUP proof $\alpha_i$ as in the case for sequential SAT solvers. When a solver incarnation $Solver_i$ terminates with the answer $\mathsf{UNSAT}$, it returns DRUP proof $\alpha_i$. Consequently, the derivation $\alpha_i$ is a DRUP refutation for the input formula, since the solver incarnations do not interact.

### 3.2 RUP Proofs with Clause Sharing

We now consider RUP proofs for the parallel portfolio setting in which the solver incarnations cooperate by *clause sharing*. In this case, we propose to merge the RUP proofs that the solver incarnations produce: we have a single RUP derivation $\alpha$, and whenever a solver incarnation learns a clause $C$, we add $C^{\mathsf{rup}}$ to the derivation $\alpha$ in the correct chronological order. If a solver incarnation $Solver_i$ imports a clause $C$ from another solver, nothing is added to the derivation $\alpha$, because this clause is already added by another solver.

This procedure is sufficient and sound. Any new clause that is generated by the clause learning procedure is added immediately to the derivation. By clause sharing, only duplicate clauses are introduced, which are already present in the proof. Since no deletion information is added to the proof, we do not need to add duplicates. However, deletion instructions can significantly reduce the costs to validate unsatisfiability proofs [15].

## 3.3 DRUP Proofs with Clause Sharing

The combination of deletion information and clause sharing is problematic. Suppose a clause $C$ is learned in $Solver_1$, which is shared with $Solver_2$. Afterwards $Solver_1$ eliminates $C$ from its working formula and the proof. However, $C$ is still present for $Solver_2$. Now, $Solver_2$ can learn a clause $D$, which is not necessarily an asymmetric tautology, i.e. the RUP check for $D$ may depend on $C$. Consequently, there are learned clauses that can no longer be validated using a RUP check.

**Example 1.** *Consider the formula* $F = (x \vee y \vee z) \wedge (\overline{z} \vee y) \wedge (x \vee \overline{y} \vee z) \wedge (\overline{y} \vee \overline{z})$. *The resolvent of the clauses* $(x \vee y \vee z)$ *and* $(\overline{z} \vee y)$ *is* $C = (x \vee y)$, *and the resolvent of the clauses* $(x \vee \overline{y} \vee z)$ *and* $(\overline{y} \vee \overline{z})$ *is* $D = (x \vee \overline{y})$. *Finally, the resolvent of the clauses* $C$ *and* $D$ *is the unit clause* $(x)$. *Suppose that a solver incarnation learns the clause* $C$, *and then* $D$. *The proof is then* $C^{\mathsf{rup}}\ D^{\mathsf{rup}}$. *After sharing* $D$ *with all other incarnations, suppose the clause* $D$ *is deleted in the proof. Then, the clause* $(x)$ *has no AT in* $\mathsf{view}(C^{\mathsf{rup}}\ D^{\mathsf{rup}}\ D^{\mathsf{del}}, F)$, *but has AT in* $F \wedge C \wedge D$.

We can avoid the above scenario by treating the proof as a *multiset*. This concept allows to have duplicates such that each clause in the proof is *owned* by a specific solver. In this way, the procedure ensures that any clause that is deleted by a solver is owned by this solver, and if the same clause is used by some other solver, then this other solver has its private copy of the clause.

**Adding Redundant Duplicates.** Given a portfolio of $n$ solver incarnations, we first replicate the input formula $F_0$ $n$ times, such that each solver owns each clause. Whenever a solver incarnation learns the clause $C$, we add this clause to the proof. Likewise, when a solver incarnation removes a clause $C$, we remove $C$ from the proof as well. Finally, when an incarnation receives a clause $C$ from some other incarnation, we add the clause $C$ to the proof again, such that both incarnations have a private copy of the clause $C$.

**Tracking the Number of Occurrences.** The above procedure produces correct DRUP proofs. However, the length of the proof is enormously increased compared to the proof without deletion information (see Section 3.2). The increased length comes from the duplicated clauses in the proof. We can avoid adding duplicates by counting the number of occurrences of a clause in the solver incarnations: before we add a clause to the proof, we test whether the clause is already present. If the clause is not present, then we add the clause to the proof. Otherwise, we still want to ensure that each solver can delete its private copy of the clause. Hence, we add a counter to each clause, which represents the number of occurrences of the clause in the current proof. When the clause is added to the proof, the counter is initialized to 1. The counter is incremented whenever a copy of the clause is learned or imported due to sharing. When a clause is deleted, the counter is decremented, and if the counter reaches 0, we remove the clause from the proof. In this case, the clause is not present in any solver. This procedure avoids duplicating clauses in the proof. However, we now need an additional component that tracks the occurrences of clauses in the solver incarnations.

