


Proofs for Incremental SAT with Inprocessing

Benjamin Kiesl-Reiter 

Amazon Web Services
Munich, Germany
Email: benkiesl@amazon.com

Michael W. Whalen 

Amazon Web Services and University of Minnesota
Minneapolis, MN, United States
Email: mww@amazon.com

Abstract—Incremental SAT solvers are automated-reasoning tools that efficiently solve sequences of related logic problems, making them a go-to tool for inherently incremental applications such as model checking, planning, or test generation. Recent advances in incremental solving using inprocessing have led to substantial performance improvements, but it has remained unclear how the resulting solvers could produce verifiable proofs of unsatisfiability. Here we provide a simple approach that enables inprocessing solvers to produce proofs for incremental results. The approach extends the standard DRAT format with clause restoration steps. These are later removed during post-processing, yielding a standard DRAT proof. Our empirical evaluation shows that our approach is sound and efficient. Proofs can be generated much faster compared to re-solving a problem non-incrementally, but the resulting proofs tend to be larger. Nevertheless, even when taking proof checking into account, our approach is still slightly faster on average. In addition, our technique has the advantage of guaranteeing proof production whereas the non-incremental approach can time out on hard problems.

Index Terms—Automated reasoning, SAT solving, incremental solving, proof.

I. INTRODUCTION

Incremental SAT solving is a key technique for the formal analysis of software and hardware. Instead of solving each input problem independently, an incremental SAT solver retains state between solver calls, allowing it to rely on previously learned information when faced with new problems. This reuse of learned information can dramatically boost performance for applications that produce sequences of closely related SAT problems, such as automated planning [1], lazy SMT solving [2], test-case generation [3], [4], and bounded model checking [5]–[7].

In practice, a user of an incremental SAT solver initializes the solver and provides it with an initial input formula. After solving the formula, the user can then extend the formula before sending another solve request to the solver. The resulting extend-and-solve loop can be repeated arbitrarily many times until eventually the user decides to release the solver. While this incremental feature makes solvers efficient and easy-to-use, the formula modifications performed between solver calls add an additional layer of complexity that can render several commonly-used reasoning techniques unsound. In particular, many preprocessing and inprocessing techniques that are crucial

to the performance of non-incremental solvers cannot be used straightforwardly in an incremental context. This is a pity since inprocessing-based solvers have won the top spots in the yearly SAT competitions since 2020 [8].

To harvest at least some of the performance gains offered by inprocessing techniques, several ways of using them in a restricted way have been suggested in the literature [6], [7], [9]. In 2019, a breakthrough was made by Fazekas, Biere, and Scholl [10], who introduced a calculus that allows the unrestricted use of inprocessing techniques *during* incremental solver runs, given that additional reasoning steps—so-called *clause restorations*—are performed *ahead* of the runs. The implementation of their approach on top of the award-winning SAT solver CaDiCaL [11] has led to impressive performance gains, showing that inprocessing and incremental solving can coexist in practice.

One key capability of non-incremental solving, however, has still failed to enter the picture—*proof*. While virtually all modern non-incremental solvers can produce independently verifiable proofs of unsatisfiability (usually in the DRAT format [12] required by SAT competitions), it has remained unclear how an incremental solver with unrestricted inprocessing could produce such proofs. In this paper, we address this issue by presenting a surprisingly simple approach for extracting a verifiable DRAT proof from an incremental solver relying on the calculus by Fazekas et al. [11]

Our approach requires a solver to augment its proof trace with additional proof steps whenever it performs clause restorations. Once the solver finishes, we transform the augmented proof trace into a verifiable DRAT proof. The key idea is to remove deletions from the proof corresponding to clauses that need to be restored. To demonstrate the feasibility of our approach in practice, we modified the solver CaDiCaL and implemented our proof-transformation algorithm as a separate tool. The resulting version of CaDiCaL is thus the first incremental SAT solver that combines unrestricted inprocessing with proof production.

Despite the theoretical simplicity of our approach, its actual implementation required us to make several careful changes to the solver, which we explain in detail. This should provide solver developers with sufficient background to implement our approach on top of other solvers. We performed an evaluation with 300 benchmarks from

the 2017 Hardware Model Checking competition [13], demonstrating that the overhead of our approach is small and that all resulting proofs can be verified with the existing proof checker DRAT-trim [14].

The main contributions of this paper are as follows:

- An approach for extracting verifiable DRAT proofs from an incremental SAT solver that performs unrestricted inprocessing.
- Implementation on top of the SAT solver CaDiCaL.
- Empirical evaluation on a comprehensive benchmark set, demonstrating the feasibility of our approach.

The rest of this paper is structured as follows. In Section II, we present the background required to understand the rest of the paper. In Section III, we describe the general problem of producing proofs for incremental SAT solving with inprocessing. In Section IV, we present our algorithm for generating proofs and establish its soundness before discussing implementation details in Section V. Finally, we present the results of our empirical evaluation in Section VI and conclude with a summary and an outlook for future work in Section VII.

II. BACKGROUND AND RELATED WORK

We first cover basics of SAT solving before giving a high-level overview of the relationship between incremental SAT solving and inprocessing.

A. SAT Solving Basics

The Boolean satisfiability problem (SAT) asks whether a formula of propositional logic can be satisfied by some assignment of truth values (*true* and *false*) to its variables. An overview can be found in [15].

As is common in practical SAT solving, we concern ourselves with formulas in *conjunctive normal form* (CNF). Formulas are built from *literals*, which are either variables (x) or their negations (\bar{x}). These are called *positive literals* and *negative literals* respectively. We write $var(l)$ to refer to the variable of the literal l ($var(x) = x$ and $var(\bar{x}) = x$). The *complement* \bar{l} of a literal l is defined as $\bar{l} = \bar{x}$ if $l = x$ and as $\bar{l} = x$ if $l = \bar{x}$. A *clause* is a finite disjunction of literals of the form $(l_1 \vee l_2 \vee \dots \vee l_n)$. A *formula* is a finite conjunction of clauses of the form $C_1 \wedge C_2 \wedge \dots \wedge C_m$. For example, $(\bar{x} \vee y \vee z) \wedge (y \vee \bar{z}) \wedge (x)$ is a formula with three clauses, where the last clause is called a *unit clause* because it contains only one literal. Formulas can be viewed as sets of clauses, which can be viewed as sets of literals.

