






Propositional Proof Skeletons^{*}

Joseph E. Reeves¹ (✉) , Benjamin Kiesl-Reiter² , and Marijn J. H. Heule^{1,2} 

¹ Carnegie Mellon University, Pittsburgh, PA, USA

{jereeves,mheule}@cs.cmu.edu

² Amazon Web Services, Seattle, WA, USA

benkiesl@amazon.com

Abstract. Modern SAT solvers produce proofs of unsatisfiability to justify the correctness of their results. These proofs, which are usually represented in the well-known DRAT format, can often become huge, requiring multiple gigabytes of disk storage. We present a technique for semantic proof compression that selects a subset of important clauses from a proof and stores them as a so-called proof skeleton. This proof skeleton can later be used to efficiently reconstruct a full proof by exploiting parallelism. We implemented our approach on top of the award-winning SAT solver CaDiCaL and the proof checker DRAT-trim. In an experimental evaluation, we demonstrate that we can compress proofs into skeletons that are 100 to 5,000 times smaller than the original proofs. For almost all problems, proof reconstruction using a skeleton improves the solving time on a single core, and is around five times faster when using 24 cores.

Keywords: SAT solving · proofs · compression.

1 Introduction

Solvers for the Boolean satisfiability problem (SAT) take as input a formula of propositional logic and decide if the formula is satisfiable. In case of satisfiability, they usually return an assignment of truth values to the variables of the formula; by plugging these truth values into the formula, users can easily convince themselves that the solver was right and that the formula is indeed satisfiable. In case of unsatisfiability, however, things are more complicated: to justify their answer, solvers need to produce an independently checkable proof that none of the—exponentially many—potential truth assignments make the formula true.

In practical SAT solving, proofs of unsatisfiability are represented in the DRAT format [10], and they are often huge, requiring several gigabytes (in some cases even terabytes [12] or petabytes [11]) of disk storage. Storing proofs is thus costly, especially since users might not require access to the proofs until sometime long after solving, at a point when proof verification or further analysis is desired.

^{*} Supported by the U.S. National Science Foundation under grant CCF-2229099, and supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

Up to now, the only options to deal with this problem were either to not store proofs and instead recompute them on demand—a laborious but plausible approach considering that proof checking typically takes longer than solving—or to use compression methods to reduce proof size. However, syntactic compression techniques (such as LZMA or DEFLATE, as supported by the ZIP file format) only provide moderate levels of compression. The same can be said about existing semantic compression techniques for proofs in SAT and SMT (c.f. [4, 18, 21]), which only achieve 20% compression on average.

In this paper, we present a novel approach to semantic compression that stores only a small subset of the clauses derived by a solver, called a *proof skeleton*. We can achieve strong compression rates with proof skeletons (around 100 to 5,000 times smaller than the original proof), while still retaining enough information to allow for a quick on-demand reconstruction of a complete proof that might differ from the original proof. This is similar to how a mathematician might put down the most important reasoning steps of a proof in a proof sketch, enabling a moderately talented reader to fill in the gaps. In our case, the gaps can even be filled independently, meaning that multiple readers can work in parallel.

We present both an online version (creating a proof skeleton during solving) and an offline version (creating a proof skeleton from a full proof) of our approach. We select the clauses that end up in a proof skeleton by relying on several heuristics such as *glue* (a heuristic used internally by solvers to estimate the usefulness of clauses) for online and *clause activity* (a measure of how often a clause is used to derive new clauses) for offline. To reconstruct a full proof from a proof skeleton, we utilize multiple incremental SAT solvers that can run in parallel. We implemented all our algorithms on top of the award-winning SAT solver CADICAL [2] and the proof checker DRAT-TRIM [22]. In an extensive empirical evaluation, we demonstrate the feasibility of our approach, with all code and data available at <https://github.com/amazon-science/unsat-proof-skeletons>.