# 4 The DRUP Refutation Construction is Sound

In this section, we formally prove that the proposed techniques for constructing DRUP refutations in the parallel portfolio approach are sound. First, we model parallel portfolio solvers in terms of a state transition system and afterwards show that the constructed DRUP refutations are sound: a state of a sequential solver $Solver_i$ is the formula $F_i$. Then, the state of a parallel portfolio solver with a DRUP proof is a snapshot of its incarnations $Solver_1, \ldots, Solver_n$ together with a DRUP derivation $\alpha$ and counters, i.e. it is a tuple $(F_1, \ldots, F_n, \alpha, \mathsf{cnt})$.

| | |
|---|---|
| UNSAT-rule: | $(F_1, \ldots, F_i, \ldots, F_n, \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{UNSAT}} \mathsf{UNSAT}\ \alpha$<br>iff $\bot \in F_i$ for some $i \in \{1, \ldots, n\}$. |
| LEARN-rule: | $(F_1, \ldots, F_i, \ldots, F_n \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{LEARN}}$<br>$(F_1, \ldots, F_i \wedge C, \ldots, F_n, \mathsf{cnt}', \alpha')$<br>iff there is a linear resolution derivation to $C$ in $F_i$,<br>$\mathsf{cnt}'(C) = \mathsf{cnt}(C) + 1$<br>$\alpha' = \alpha$ if $\mathsf{cnt}(C) \geq 1$, otherwise $\alpha' = \alpha C^{\mathsf{rup}}$. |
| SHARE-rule: | $(F_1, \ldots, F_i, \ldots, F_n, \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{SHARE}}$<br>$(F_1, \ldots, F_i \wedge C, \ldots, F_n, \mathsf{cnt}', \alpha)$<br>iff $C \in F_j$ for some $j \in \{1, \ldots, n\} \setminus \{i\}$, $\mathsf{cnt}'(C) = \mathsf{cnt}(C) + 1$. |
| FORGET-rule: | $(F_1, \ldots, F_i \wedge C, \ldots, F_n, \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{FORGET}}$<br>$(F_1 \ldots, F_i, \ldots, F_n, \mathsf{cnt}, \alpha)$. |
| DELETE-rule: | $(F_1, \ldots, F_i \wedge C, \ldots, F_n, \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{DELETE}}$<br>$(F_1 \ldots, F_i, \ldots, F_n, \mathsf{cnt}', \alpha')$<br>with $\alpha' = \alpha$ if $\mathsf{cnt}(C) > 1$,<br>otherwise $\alpha' = \alpha C^{\mathsf{del}}$ and $\mathsf{cnt}'(C) = \mathsf{cnt}(C) - 1$. |

Figure 1: Transition relations used to characterize clause sharing models by means of portfolio systems with input formula $F_0$ and multiplicity $n$. These definitions apply to all formulas $F_1, \ldots, F_n, F'_1, \ldots, F'_n$, clauses $C$, DRUP derivations $\alpha, \alpha'$ and $i \in \{1, \ldots, n\}$.

We consider the state transition system RUP and DRUP, whose set of states is

$$\{(F_1, \ldots, F_n, \mathsf{cnt}, \alpha)\} \cup \{\mathsf{SAT}, \mathsf{UNSAT}\ \alpha\},$$

where $F_i$ are formulas, $\alpha$ is a DRUP derivation, and cnt are counters. The *initial state for the input formula $F_0$ with multiplicity $n$* is $\mathsf{init}_n(F_0) = (F_0, \ldots, F_0, \mathsf{cnt}, \varepsilon)$, where $\mathsf{cnt}(C) = n$ for every $C \in F_0$ and $\mathsf{cnt}(D) = 0$ for every clause $D \notin F_0$. The single terminal state is $\mathsf{UNSAT}\ \alpha$, and we consider two different transition relations, which differ in the clause deletion rule.

$$\begin{aligned} \rightsquigarrow_{\mathsf{RUP}} &:= \{\rightsquigarrow_{\mathsf{UNSAT}}, \rightsquigarrow_{\mathsf{LEARN}}, \rightsquigarrow_{\mathsf{SHARE}}, \rightsquigarrow_{\mathsf{FORGET}}\} \\ \rightsquigarrow_{\mathsf{DRUP}} &:= \{\rightsquigarrow_{\mathsf{UNSAT}}, \rightsquigarrow_{\mathsf{LEARN}}, \rightsquigarrow_{\mathsf{SHARE}}, \rightsquigarrow_{\mathsf{DELETE}}\}. \end{aligned}$$

The detailed rules of the transition system are presented in Fig. 1.

#### 4.0.1 System RUP

*System* RUP models parallel portfolio solvers with clause sharing where the solver incarnations apply conflict-directed backtracking and learning. The system produces RUP derivations and its transition relation is composed of the UNSAT-, LEARN-, SHARE-, and FORGET-rule: the UNSAT-rule terminates the computation in the final state UNSAT $\alpha$, if some formula $F_i$ contains the empty clause $\bot$. The LEARN-rule adds a clause $C$ to the formula $F_i$, if there is a linear resolution derivation in $F_i$ to the clause $C$ and increments the counter $\mathsf{cnt}(C)$ by one. If $\mathsf{cnt}(C)$ is 1, $C^{\mathsf{rup}}$ is added to the proof derivation $\alpha$. The SHARE-rule models clause sharing and adds a clause $C \in F_j$ to the formula $F_i$ and increments the counter $\mathsf{cnt}(C)$ by one. The FORGET-rule models clause forgetting, i.e. we eliminate the clause $C$ from the solver incarnation $Solver_i$, but it does not modify the RUP derivation $\alpha$ nor the counters. We demonstrate System RUP in the following example:

**Example 2.** *Consider the input formula $F_0 = (x) \wedge (\overline{x} \vee y) \wedge F_0'$, where $F_0'$ is some unsatisfiable formula. Then $\mathsf{init}_2(F_0) = (F_0, F_0, \mathsf{cnt}_0, \varepsilon)$, where $\mathsf{cnt}_0(x) = 2$ and $\mathsf{cnt}_0(\overline{x} \vee y) = 2$. $Solver_2$ then learns the clause $(y)$, i.e. $\mathsf{init}_2(F_0) \rightsquigarrow_{\mathsf{LEARN}} (F_0, F_0 \wedge (y), \mathsf{cnt}_1, (y)^{\mathsf{rup}})$. The counter $\mathsf{cnt}_1$ is obtained from $\mathsf{cnt}_0$ by incrementing $\mathsf{cnt}(y)$ by one. Afterwards, it shares $(y)$ with $Solver_1$: $(F_0 \wedge y, F_0 \wedge (y), \mathsf{cnt}_2, (y)^{\mathsf{rup}})$, where $\mathsf{cnt}_2$ is obtained from $\mathsf{cnt}_1$ by incrementing $\mathsf{cnt}_1(y)$ by one. Then, let $Solver_1$ deletes its clause $(\overline{x} \vee y)$ as it is subsumed by $(y)$, resulting in the state $((x) \wedge (y) \wedge F_0', F_0 \wedge (y), \mathsf{cnt}_2, (y)^{\mathsf{rup}})$.*

#### 4.0.2 System DRUP

*System* DRUP extends System RUP by clause deletion information. The DELETE-rule eliminates a clause from the formula and, if the clause does not occur in another solver, it is also eliminated from the proof. This is the case when counter $\mathsf{cnt}(C)$ becomes 0. We demonstrate System DRUP in the following example.

**Example 3.** *Consider the input formula $F_0 = (x) \wedge (x \vee y) \wedge F_0'$, where the formula $F_0'$ is unsatisfiable not containing $(x \vee y)$. Then $\mathsf{init}_2(F_0) = (F_0, F_0, \mathsf{cnt}_0, \varepsilon)$. First, $Solver_1$ deletes the subsumed clause $(x \vee y)$: $((x) \wedge F_0', F_0, \mathsf{cnt}_1, \varepsilon)$, where $\mathsf{cnt}_1(x \vee y) = 1$. Afterwards $Solver_2$ does the same: $((x) \wedge F_0', (x) \wedge F_0', \mathsf{cnt}_2, (x \vee y)^{\mathsf{del}})$, where $\mathsf{cnt}_2(x \vee y) = 0$. Consequently, the clause $(x \vee y)$ is not present any more in the proof.*

In the next subsection we investigate the question whether System RUP and DRUP produce a correct DRUP refutations for unsatisfiable input formulas. For a state transition relation $\rightsquigarrow$, the symbol $\overset{*}{\rightsquigarrow}$ denotes the reflexive and transitive closure of $\rightsquigarrow$. Let $x \overset{0}{\rightsquigarrow} x$ for all states $x$, and $x \overset{n}{\rightsquigarrow} z$ for all natural numbers $n \in \mathbb{N}$ if and only if there exists a state $y$ such that $x \overset{n-1}{\rightsquigarrow} y \rightsquigarrow z$. Formally, we define a state transition system to be *sound* iff for all unsatisfiable formulas $F_0$ we have that $\mathsf{init}(F_0) \overset{*}{\rightsquigarrow} \mathsf{UNSAT}\ \alpha$ implies that $\alpha$ is a DRUP refutation for $F_0$. In particular, this means that the input formula $F_0$ is then unsatisfiable.

## 4.1 System RUP

We begin our investigation with invariant properties in System RUP in the proposition below: *1.* states that the constructed proof derivation is correct, i.e. if the proof derivation $\alpha$ is of the form $\beta C^{\mathsf{rup}} \beta'$, then the clause $C$ has AT in $\mathsf{view}(\beta, F)$. *2.* states that the counter for the clause $C$ is always greater than one, if the clause $C$ occurs in some formula $F_i$. *3.* states that for every clause having a counter $\mathsf{cnt}(C) \geq 1$, the clause $C$ is present in the proof, i.e. $C \in \mathsf{view}(\alpha, F_0)$.