A *truth assignment* (or *assignment* for short) is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). A literal is *satisfied* by an assignment α if l is positive and $\alpha(var(l)) = 1$ or if l is negative and $\alpha(var(l)) = 0$. A literal l is *falsified* by α if \bar{l} is satisfied by α . An assignment α can be viewed as the set $\{l \mid l \text{ is satisfied by } \alpha\}$ of literals. A clause is satisfied by an assignment if the assignment satisfies at least one of its literals. A formula is satisfied by an assignment if the assignment satisfies all of its clauses.

DIMACS					DRAT				
p	cnf	4	8						
1	-2		0			-3	0		
	2		-4	0	1	2	0		
1	2		4	0		-1	0		
-1	-3		0			-3	0		
1	-3		0			1	0		
-1	3		0		d	1	2	0	
1	3	-4	0				0		
1	3	4	0						

Fig. 1. DIMACS formula and corresponding proof in DRAT format.

A formula is *satisfiable* if there exists an assignment that satisfies it, otherwise it is *unsatisfiable*. Two formulas are *logically equivalent* if they are satisfied by the same assignments; they are *equisatisfiable* if they are either both satisfiable or both unsatisfiable.

Incremental SAT. An incremental SAT problem is a sequence $\langle \Delta_0, A_0 \rangle, \dots, \langle \Delta_n, A_n \rangle$ of pairs, where each Δ_i is a set of clauses and each A_i is a set of literals called *assumptions*. In each solving phase $i \in 0, \dots, n$, the task is to determine satisfiability of the union of the first i sets of clauses under the single set A_i of assumptions, i.e., to determine satisfiability of $\Delta_0 \cup \dots \cup \Delta_i \cup \{(l) \mid l \in A_i\}$.

File Formats and Proofs. In non-incremental SAT, formulas are typically specified in the DIMACS format. DIMACS files feature a header of the form ‘p cnf #variables #clauses’ followed by a list of clauses. Each clause is represented by a list of integers, with a 0 denoting the end of a clause. For example, the clause $(x_1 \vee \bar{x}_2 \vee x_3)$ is represented as ‘1 -2 3 0’. An example formula in DIMACS format is given in Fig. 1.

The current standard format for proofs is DRAT (short for *Deletion Resolution Asymmetric Tautology*) [12]. A DRAT file specifies a sequence of proof statements, which are either clause *additions* or clause *deletions*. Formally, a DRAT proof of a formula F can be seen as a sequence $\langle s_1, C_1 \rangle, \dots, \langle s_m, C_m \rangle$, where each $s_i \in \{a, d\}$ and each C_i is a clause. A proof of F gives rise to an *accumulated formula* as follows (where $F_0 := F$):

$$F_i = \begin{cases} F_{i-1} \cup \{C_i\} & \text{if } s_i = a \\ F_{i-1} \setminus \{C_i\} & \text{if } s_i = d \end{cases}$$

Each added clause C_i must have the *RAT property* [16] with respect to F_{i-1} . RAT (short for *Resolution Asymmetric Tautology*) is a non-monotonic syntactic property that is checkable in polynomial time and that guarantees that the clause addition preserves satisfiability. Further details of RAT are not essential to our paper, we refer the interested reader to [16] for more information.

Deletions can remove arbitrary clauses from the accumulated formula; they clearly preserve satisfiability. A valid DRAT proof of unsatisfiability ends with the addition of the empty clause. Because the empty clause is trivially unsatisfiable (and since each proof step preserves satisfiability) the unsatisfiability of the original formula

F can be concluded. DRAT has a plain-text format and a more compact binary format. An example plain-text DRAT proof is given in Fig. 1 (note that deletions are preceded by a d symbol whereas additions are not preceded by any symbol; a 0 marks the end of a statement).

Example 1. *The DRAT proof on the right of Fig. 1 is a proof of the DIMACS file on the left. It derives all clauses from earlier clauses via a RAT derivation step. For example, the first line of the DRAT proof, clause -3, can be derived using resolution (which satisfies RAT) from the fourth and fifth clauses of the DIMACS file: -1 -3 and 1 -3. Similarly, the next clause 1 2 can be derived via resolution from the second and third clauses: 2 -4 and 1 2 4. The remaining clauses can be derived via similar resolution steps. The deletion step d 1 2 removes the clause 1 2 from the accumulated formula, reducing the proof search space. As the derivation ends with the empty clause, unsatisfiability of the original formula is proved.*

B. Inprocessing in Incremental SAT Solving

Inprocessing in SAT relies on adding and deleting *redundant* clauses both *before* and *during* solving. A clause C is considered redundant with respect to a formula F if F and $F \wedge C$ are equisatisfiable. An overview of common inprocessing techniques is given in [8]. In practice, solvers often delete clauses that are not implied but still redundant. When it comes to clause additions, however, they usually only add clauses that are implied (clauses that are not implied are only learned by special techniques like extended resolution [17], blocked-clause addition [18], and satisfaction-driven clause learning [19], which most solvers don't use by default). Fazekas et al. [10] have recently presented a calculus to capture the inner workings of modern incremental SAT solvers. Intuitively, a solver whose solving process can be expressed by the calculus is guaranteed to return correct results. We refer the interested reader to the paper for details. Since our work builds on their results, we give a high-level overview here.

The calculus consists of seven derivation rules that operate over triples $\langle \varphi, \rho, \sigma \rangle$, where φ is called the set of *irredundant clauses*, ρ is the set of *redundant clauses*, and σ is the so-called *reconstruction stack*, which we describe in detail later.

Most practical techniques in non-incremental SAT solving without inprocessing are captured by four derivation rules called LEARN^- , STRENGTHEN , FORGET , and DROP .¹ Intuitively, the calculus starts out with all input clauses in the irredundant set φ and adds new implied clauses to the redundant set ρ via the LEARN^- rule. Clauses from the redundant set ρ can be moved to the irredundant set φ via the unconditional STRENGTHEN rule. The FORGET rule enables the unconditional deletion of clauses from ρ ,

and the DROP rule enables the deletion of clauses from φ if they are implied.