Beyond being a tool for compression, proof skeletons can also serve as a source of insight into a solver’s reasoning. Getting any sort of intuition from a million-line proof is difficult; by computing a skeleton, we obtain a small set of facts—logically implied by the problem—that can give us an idea of how a solver established the unsatisfiability of a formula. This can lead to a feedback loop that improves solver performance. For example, when inspecting skeletons for some bounded-model-checking benchmarks, we observed many unit clauses and binary clauses of a certain type. From this, we hypothesized that the problems required more preprocessing, which did indeed improve performance.

Our main contributions are as follows: (1) We present a semantic approach for proof compression that selects only the most important clauses of a proof. (2) We implemented an online version and an offline version of our approach on top of the SAT solver CADICAL and the proof checker DRAT-TRIM. (3) In an extensive empirical evaluation, we demonstrate that our approach can drastically reduce proof size while still enabling efficient proof reconstruction.

The rest of this paper is structured as follows. In Section 2, we discuss background required to understand our paper and review related work. In Section 3,

we outline the main idea behind our proof-compression approach. In Section 4, we show multiple ways to create proof skeletons, and in Section 5 we show how to reconstruct full proofs from skeletons. Finally, in Section 6, we present an empirical evaluation of our approach before concluding in Section 7.

2 Background and Related Work

The Boolean satisfiability problem (SAT) takes as input a formula of propositional logic and asks if there exists a truth assignment under which the formula evaluates to true. As is common in SAT solving, we consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is either a variable x (a *positive literal*) or the negation \bar{x} of a variable x (a *negative literal*). The *complement* \bar{l} of a literal l is defined as $\bar{l} = \bar{x}$ if $l = x$ and as $\bar{l} = x$ if $l = \bar{x}$. For a literal l , we denote the variable of l by $\text{var}(l)$. A *clause* is a finite disjunction of the form $(l_1 \vee \dots \vee l_n)$, where l_1, \dots, l_n are literals. Clauses with only one literal are called *unit clauses* and clauses with two literals are called *binary clauses*. We denote the empty clause by \perp . A *formula* is a finite conjunction of the form $C_1 \wedge \dots \wedge C_m$, where C_1, \dots, C_m are clauses. For example, $(x \vee \bar{y}) \wedge (z) \wedge (\bar{x} \vee \bar{z})$ is a formula consisting of the clauses $(x \vee \bar{y})$, (z) , and $(\bar{x} \vee \bar{z})$.

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 l is *satisfied* by an assignment α if l is positive and $\alpha(\text{var}(l)) = 1$ or if l is negative and $\alpha(\text{var}(l)) = 0$. A literal l is *falsified* by an assignment if its complement \bar{l} is satisfied by the assignment. A clause C is satisfied by an assignment α if α satisfies at least one of C 's literals. A formula ψ is satisfied by an assignment α if α satisfies all of ψ 's clauses. A formula is *satisfiable* if there exists an assignment that satisfies it, otherwise it is *unsatisfiable*. A clause $C = (l_1 \vee \dots \vee l_k)$ is *implied* by a formula ψ , denoted by $\psi \models C$, if all satisfying assignments of ψ satisfy C , or equivalently, if $\psi \wedge \bar{C}$ is unsatisfiable, where $\bar{C} = (\bar{l}_1) \wedge \dots \wedge (\bar{l}_k)$. In case a formula is satisfiable, modern solvers can output a satisfying assignment; in case the formula is unsatisfiable, most solvers can output a proof of unsatisfiability.

Proofs of Unsatisfiability. State-of-the-art SAT solvers produce so-called *clausal proofs*. Intuitively, a clausal proof is a list of clause additions and clause deletions. Formally, a clausal proof is a list of pairs $\langle s_1, C_1 \rangle, \dots, \langle s_m, C_m \rangle$, where for each $i \in 1, \dots, m$, $s_i \in \{\mathbf{a}, \mathbf{d}\}$ and C_i is a clause. If $s_i = \mathbf{a}$, the pair is called an *addition*, and if $s_i = \mathbf{d}$, it is called a *deletion*. For a given input formula ψ_0 , a clausal proof gives rise to *accumulated formulas* ψ_i ($i \in 1, \dots, m$) as follows:

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

The clauses of an accumulated formula ψ_i are also called the *active clauses* at point i . Clause additions must preserve satisfiability, which is usually guaranteed by requiring the added clauses to fulfill some efficiently decidable syntactic

criterion that itself implies satisfiability is preserved. Deletions are unrestricted and are not useful for proving unsatisfiability as they only make a formula “more satisfiable”; their main purpose is to speed up proof checking by keeping the set of active clauses small. A valid proof of unsatisfiability must end with the addition of the empty clause. As the empty clause is trivially unsatisfiable, and since all proof steps preserve satisfiability, the unsatisfiability of the original formula can then be concluded.