**Proposition 1** (Invariants ***RUP***). *Let $n \geq 1$, $F_0, F_1, \ldots, F_n$ be formulas, and let $m \geq 0$. If $\mathsf{init}_n(F_0) \overset{m}{\rightsquigarrow}_{\mathsf{RUP}} (F_1, \ldots, F_n, \mathsf{cnt}, \alpha)$, then the following holds:*

1. *The derivation $\alpha$ is correct for $F_0$,*
2. *$\mathsf{cnt}(C) \geq 1$ for every clause $C \in F_i$ and $i \in \{1, \ldots, n\}$,*
3. *$C \in \mathsf{view}(\alpha, F_0)$ for every clause $C$ with $\mathsf{cnt}(C) \geq 1$, and*
4. *$\alpha$ is a RUP derivation.*

*Proof.* We show the claims by induction on the number $m$ of transition steps. For the base case $m = 0$, the claims *1.* is easy to see since $\alpha$ is the empty derivation, and *2.* and *3.* follows immediately form the definition of the initial state for the input formula $F_0$. For the induction step, assume that the claim holds for the state $(F_1, \ldots, F_n, \mathsf{cnt}, \alpha)$ and that $(F_1, \ldots, F_n, \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{R}} (F_1, \ldots, F_{i-1}, F_i', F_{i+1}, \ldots, F_n, \mathsf{cnt}', \alpha')$ for some rule $\mathsf{R}$ in System $\mathsf{RUP}$. Note that invariant *4.* follows easily from the definitions of the rules. For the remaining claims, we distinguish between the applied rule $\mathsf{R} \subseteq \{\mathsf{LEARN}, \mathsf{SHARE}, \mathsf{FORGET}\}$:

- $\mathsf{LEARN}$-rule: Suppose that $\mathsf{cnt}(C) > 0$. Then *1.* follows immediately by induction, because $\alpha = \alpha'$. Otherwise, we know that the clause $C$ has AT in the formula $F_i$. By *2.* and *3.*, we conclude that the clause $C$ has AT in the formula $\mathsf{view}(\alpha, F_0)$. Hence, the derivation $\alpha C^{\mathsf{rup}}$ is correct for $F_0$. Invariant *2.* follows immediately since we increase $\mathsf{cnt}(C)$ by one. Invariant *3.* holds immediately in the case that $\mathsf{cnt}(C) \geq 1$. If $\mathsf{cnt}(C) = 0$, then $\mathsf{cnt}'(C) = 1$ and $C \in \mathsf{view}(\alpha', F_0)$.
- $\mathsf{SHARE}$-rule: Invariant *1.* follows immediately by induction. Invariant *2.* follows immediately since $\mathsf{cnt}(C)$ is increased by one. Invariant *3.* follows immediately.
- $\mathsf{FORGET}$-rule: Invariant *1.* follows immediately by induction, because $\alpha = \alpha'$. Invariant *2.* follows easily as we have to consider less clauses. Invariant *3.* follows immediately by induction, because $\alpha = \alpha'$ and $\mathsf{cnt} = \mathsf{cnt}'$. □

We can now proceed to the statement that System $\mathsf{RUP}$ produces correct RUP refutations.

**Theorem 1.** *System ***RUP*** is sound.*

*Proof.* Suppose that $\mathsf{init}_n(F_0) \overset{*}{\rightsquigarrow}_{\mathsf{RUP}} (F_1, \ldots, F_n, \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{UNSAT}} \mathsf{UNSAT}\ \alpha$. By the definition of the $\mathsf{UNSAT}$-rule we know that $\bot \in F_i$ for some $i \in \{1, \ldots, n\}$. From Prop. 1 *2.* and *3.* we know that $\bot \in \mathsf{view}(\alpha, F_0)$. By Prop. 1 *1.* and *4.* we know that the RUP derivation $\alpha$ is correct for the input formula $F_0$. Consequently, $\alpha$ is a RUP refutation for input formula $F_0$. □

Note that all solver incarnations in System $\mathsf{RUP}$ are allowed to eliminate clauses in their formula, but not to add deletion information the proof derivation. In fact, System $\mathsf{RUP}$ distinguishes only whether a clause $C$ has a counter $\mathsf{cnt}(C) = 0$ or $\mathsf{cnt}(C) > 0$. Therefore, an efficient counter implementation can be done by clause hashing.

## 4.2 System DRUP

The proposition below states the invariants of System $\mathsf{DRUP}$, which are slightly different than for the System $\mathsf{RUP}$: *1.* states that the derivation $\alpha$ is correct for the input formula $F_0$. *2.* states that the number of occurrences of the clause $C$ in the formula $F_1, \ldots, F_n$ is *exactly equal* to $\mathsf{cnt}(C)$. *3.* states that a clause $C$ is in the formula $\mathsf{view}(\alpha, F_0)$ if and only if its counter is greater or equal than 1.

**Proposition 2** (Invariants ***DRUP***). *Let $n \geq 1$, let $F_0, F_1, \ldots, F_n$ be formulas, and let $m \geq 0$. If $\mathsf{init}_n(F_0) \overset{m}{\rightsquigarrow}_{\mathsf{DRUP}} (F_1, \ldots, F_n, \mathsf{cnt}, \alpha)$, then the following hold:*

1. *the derivation $\alpha$ is correct for $F_0$,*
2. *the number of occurrences of the clause $C$ in $F_1, \ldots, F_n$ is $\mathsf{cnt}(C)$,*
3. *$C \in \mathsf{view}(\alpha, F_0)$ if and only if $\mathsf{cnt}(C) \geq 1$.*