The calculus contains three more rules— WEAKEN^+ , ADDCLAUSES , and RESTORE —that enable the sound combination of inprocessing and incremental solving. These three rules interact with the reconstruction stack σ , which in practice is a crucial ingredient for solvers that perform non-trivial preprocessing or inprocessing. Formally, the reconstruction stack is a sequence $(\omega_1 : C_1), \dots, (\omega_n : C_n)$, where each C_i is a clause and each ω_i is a set of literals (called the *witness*) such that $C_i \cap \omega_i \neq \emptyset$; $(\omega_i : C_i)$ is also called a *witness-labeled clause*.

Using the WEAKEN^+ rule, clauses of the irredundant set φ can be removed if they are determined to be equisatisfiability-redundant. When such clauses are deleted during solving, however, the solver might later find an assignment that satisfies the resulting formula but not the deleted clauses. To efficiently recover a satisfying assignment of the original formula, the solver stores these clauses on the reconstruction stack to later perform *model reconstruction*. When performing model reconstruction, the solver starts with the assignment α and iterates over the reconstruction stack in reverse order, checking for each witness-labeled clause $(\omega : C)$ whether C is satisfied by α . If C is satisfied, it can be skipped, otherwise α is modified by making all literals in ω true (denoted by $\alpha \circ \omega$).

Formally, the reconstruction function \mathcal{R} , which maps an assignment and a reconstruction stack to a new assignment, is defined as follows (ϵ denotes the empty sequence; concatenation of sequences σ and σ' is denoted by $\sigma \cdot \sigma'$):

$$\begin{aligned} \mathcal{R}(\alpha, \epsilon) &= \alpha, \\ \mathcal{R}(\alpha, \sigma \cdot (\omega : C)) &= \begin{cases} \mathcal{R}(\alpha, \sigma) & \text{if } \alpha(C) = 1 \\ \mathcal{R}(\alpha \circ \omega, \sigma) & \text{otherwise} \end{cases} \end{aligned}$$

Since $C \cap \omega \neq \emptyset$, making ω true also makes C true, but the solver must ensure that making ω true does not falsify any of the other clauses. For non-incremental SAT solving, all state-of-the-art inprocessing techniques generate witness-labeled clauses in such a way that this is guaranteed, and in most cases (like bounded variable elimination [20], pure-literal elimination, or blocked-clause elimination [21]), ω consists of only a single literal.

When incremental solving comes into play, however, things get tricky because the deletion of non-implied clauses can weaken a formula. The naive addition of clauses later on during incremental calls can then lead to unsound results. This observation led Fazekas et al. to the introduction of the rules ADDCLAUSES and RESTORE in their calculus. Before explaining the two rules, we give an example to illustrate the problem.

Example 2 (from [10]). *Consider the Boolean formula $F = (a \vee b) \wedge (\bar{a} \vee \bar{b})$. This formula is clearly satisfiable, and there are inprocessing techniques (e.g., blocked-clause elimination) that would delete the clause $(\bar{a} \vee \bar{b})$ to obtain $F' = (a \vee b)$, which is also satisfiable. Now, assume that at*

¹The minus symbol in the rule LEARN^- and the plus symbol in the later rule WEAKEN^+ are used because the rules are modified versions of rules LEARN and WEAKEN from an earlier calculus described in [16].

the next incremental call, the unit clauses (a) and (b) are added. The formula $F \wedge (a) \wedge (b) = (a \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a) \wedge (b)$ is then unsatisfiable whereas $F' \wedge (a) \wedge (b) = (a \vee b) \wedge (a) \wedge (b)$ is satisfiable. The deletion of $(\bar{a} \vee \bar{b})$ thus weakened the formula too much, leading to unsound results.

The key insight for incremental solving from [10] is that whenever a new set Δ_i of clauses is added to the problem at an incremental solver call, the solver can ensure soundness by moving some of the previously-deleted clauses back from the reconstruction stack σ to the set φ of irredundant clauses. This process is called *clause restoration*, and it is based on the notion of a *clean* clause.

Definition 1 (Clean Clause [10]). *A clause C is clean with respect to a sequence of witness-labeled clauses σ if for all $(\omega : C') \in \sigma$, we have that $\{\bar{l} \mid l \in C\} \cap \omega = \emptyset$.*

A clause is thus clean with respect to σ if it does not contain the negations of literals that serve as witnesses in σ . Cleanliness of a clause ensures that whenever model reconstruction takes place at the end of incremental solving, the truth of clean clauses is not affected because making a witness ω true cannot falsify clean clauses.

Given that satisfiability is preserved for clean clauses, when we add a set Δ of new clauses, we must ensure that they are clean with respect to the current reconstruction stack. The ADDCLAUSES rule thus has the precondition that only clean clauses can be added. Taken at face value, this would seem to make it very difficult to perform inprocessing with incremental solving: what if we need to add clauses that are not clean? This situation is where the RESTORE rule comes into play. The RESTORE rule allows moving a clause from the reconstruction stack back to the set of irredundant clauses as long as the clause is clean with respect to the subsequent portion of the reconstruction stack:

$$\text{RESTORE: } \frac{\langle \varphi, \rho, \sigma \cdot (\omega : C) \cdot \sigma' \rangle}{\langle \varphi \wedge C, \rho, \sigma \cdot \sigma' \rangle} \quad (C \text{ is clean w.r.t. } \sigma')$$

Thus, whenever we want to add unclean clauses at an incremental solver call, we can turn them into clean clauses by first restoring all labeled clauses of the reconstruction stack that would make them unclean. This is achieved with the algorithm `RestoreAddClauses` [10] shown in Fig. 2. The algorithm iterates over the stack starting at the bottom, continuously restoring clauses that would prevent cleanliness of the new clauses. To make sure that the precondition of the RESTORE rule (C is unclean w.r.t. σ') is fulfilled, it also restores clauses that would prevent cleanliness of previously restored clauses.