*Proof.* We show the claims by induction on the number $m$ of transition steps. For the base case $m = 0$, the claims *1.* is easy to see since $\alpha$ is the empty derivation, and *2.* and *3.* follows immediately form the definition of the initial state for the input formula $F_0$. For the induction step, assume that the claim holds for the state $(F_1, \ldots, F_n, \mathsf{cnt}, \alpha)$ and that $(F_1, \ldots, F_n, \mathsf{cnt}, \alpha) \rightsquigarrow_{\mathsf{R}} (F_1, \ldots, F_{i-1}, F_i', F_{i+1}, \ldots, F_n, \mathsf{cnt}', \alpha')$ for some rule $\mathsf{R}$ in System 1. We distinguish between the applied rule $\mathsf{R} \subseteq \{\mathsf{LEARN}, \mathsf{SHARE}, \mathsf{DELETE}\}$:

- LEARN-rule: Suppose that $\mathsf{cnt}(C) > 0$. Then *1.* follows immediately by induction. Otherwise, we know that $C$ is RUP-inferable in $F_i$. By *3.*, we then know that $C$ is RUP-inferable in $\mathsf{view}(\alpha, F_0)$- Hence, the derivation $\alpha C^{\mathsf{rup}}$ is correct for $F_0$. Invariant *2.* follows immediately since we increase $\mathsf{cnt}(C)$ by one. Invariant *3.* holds immediately in the case that $\mathsf{cnt}(C) \geq 1$. If $\mathsf{cnt}(C) = 0$, then $\mathsf{cnt}'(C) = 1$ and $C \in \mathsf{view}(\alpha', F_0)$.
- SHARE-rule: Invariant *1.* follows immediately by induction. Invariant *2.* follows immediately since $\mathsf{cnt}(C)$ is increased by one. Invariant *3.* follows immediately.
- DELETE-rule: Invariant *1.* is clear from the definition of correctness of DRUP derivations. Invariant *2.* follows immediately since $\mathsf{cnt}(C)$ is decreased by one. Invariant *3.* follows immediately since we delete the clause $C$ from the proof, if $\mathsf{cnt}(C) = 1$.

□

We can now show the main theorem, stating that the constructed DRUP proofs are refutations for the input formula.

**Theorem 2.** *System ***DRUP*** is sound.*

*Proof.* The proof is analogous to the proof of Theorem 1, but uses Prop. 2 instead of Prop. 1. □

# 5 Empirical Investigation

For the empirical evaluation we use the SAT solver `Riss` [23], and turned it into a simple parallel portfolio solver `Priss` with clause sharing.[1] Any learned clause that has at most 10 literals, and whose LBD [2] is less than 6 is shared with all solver incarnations. Any shared clause is received by all solver incarnations, if the clause is not yet satisfied. Clauses are received whenever the search of a solver is at level 0. The pool for shared clauses is a ring buffer with a capacity of 16000 clauses. Old clauses are overwritten in a FIFO manner.

## 5.1 The Proof Master

For the portfolio solver the *proof master*, which is a shared object, is added to the portfolio solver. The proof master takes care of generating the DRUP proof for the solver, and is the only object in the system that has access to the proof file. This proof master can apply the

[1] The implementation is available at `http://tools.computational-logic.org/content/riss/riss427.tar.gz`.

counting scheme that has been presented above, or it simply prints each request directly to the proof file. Both cases have its benefits: when the counting scheme is applied, then no duplicate clauses will be placed on the proof. On the other hand, the whole proof has to be kept in main memory. The current implementation of this scheme does not support garbage collection, and therefore the required main memory for longer proofs is very high. However, for shorter proofs this implementation yields as a proof of concept. Consequently, when no duplicate clauses are not managed, then the proof becomes longer, but the required memory of the proof master is much lower.

Tracing duplicate clauses is done with a hash table: we represent literals in the memory as natural numbers: the representation $l$ of the positive literal $x$ is $l = 2\,x$, and for negative literals $x$ it is $l = 2\,x + 1$. Then, the clause $C$ is stored in a hash table using the hash function below:

```
unsigned int getHash (int* clause) {
  unsigned int sum = 0, prod = 1, xor = 0;
  while (*clause) {
    prod *= *clause; sum += *clause;
    xor ^= *clause; clause++; }
  return (1023 * sum + prod ^ (31 * xor)) % HASHMAX; }
```

When a new clause $D$ is added to the proof, or the clause $D$ is removed, the proof master checks the hash table for a clause with the same hash, and next identifies the right clause by comparing all the literals in the two clauses $C$ and $D$. When $D$ is added and no matching clause is present, then $D$ is added to the hash table, and the counter is initialized to 1. Additionally, we add the clause $D$ to the proof file. Otherwise, the counter of the matching clause is increased. When $D$ is removed, the counter of the matching clause $C$ is decreased. If this counter becomes zero, we remove the clause $C$ from the hash table, and the corresponding deletion line is added to the proof file. For *testing purposes*, we require that every deleted clause must be present in the hash table.