With this background, we can now state the problem we are trying to solve.

III. PROBLEM STATEMENT

The work in [10] presents a sound calculus for solvers to perform incremental solving with inprocessing, but it does

```

1: function RESTOREADDCLAUSES( $\Delta$ : clauses,  $\sigma$ : stack)
2:    $(\omega_1 : C_1), \dots, (\omega_n : C_n) := \sigma$ 
3:   for  $i$  from 1 to  $n$  do
4:     if exists  $l \in \omega_i$  where  $\bar{l}$  occurs in  $\Delta$  then
5:        $\Delta := \Delta \cup \{C_i\}$ ,  $\sigma := \sigma \setminus (\omega_i : C_i)$ 
6:   return  $\langle \Delta, \sigma \rangle$ 

```

Fig. 2. Algorithm `RestoreAddClauses` to restore clauses.

not define how a solver based on that calculus could efficiently produce an independently-checkable proof. This is also the reason why the SAT solver CaDiCaL, which in [10] was augmented with the `RestoreAddClauses` algorithm, does not produce valid proofs when using inprocessing during incremental solving.

Our goal is to obtain a valid DRAT proof from a solver based on the incremental inprocessing calculus in [10]. The calculus requires that all clauses derived by a solver are implied. Strictly speaking, this would allow the addition of implied clauses that do not necessarily have the RAT property. In practice, however, all state-of-the-art CDCL solvers derive only clauses with the so-called RUP (short for *Reverse Unit Propagation*) property—a simpler property guaranteeing that the clauses are implied *and* have the RAT property (see [22], [23] for details on RUP). Hence, we require that solvers derive clauses with the RUP property, meaning that our approach applies to virtually all existing solvers. RUP is monotonic: If a clause has the RUP property with respect to a formula F , it also has it with respect to each superset of F .

Proof checkers for DRAT require as input both a formula and a corresponding proof. We produce proofs in such a way that they can be checked against the formula consisting of all incrementally added clauses, plus a unit clause for each literal in the final assumption. More formally, let $\langle \Delta_0, A_0 \rangle, \dots, \langle \Delta_n, A_n \rangle$ be an unsatisfiable incremental problem. We then produce a DRAT proof of the formula $\Delta_0 \cup \dots \cup \Delta_n \cup \{(l) \mid l \in A_n\}$.

The four derivation rules `LEARN-`, `STRENGTHEN`, `FORGET`, and `DROP` can be accommodated in a straightforward way in DRAT: `LEARN-` corresponds to a clause addition (of the learned clause), `STRENGTHEN` is not reflected in the proof (as DRAT does not distinguish between redundant and irredundant clauses), and both `FORGET` and `DROP` correspond to clause deletions in DRAT.

Applications of `ADDCLAUSES` don't need to be represented in the proof trace because we simply consider all clauses that are added incrementally part of the initial formula. Thus, in the DRAT proof trace, everything that was derived using these clauses can still be derived because they are part of the proof's accumulated formula.

The problem is how to express applications of `WEAKEN+` and corresponding applications of `RESTORE`. One option is to represent `WEAKEN+` by clause deletion. However, if we then naively express `RESTORE` by clause addition, we run into soundness problems. Specifically,

a clause that had the RAT property at the point of WEAKEN⁺/deletion (which is the case for most clauses deleted during practical inprocessing) might not have the RAT property at the point of restoration anymore. This would render the clause addition in the proof invalid. Intuitively, this is because the RAT property is influenced by additions and deletions happening between a clause's initial deletion and its restoration. For readers familiar with the details of the RAT property, the following example demonstrates this on a concrete formula:

Example 3. Consider the formula $(\bar{x} \vee y) \wedge (x \vee \bar{z}) \wedge (\bar{y} \vee z) \wedge (\bar{x} \vee z)$. The clause $(\bar{y} \vee z)$, which has the RAT property (as an interested reader might convince themselves of) can be deleted and put on the reconstruction stack. The clause $(\bar{x} \vee z)$ can be subsequently deleted because it has the RAT property with respect to the remaining clauses. We end up with the formula $(\bar{x} \vee y) \wedge (x \vee \bar{z})$. Now assume that we want to restore the clause $(\bar{y} \vee z)$, e.g., because we want to solve under the assumption (y) . If we tried to add the clause to a DRAT proof, the proof would become invalid because $(\bar{y} \vee z)$ does not have the RAT property with respect to the remaining formula—the clause $(x \vee \bar{z})$ was crucial for establishing the RAT property, but it has been deleted.

Because solvers can use restored clauses to derive further clauses, the restored clauses need to enter the accumulated formula of the proof, otherwise those derivations become invalid. We thus need another way to express RESTORE in a DRAT proof.

IV. ALGORITHM FOR PROOF PRODUCTION

To handle applications of RESTORE in DRAT proofs and thus support proofs for incremental solving with inprocessing, we propose the following approach:

- We first produce an *augmented* proof trace (which itself is not a valid DRAT proof) where all applications of WEAKEN⁺ are logged as deletions and all applications of RESTORE are logged with a new dedicated proof rule.
- In a post-processing step, we then make use of the restoration information to convert the augmented proof trace into a valid DRAT proof.

The idea is surprisingly simple: Instead of trying to add restored clauses to the proof via clause additions, we act as if they hadn't been deleted in the first place.

Specifically, whenever a solver applies the RESTORE rule to restore a clause C from the reconstruction stack, we add to the proof trace a novel statement of the form $\langle r, C \rangle$. The resulting augmented proof trace is a sequence $\langle s_1, C_1 \rangle, \dots, \langle s_m, C_m \rangle$, where each $s_i \in \{a, d, r\}$ and each C_i is a clause. Similar to a normal DRAT proof trace, we define an accumulated formula as follows:

$F_0 =$ clauses in the original problem

$$F_i = \begin{cases} F_{i-1} \cup \{C_i\} & \text{if } s_i \in \{a, r\} \\ F_{i-1} \setminus \{C_i\} & \text{if } s_i = d \end{cases}$$

Given that, as assumed earlier, all clauses derived by the solver are RUP (and thus RAT) clauses with respect to the accumulated formula, this holds for all clause additions in the augmented proof as well. Only the clause restorations are not justified, but we deal with them during post-processing.