## 5.2 Generating the Proof

Let $n$ be the number of used threads in `Priss`. The formula $F$ is first read by the first solver incarnation and simplified resulting in the formula $F'$. All simplification techniques that can be expressed in DRUP proofs, as for example variable elimination [9] or blocked clause elimination [20], could be performed by this solver. Then, the $n-1$ remaining solver incarnations are created, and initialized with the simplified formula $F'$. In this paper the preprocessor is turned off, so that the two formulas $F$ and $F'$ match. As discussed above, each clause in the formula $F'$ has to be present in the generated proof $n-1$ times, so that each solver incarnation is allowed to delete such a clause again. Therefore, for each initialized solver incarnation, the formula $F'$ is added to the proof once more.

When the proof master uses counting, then for each clause $C \in F'$ the counter is set to the requested value. For the added checks, the counter for each clause $C$ is set to $n$, so that the deletion of the $n$ solver incarnations can be traced. This way, the generated proof contains each clause $C \in F'$ one time more than actually necessary. Due to the counting no duplicates are added to the proof. The only drawback here is that the clause is added to the proof additionally to the clauses in $F$, which are added to the proof by the proof verification tool.

**Handling Shared Clauses.** Shared clauses are added in the following way: whenever a shared clause $C$ is added to the clause pool, then this clause is also sent to the proof master.

Consequently, any solver incarnation that receives this clause, adds the clause $C$ to the proof once more. If an incarnation wishes to delete this clause again, this request is submitted to the proof master. Finally, when the clause $C$ is removed from the ring buffer in the clause pool, then this clause $C$ is removed from the proof by the proof master as well. This way, the size of the portfolio solver without counting clauses might become much larger than for a sequential solver, because for $n$ solver incarnations any learned clause that becomes a shared clause might be added $n + 1$ times to the proof. Therefore, very good clause sharing filters are not only necessary for reducing the communication overhead for portfolio solvers, but also to keep the size of the generated proof reasonable.

## 5.3 Experimental Evaluation

We use the tool `drat-trim` [29] to verify the generated DRUP proofs. `Priss` uses four threads, which differ only in the way they schedule their restarts, and how to increase the activity of variables. The evaluation of the techniques is performed on all unsatisfiable application instances of the SAT Competition 2013. The wall clock limit for solving the formula is set to 1800 seconds and the wall clock limit for verifying the proof is 7200 seconds. A memory limit of 18 GB and a maximum file size of 16 GB is applied. The used cluster uses Intel Xeon E5-2670 CPUs with 8 cores and 20 MB level 3 cache that is shared by all cores. Due to the cluster architecture, the experiment uses a pair of CPUs exclusively for one solver. For the empirical evaluation we set up different configurations of the portfolio solver. Seq is the sequential solver of the solver incarnation $S_0$. NoShareNoDRUP runs the portfolio without sharing and without generating proofs. NoShare additionally generates a DRUP proof with using counting. ShareNoDRUP uses sharing, but does not generate a proof. Share also generates a proof with counting. Finally, ShareNoCount generates a proof, but without counting.

The run time and the used memory for the solving the formulas without verification is presented in Fig. 2. The memory consumption for the sequential solver is smallest among the configurations that generate a proof, because only the formula and the learned clauses have to be maintained once. For the parallel configurations memory consumption is much higher. Surprisingly, for the plain portfolio solver, the consumption is the highest, followed by the Share variant. Moreover, the diagram shows that clause sharing improves the performances of parallel SAT solvers. However, when DRUP proofs are generated, the sequential solver can produce more verifiable proofs in the resource limit. When counting is used, then the best parallel overall procedure is obtained. Not using the counting mechanism slightly decreases the

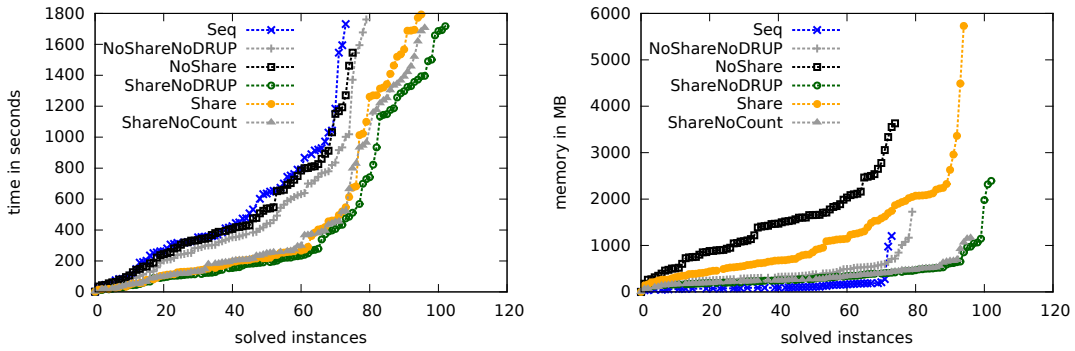


Figure 2: Run time and memory consumption.

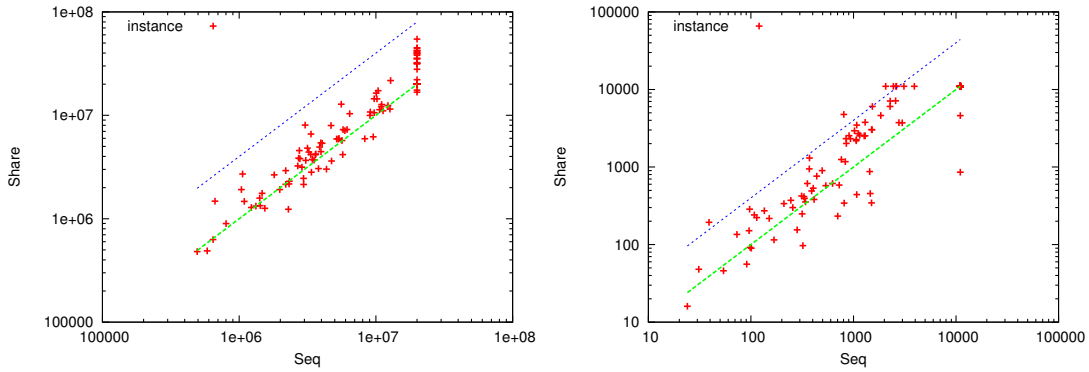


Figure 3: Proof length (left) and proof verification time (right) for the sequential solver and the counting clause sharing portfolio solver.

overall run time of the tool chain. Consider the 96 instances that were solved by the SHARE and the SHARENOCOUNT configuration. Then, 32 generated proofs without the counting scheme could not be verified within the time bound. The counting scheme improves the number of verified proofs within the resource limit to 26, i.e. with the counting scheme we can verify 6 more proofs. Moreover, the counting scheme increases the speed of proof verification: The average time for the successfully verified proofs is 2012 seconds for the SHARENOCOUNT configuration and 1550 seconds for the SHARE configuration.

## 5.4 Proof Size

The size of the generated proof is considered as one of the most important properties. Fig. 3 shows this property, and furthermore gives the verification time for the configurations SHARE and SEQ. The proof length of the parallel solver is bounded by the length of the sequential proof and the number of threads as factor. The tendency is that the longer the sequential proof is, the closer is the parallel proof.

The time to verify the proof does not share these boundaries. The reason for this effect is the way the proof verification works: instead of processing each clause from the beginning of the proof to the end, `drat-trim` starts at the end of the proof and only checks clauses that are relevant to prove unsatisfiability. Hence, both in the sequential as well as in the parallel proof, there might be short-cuts in the one proof, which are not present in the other proof, also resulting in parallel proofs that can verified faster.

## 5.5 Discussion

Given the implementation with the metrics for clause sharing and handling the proof, under the specified resources the sequential version of the solver can produce more verified results than the used portfolio. Still, from the given results an outlook to future solvers with improved parameters and an improved implementation is interesting. First, the ratio between the solving time and the verification time is shifted to the proof verification tool, since the parallel SAT solver solves its formula faster. For the verified output, the current difference is shown in Fig. 4. Sequential proofs can be verified in a time of up to four times the solving times. This ratio does not hold for the parallel solver, and within the given resource limits, not all generated proofs

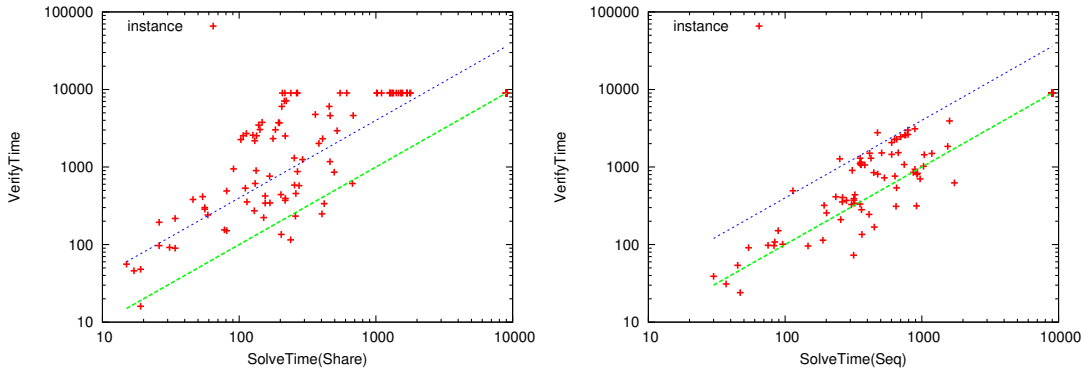


Figure 4: Solving time compared to verification time for the clause sharing portfolio (left) and the sequential solver (right).

can be verified. Hence, a parallel proof verification tool would be the next step to set this ratio into the right shape again.

Next, the overhead of logging the DRUP proof is interesting, since for the parallel solver the proof needs to be locked. In our implementation we used a lock to maintain the hash table. With the same lock we also granted access to the proof file. Our experiments showed, that the overhead of this simple locking scheme is already very low. An improved version of the proof master could separate between locally learned clauses, that need to be present only for the proof of a single solver incarnation. Only when this incarnation shared the next clause with the whole solver, this local proof needs to be made available to the global proof. This way, lock accesses can be avoided. However, on the other hand, more memory becomes necessary for the local proof of each solver incarnation. An analysis into this direction is considered as future work.