The post-processing, which we describe in detail below, is formalized via the post-processing function Φ , which takes as arguments an augmented proof trace P together with an (initially empty) set R of restored clauses.

$$\Phi(\epsilon, R) = \epsilon$$

$$\Phi(P \cdot \langle s, C \rangle, R) = \begin{cases} \Phi(P, R) \cdot \langle a, C \rangle & \text{if } s = a \\ \Phi(P, R \cup \{C\}) & \text{if } s = r \\ \Phi(P, R \setminus \{C\}) & \text{if } s = d \wedge C \in R \\ \Phi(P, R) \cdot \langle d, C \rangle & \text{if } s = d \wedge C \notin R \end{cases}$$

Intuitively, Φ traverses the augmented proof trace in reverse order, statement by statement. When the algorithm encounters a clause addition, it simply adds it to the new proof trace. When it encounters a clause restoration, it adds the clause to the set R but does not add anything to the new proof trace. Finally, when it encounters a deletion, it checks whether the deleted clause is contained in R , and if so, removes the deleted clause from R without changing the proof trace, otherwise it just adds the deletion to the new proof trace. The resulting proof trace is a subsequence of the original proof P .

Example 4. Consider the augmented proof trace $P = \langle a, x \vee y \rangle, \langle d, y \vee z \rangle, \langle d, \bar{z} \vee u \rangle, \langle r, y \vee z \rangle, \langle a, x \vee \bar{u} \rangle$. At the beginning, $R := \emptyset$. The post-processing function Φ traverses P in reverse order, starting with $\langle a, x \vee \bar{u} \rangle$, which it appends to the result of the recursive call for the remainder of the list: $\Phi(\langle a, x \vee y \rangle, \langle d, y \vee z \rangle, \langle d, \bar{z} \vee u \rangle, \langle r, y \vee z \rangle, \emptyset) \cdot \langle a, x \vee \bar{u} \rangle$. It next encounters the restoration $\langle r, y \vee z \rangle$ and thus sets $R := \{y \vee z\}$ for the recursive call. When it next encounters the deletion $\langle d, \bar{z} \vee u \rangle$, it adds it to the processed proof because the clause is not in the restore set R : $\Phi(\langle a, x \vee y \rangle, \langle d, y \vee z \rangle, \{y \vee z\}) \cdot \langle d, \bar{z} \vee u \rangle, \langle a, x \vee \bar{u} \rangle$. However, for the next deletion, $\langle d, y \vee z \rangle$, the clause is already in R . Thus the deletion is not added to the processed proof but instead, the clause is removed from R . Finally, the addition $\langle a, x \vee y \rangle$ is also added to the processed proof, so we end up with the proof $\langle a, x \vee y \rangle, \langle d, \bar{z} \vee u \rangle, \langle a, x \vee \bar{u} \rangle$.

As we can see in Example 4, the clause $(y \vee z)$ was restored during solving and thus $\langle r, y \vee z \rangle$ was part of the augmented proof trace. In the final proof trace, however, the clause is never even deleted. Intuitively, the consequence for the accumulated formula is that it always contains

all the clauses required to make further derivations. More specifically, consider the augmented proof trace P and the processed proof P' , where the latter is a subsequence of the former. If we compare the accumulated formulas of each proof after any proof statement $\langle s_i, C_i \rangle$ contained in both P' and P , we observe that the accumulated formula with respect to P' contains all clauses of the accumulated formula with respect to P .

Since we assume all added clauses are RUP (and thus RAT) clauses, and since RUP is monotonic, we obtain a valid DRAT proof in which all clause additions fulfill the RAT property.

V. IMPLEMENTATION

We implemented our approach on top of the incremental inprocessing SAT solver CaDiCaL. In particular, we added the capability to produce augmented proofs with restore statements. As restorations only occur during incremental solving with inprocessing, our changes to CaDiCaL do not impact non-incremental solving. Additionally, we implemented proof post-processing in a dedicated tool. The tool traverses a proof backwards, printing all proof statements immediately (instead of prepending them to an internal data structure) to keep the memory requirements low. Since this leads to a reversed proof, we reverse it again at the end to get a valid DRAT proof. Our toolchain can produce DRAT proofs in both the plain-text format and the binary format.

In order to obtain valid proofs, we made changes to CaDiCaL to maintain the invariant that each clause restoration is preceded by a corresponding deletion. This invariant, which we used when arguing about correctness of our approach, is guaranteed to hold if a solver records all applications of the WEAKEN⁺ rule as deletions in the proof. In practice, however, there are additional subtleties that need to be taken care of to make sure this is the case. We now explain the three most important changes to CaDiCaL to provide solver developers with guidance for implementing our approach on top of other solvers.

A. Deletion of Binary Clauses

When CaDiCaL logically deletes a clause during solving, it immediately marks the clause as deleted but only later, during garbage collection, really removes it from memory. For non-binary clauses, it logs a deletion statement to the proof immediately after logical deletion. For binary clauses, however, the deletion is only logged once the clause is really removed during garbage collection. For binary clauses, this could lead to the case where a clause is first marked deleted and then restored, but the deletion is only logged in the proof trace after restoration. The fix is simple: ensure garbage collection is triggered before restoration; thus all deletions occur in the proof before their corresponding restorations.

B. Proper Handling of Equivalent-Literal Substitution

CaDiCaL performs an inprocessing technique called *equivalent-literal substitution* [24] (“decompose” in CaDiCaL’s code). The technique identifies equivalent literals in the binary implication graph of a formula and then substitutes a single representative literal for each equivalence class of literals. For example, if x and y are identified as equivalent, CaDiCaL might replace all occurrences of y by x . To later reconstruct a proper model for the removed literals, CaDiCaL adds to the reconstruction stack an equivalence $(\bar{x} \vee y) \wedge (x \vee \bar{y})$ for each removed y and representative x . These clauses are never explicitly added or deleted and thus they are not represented in the proof. However, as they are on the reconstruction stack, CaDiCaL may restore them. To deal with this situation, we add the equivalences to the proof (which is allowed because they are trivially RUP) and then immediately delete them again. This allows us to remove the deletions during post-processing, ensuring proper derivation of the equivalences in the proof.