The presented results show that portfolio solvers can be used to generate proofs, and that the generated proofs do not contain much redundancy. However, currently the required resources enforce a limitation. As soon as ordinary machines offer sufficient main memory, and there is enough space for the proof files, using a portfolio solver to produce verified unsatisfiability answers for formulas should be considered. This paper presented how such a portfolio can be built, such that the generated proof is valid.

# 6 Conclusion

In this paper we showed how the unsatisfiability answer of modern portfolio solvers can be verified. By exploiting a counting mechanism, the generated proof is compact and can be verified in reasonable time. Still, this development is only a first step and opens the door for many improvements and further investigations.

First, the results are only presented for parallel portfolio SAT solvers without inprocessing. Consequently, the next step is incorporating inprocessing into the model, as well as extending the proof generation to other parallel solving approaches, for example to instance-decomposition-based solvers such as `PCASSO` [19] and `Treengeling` [6].

Furthermore, not all simplification techniques can be described with DRUP. Hence, we plan to analyze how we can model simplification techniques with the extended format DRAT. Then,

we would like to incorporate these simplification techniques into the portfolio solver and still generate valid DRAT proofs for such a parallel system.

On the verification side our run time analysis showed that with the parallelization of the SAT solver, most of the run time is spend on proof verification. Hence, a natural next step is to parallelize the proof verification, to achieve a ratio between solving the formula and checking the unsatisfiability proof for the parallel scenario, which is equal to the sequential scenario.

**Acknowledgments** The author thanks the ZIH of TU Dresden for providing the computational resources to produce the experimental data for the empirical evaluation.

# References

[1] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT 2012*, volume 7317 of *LNCS*, pages 200–213, Heidelberg, 2012. Springer.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Boutilier [8], pages 399–404.

[3] A. Balint, A. Belov, M. J. H. Heule, and M. Järvisalo, editors. *Proceedings of SAT Challenge 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, Helsinki, Finland, 2013.

[4] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22(1):319–351, 2004.

[5] A. Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In Balint et al. [3], pages 51–52.

[6] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Report Series Technical Report 10/1, Johannes Kepler University, Linz, Austria, 2010.

[7] Armin Biere. BooleForce and TraceCheck, 2014.

[8] Craig Boutilier, editor. Pasadena, 2009. Morgan Kaufmann Publishers Inc.

[9] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT 2005*, volume 3569 of *LNCS*, pages 61–75, Heidelberg, 2005. Springer.

[10] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE 2003*, pages 10886–10891, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In *CP 2010*, volume 6308 of *LNCS*, pages 252–265. Springer, 2010.

[12] Youssef Hamadi, Saïd Jabbour, Cédric Piette, and Lakhdar Sais. Deterministic parallel DPLL. *JSAT*, 7(4):127–132, 2011.

[13] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In Boutilier [8], pages 499–504.

[14] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.

[15] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *FMCAD 2013*, pages 181–188. IEEE, 2013.

[16] S. Hölldobler, N. Manthey, V.H. Nguyen, J. Stecklina, and P. Steinke. A short overview on modern parallel SAT-solvers. In *Proceedings of the International Conference on Advanced Computer Science and Information Systems (ICACSIS 2011)*, pages 201–206. IEEE, 2011.

[17] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In J. H.-M. Lee, editor, *Principles and Practice of Constraint Programming,*, volume 6876 of *LNCS*, pages 385–399, 2011.

[18] Antti Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving SAT in grids. In Armin Biere and Carla P. Gomes, editors, *SAT 2006*, volume 4121 of *LNCS*, pages 430–435. Springer, 2006.

[19] Ahmed Irfan, Davide Lanti, and Norbert Manthey. PCASSO — a Parallel CooperAtive Sat SOlver. In Balint et al. [3], pages 64–65.

[20] Matti Järvisalo, Armin Biere, and Marijn J. H. Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 129–144, Heidelberg, 2010. Springer.

[21] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 355–370, Heidelberg, 2012. Springer.

[22] Davide Lanti and Norbert Manthey. Sharing information in parallel search with search space partitioning. In Giuseppe Nicosia and Panos Pardalos, editors, *Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION 7)*, volume 7997 of *LNCS*, 2013.

[23] N. Manthey. The SAT solver RISS3G at SC 2013. In Balint et al. [3], pages 72–73.

[24] Norbert Manthey, Tobias Philipp, and Christoph Wernhard. Soundness of inprocessing in clause sharing SAT solvers. In Matti Järvisalo and Allen Van Gelder, editors, *SAT 2013*, volume 7962 of *LNCS*, pages 22–39, Heidelberg, 2013. Springer.

[25] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[26] Daniel Singer. *Parallel Resolution of the Satisfiability Problem: A Survey*, chapter 5, pages 123–148. Wiley Interscience, 2006.

[27] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Symposium on Artificial Intelligence and Mathematics 2002*, 2002.

[28] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*. Springer, 2008.

[29] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim efficient checking and trimming using expressive clausal proofs. In *SAT 2014*, 2014, accepted.

[30] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21, 1996.