C. Internal and External Representations of Literals

CaDiCaL maintains both an internal and an external representation of literals, together with mappings between these representations. This is because the solver sometimes removes literals and then remaps the remaining ones to save memory (e.g., when performing equivalent-literal substitution as above). This can lead to problems when restoring clauses. For example, when a clause is restored and immediately simplified (because some of its literals are falsified or satisfied at the top level), the solver deletes the original clause and adds the simplified clause to the proof. In particular, the solver first maps the restored clause to an internal representation before remapping it to an external representation when performing the deletion—this “round trip” can lead to a different external representation than the one originally deleted. In our implementation, we modified the corresponding code to ensure that the deleted and restored clauses match in the proof.

VI. EVALUATION

To evaluate our approach in practice, we plugged our proof-producing version of CaDiCaL into CaMiCaL [10], a SAT-based bounded model checker that runs on AIGER models [25] used in the hardware model checking competition (HWMCC) [13]. Following the experiment in [10], we ran CaMiCaL on the 300 models of the single safety property track of HWMCC’17 [13], up to bound 1000 with a time limit of 3600 seconds per model. In [10], it was shown that on these models, enabling inprocessing with clause restoration leads to a significant performance improvement compared to disabling inprocessing or running it only in a restricted way, e.g., by *freezing* [6] certain variables.

We ran our experiments on an Amazon EC2 m5d.metal instance, which has an AWS-custom Intel Xeon Scalable (Skylake) processor with 96 vCPUs, 384 GiB memory, and

four 900 GB SSDs, running Amazon Linux 2. We ran 24 benchmark processes in parallel.

Our primary goal was to demonstrate that our approach produces valid DRAT proofs. For each model of the benchmark set, we performed a CaMiCaL run and generated a DRAT proof for the highest unsatisfiable bound solved by CaMiCaL within the time limit. This means that for unsatisfiable problems, we took the highest solved bound whereas for satisfiable problems, we took the second highest solved bound (as it yields an UNSAT result). There were four problems that were satisfiable at the first bound and another two problems for which CaMiCaL timed out while attempting to solve the first bound, leaving 294 problems with an UNSAT result.²

To check the correctness of the resulting DRAT proof for each problem, we extracted from CaMiCaL a DIMACS file including all clauses that were added incrementally to CaDiCaL during solving as well as unit clauses for the assumptions of the final unsatisfiable call (i.e., one unit clause per assumption). We then passed the DIMACS file together with the corresponding DRAT proof—obtained by post-processing the original proof trace with our tool—to the proof checker DRAT-trim [12]. For all 294 problems, DRAT-trim confirmed that the proofs were correct.

To get an idea of the overhead introduced by our post-processing approach, we measured the time spent on post-processing as compared to solving. On average, the overhead of post-processing was 5.3% of the actual solving time (i.e., the time spent inside the SAT solver, not the whole time spent by the model checker). The time to post-process is linear in the size of the proof trace, as shown in Fig. 3. The figure shows that our implementation takes a little more than a minute to post-process 1 GB of proof. While fast, the overhead varies considerably between problems, as the proof size is not a function of the solving time, i.e., two solver runs that take the same time might produce proofs of different sizes, as is illustrated in Fig. 4.

Next, we present performance measurements from our experiments to give an indication of the performance of our approach as compared to an alternate approach to generating a proof by solving the corresponding bound with CaDiCaL monolithically from scratch using non-incremental solving with proof generation enabled. For example, if CaMiCaL was able to solve n bounds of an incremental problem, we take the propositional formula corresponding to bound n and solve it non-incrementally with a new CaDiCaL instance that produces a proof. We then compare the time it takes CaDiCaL to produce that proof with the time it takes to post-process the incremental proof with our approach, to see which approach is more efficient. In this experiment, we gave the non-incremental CaDiCaL a timeout of 7200 seconds, i.e., twice

²The problems satisfiable at the first bound are 6s389b02.aig, bobminterbm1or.aig, bobsynth13.aig, and bobtuint24.aig; the problems for which CaMiCaL timed out at the first bound are 6s128.aig and 6s398b09.aig.

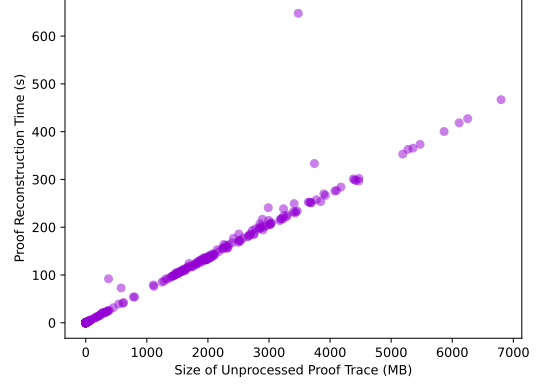


Fig. 3. Proof Reconstruction Time vs. Size of Proof Trace.

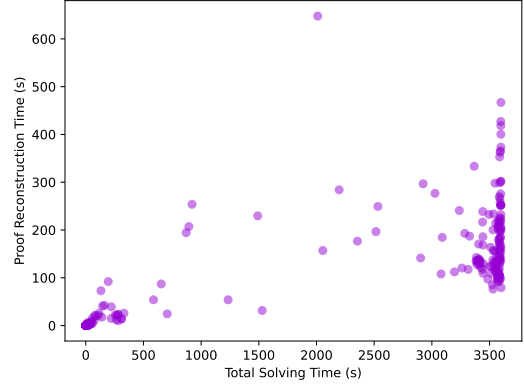


Fig. 4. Proof Reconstruction Time vs. Total Solving Time.

the CaMiCaL timeout. While we believe these numbers are informative, we note that they are not representative for general incremental SAT solving as they only apply to our restricted benchmark set (which we chose primarily to demonstrate soundness, as it triggers many clause restorations).

Fig. 5 compares the two approaches. Clearly, post-processing an incremental proof trace is much more efficient than re-solving a formula from scratch. In particular, there were 13 instances for which CaDiCaL timed out when trying to solve them in a single shot. This is not a surprise: there is no guarantee that a problem that is solvable incrementally can also be solved in a single shot, whereas post-processing a proof takes a small overhead that can be estimated based on the size of the unprocessed proof trace.

The results are less clear when we consider the sum of time spent on post-processing/single-shot solving and on checking the resulting proofs. For this comparison, we excluded the 13 instances for which CaDiCaL timed out when solving them in a single shot; Fig. 6 shows the results. Although our approach is 13% faster on average,

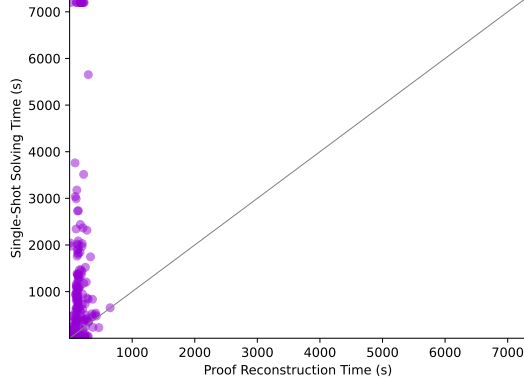


Fig. 5. Proof Reconstruction vs. Single-Shot Solving.

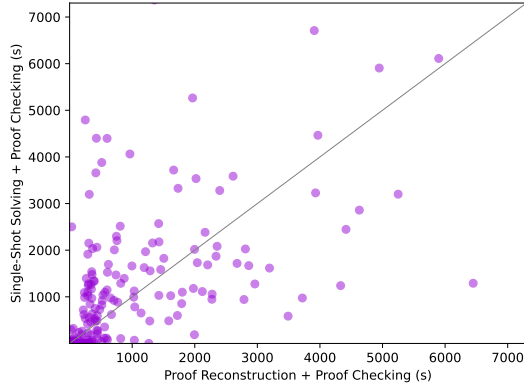


Fig. 6. Proof Reconstruction vs. Single-Shot Solving (incl. Checking).

there are several problems where the monolithic approach is faster. One reason for this is that incremental proofs are usually larger than the ones produced by single-shot solving, as illustrated in Fig. 7. To summarize, on the models of the single safety property track of HWMCC’17, post-processing incremental proofs is faster than producing proofs from scratch in a single shot, with the latter carrying the risk of timeouts. On the flip side, if single-shot solving succeeds, it tends to produce shorter proofs, which can in turn be checked faster.

VII. CONCLUSION AND FUTURE WORK

We have presented an efficient approach to generate proofs for inprocessing incremental solvers based on the calculus from [10]. We augment the DRAT [12] proof format by adding *restore* steps to the proof. These steps can be efficiently removed during post-processing, yielding a standard DRAT proof. We implemented the approach on top of the CaDiCaL [26] solver and demonstrated its soundness and efficiency against the benchmark suite from HWMCC’17 [13]. For these benchmarks, proof generation adds 2.9% average overhead to solving. Post-processing adds 5.3% average overhead.

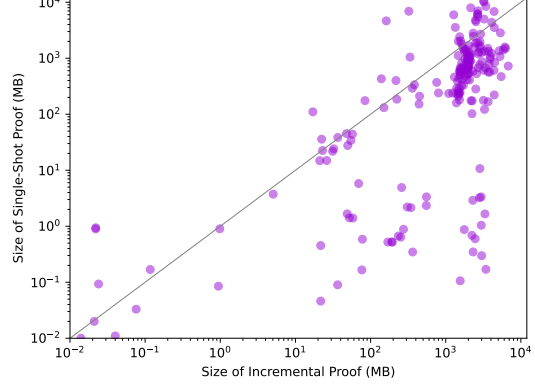


Fig. 7. Proof Size: Incremental vs. Single-Shot Solving (log scale).

We compared our approach to a monolithic one in which we solved the final formula in a single shot and generated proofs during this step. If solving times are compared, our approach is faster, as it does not require a “from scratch” solve of the entire problem. As our incremental proofs are larger than the DRAT proofs generated by the monolithic solve, the picture including proof-checking times is mixed. Although our approach is faster on average when measuring the sum of solving plus proof checking times, there are examples where the situation is reversed.

There are two important advantages of our approach vs. re-solving the final problem monolithically. First, it always yields a solution, whereas the monolithic approach sometimes times out on our benchmark set. Second, it provides a foundation that can be adapted in future work towards efficiency and robustness improvements for inprocessing incremental solvers as discussed in the following.

When the LRAT support for CaDiCaL described in a future paper [27] is integrated into the main repository, we will convert the augmented DRAT format to LRAT (the deletions and restorations are the same). In the results in [27], converting, trimming, and checking proofs is approximately 5.5x faster than for DRAT; this will lower the overhead for proof checking in our approach relative to the solving time for the monolithic problem. We also plan to examine how to use augmented proof traces to migrate state of incremental solvers, extending the work from [28] to support incremental use-cases. By combining the migration approach with our proof approach, we can migrate solver state between multiple platforms and later combine the resulting incremental proof fragments. This support allows us to better utilize cloud resources and build solvers that can be restarted after machine failures. Finally, we can adapt our approach for use in distributed incremental solving, as was earlier demonstrated for monolithic solving [29].

REFERENCES

- [1] S. Gocht and T. Balyo, “Accelerating SAT based planning with incremental SAT solving,” in *Proceedings of the Twenty-*

- Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, L. Barbulescu, J. Frank, Mausam, and S. F. Smith, Eds. AAAI Press, 2017, pp. 135–139. [Online]. Available: <https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15580>
- [2] R. Sebastiani, “Lazy satisfiability modulo theories,” *J. Satisf. Boolean Model. Comput.*, vol. 3, no. 3-4, pp. 141–224, 2007. [Online]. Available: <https://doi.org/10.3233/sat190034>
 - [3] P. Mishra and M. Chen, “Efficient techniques for directed test generation using incremental satisfiability,” in *Proceedings of the 2009 22nd International Conference on VLSI Design*, ser. VLSID ’09. USA: IEEE Computer Society, 2009, p. 65–70. [Online]. Available: <https://doi.org/10.1109/VLSI.Design.2009.72>
 - [4] A. Yamada, T. Kitamura, C. Artho, E.-H. Choi, Y. Oiwa, and A. Biere, “Optimization of combinatorial testing by incremental sat solving,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.
 - [5] A. Biere, “Bounded model checking,” in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 739–764. [Online]. Available: <https://doi.org/10.3233/FAIA201002>
 - [6] N. Eén and N. Sörensson, “Temporal induction by incremental SAT solving,” *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003. [Online]. Available: [https://doi.org/10.1016/S1571-0661\(05\)82542-3](https://doi.org/10.1016/S1571-0661(05)82542-3)
 - [7] S. Kupferschmid, M. Lewis, T. Schubert, and B. Becker, “Incremental preprocessing methods for use in BMC,” *Formal Methods Syst. Des.*, vol. 39, no. 2, pp. 185–204, 2011. [Online]. Available: <https://doi.org/10.1007/s10703-011-0122-4>
 - [8] A. Biere, M. Järvisalo, and B. Kiesl, “Preprocessing in SAT solving,” *Handbook of Satisfiability*, vol. 336, pp. 391–435, 2021.
 - [9] A. Nadel, V. Ryvchin, and O. Strichman, “Preprocessing in incremental SAT,” in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 256–269. [Online]. Available: https://doi.org/10.1007/978-3-642-31612-8_20
 - [10] K. Fazekas, A. Biere, and C. Scholl, “Incremental inprocessing in SAT solving,” in *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 136–154. [Online]. Available: https://doi.org/10.1007/978-3-030-24258-9_9
 - [11] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracocoa, Plingeling and Treengeling entering the SAT Competition 2020,” in *Proc. of SAT Competition 2020 - Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.
 - [12] M. J. H. Heule, “The DRAT format and drat-trim checker,” *CoRR*, vol. abs/1610.06229, 2016. [Online]. Available: <http://arxiv.org/abs/1610.06229>
 - [13] A. Biere, T. van Dijk, and K. Heljanko, “Hardware model checking competition 2017,” in *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, p. 9. [Online]. Available: <https://doi.org/10.23919/FMCAD.2017.8102233>
 - [14] N. Wetzler, M. J. Heule, and W. A. H. Jr., “DRAT-trim: Efficient checking and trimming using expressive clausal proofs,” in *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer, 2014, pp. 422–429. [Online]. Available: https://doi.org/10.1007/978-3-319-09284-3_31
 - [15] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185. [Online]. Available: <http://dblp.uni-trier.de/db/series/faia/faia185.html>
 - [16] M. Järvisalo, M. J. Heule, and A. Biere, “Inprocessing rules,” in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370. [Online]. Available: https://doi.org/10.1007/978-3-642-31365-3_28
 - [17] G. Audemard, G. Katsirelos, and L. Simon, “A restriction of extended resolution for clause learning SAT solvers,” in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, M. Fox and D. Poole, Eds. AAAI Press, 2010. [Online]. Available: <http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1811>
 - [18] O. Kullmann, “On a generalization of extended resolution,” *Discret. Appl. Math.*, vol. 96-97, pp. 149–176, 1999. [Online]. Available: [https://doi.org/10.1016/S0166-218X\(99\)00037-2](https://doi.org/10.1016/S0166-218X(99)00037-2)
 - [19] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere, “Pruning through satisfaction,” in *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, ser. Lecture Notes in Computer Science, O. Strichman and R. Tzoref-Brill, Eds., vol. 10629. Springer, 2017, pp. 179–194. [Online]. Available: https://doi.org/10.1007/978-3-319-70389-3_12
 - [20] N. Eén and A. Biere, “Effective preprocessing in SAT through variable and clause elimination,” in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75. [Online]. Available: https://doi.org/10.1007/11499107_5
 - [21] M. Järvisalo, A. Biere, and M. Heule, “Blocked clause elimination,” in *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 129–144. [Online]. Available: https://doi.org/10.1007/978-3-642-12002-2_10
 - [22] E. I. Goldberg and Y. Novikov, “Verification of proofs of unsatisfiability for CNF formulas,” in *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 2003, pp. 10 886–10 891. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/DATE.2003.10008>
 - [23] A. V. Gelder, “Verifying RUP proofs of propositional unsatisfiability,” in *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008, 2008*. [Online]. Available: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf
 - [24] M. Heule, M. Järvisalo, and A. Biere, “Efficient CNF simplification based on binary implication graphs,” in *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 201–215. [Online]. Available: https://doi.org/10.1007/978-3-642-12581-0_17
 - [25] A. Biere, T. van Dijk, and K. Heljanko, “Aiger 1.9 and beyond,” Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstrasse 69, 4040 Linz, Austria, FMV Reports Series, 2011.
 - [26] A. Biere, “CaDiCaL at the SAT Race 2019,” in *Proc. of SAT Race 2019 - Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.

- [27] F. Pollitt, M. Fleury, and A. Biere, “Efficient proof checking with *lrat* in *cadical* (work in progress),” in *26th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-23, 2023*, A. Biere and D. Groÿe, Eds. VDE, 2023, pp. 64–67, accepted.
- [28] A. Biere, M. S. Chowdhury, M. J. Heule, B. Kiesl-Reiter, and M. Whalen, “Migrating solver state,” in *SAT 2022*, 2022. [Online]. Available: <https://www.amazon.science/publications/migrating-solver-state>
- [29] D. Michaelson, D. Schreiber, M. J. Heule, B. Kiesl-Reiter, and M. Whalen, “Unsatisfiability proofs for distributed clause-sharing sat solvers,” in *TACAS 2023*, 2023. [Online]. Available: <https://www.amazon.science/publications/unsatisfiability-proofs-for-distributed-clause-sharing-sat-solvers>