Clausal proof systems are distinguished by the syntactic criterion they impose on clause additions. The standard SAT solving paradigm *conflict-driven clause learning* (CDCL) [15,16] adds so-called *RUP* (short for *reverse unit propagation*) clauses [20], whose definition is based on the notion of *unit propagation*. Unit propagation is the process of repeatedly applying the *unit-clause rule* to a formula until no unit clauses are left. Given a formula ψ , the unit-clause rule takes a unit clause (l) and makes its literal l true, meaning that (1) all clauses that contain l are removed from ψ , and (2) the negation \bar{l} of l is removed from all remaining clauses. If unit propagation produces the empty clause, we say it derived a *conflict*. For example, unit propagation derives a conflict on $(x) \wedge (\bar{x} \vee y) \wedge (\bar{x} \vee \bar{y})$ as the application of the unit-clause rule for (x) produces the formula $(y) \wedge (\bar{y})$, on which another application of the unit-clause rule, with either of (y) or (\bar{y}) , produces the empty clause. If unit propagation derives a conflict on a formula, the formula is clearly unsatisfiable, but not vice versa.

A clause $C = (l_1 \vee \dots \vee l_k)$ is a RUP for a formula ψ if unit propagation derives a conflict on $\psi \wedge \bar{C}$. If C is a RUP for ψ , it is implied by ψ since $\psi \wedge \bar{C}$ is unsatisfiable; we thus sometimes write $\psi \vdash_1 C$ to denote that C is a RUP for ψ . The clausal proof system allowing the addition of RUP clauses together with deletions is called DRUP. Solvers participating in the SAT competition must produce DRAT proofs, but since each DRUP proof is also a DRAT proof (but not vice versa) and since all state-of-the-art solvers actually produce DRUP proofs by default, we restrict this study of proof compression to DRUP proofs.

A *proof checker* is an independent tool that verifies the correctness of proofs. There exist formally verified proof checkers that provide strong correctness guarantees (c.f., [5, 9, 14, 19]). Because these tools are inefficient, proofs are often passed through an—efficient but unverified—intermediary proof checker (such as DRAT-TRIM [22]) that transforms a DRAT proof into a so-called *LRAT proof* [5]. The resulting LRAT proof includes additional information (called *hints*), which allows a formally verified checker to efficiently check the proof.

3 Problem Overview

We want to compress proofs into small representations that can be efficiently decompressed into full proofs. Existing techniques for SAT and SMT focus on transformations and substitutions that preserve validity to generate smaller proofs [4, 18, 21]. We achieve greater compression by storing only a so-called *proof skeleton*, which itself is not a valid proof.

Tools like SLEDGEHAMMER [3] automatically solve proof obligations from interactive theorem provers, filling gaps in the proof by translating lower-level reasoning into the theorem provers’ logic. More recent work proposed a method for constructing proofs for complex SMT rewriting steps on demand in a post-processing step [17]. In a similar way, we use proof skeletons to efficiently reconstruct valid proofs that can differ from the original proofs.

Suppose you solved an unsatisfiable CNF formula ψ , and out of the many facts you learned during solving, there were three facts A , B , and C , which you deem particularly important for showing the unsatisfiability of ψ . You can then build a proof skeleton from A , B , and C . Later, you can rephrase the question $\psi \models \perp$ (“does ψ imply the empty clause?”, or equivalently, “is ψ unsatisfiable?”) into the following questions:

$$\psi \models A \quad \psi \wedge A \models B \quad \psi \wedge A \wedge B \models C \quad \psi \wedge A \wedge B \wedge C \models \perp$$

Not only do A , B , and C provide a way to partition the proof effort, when ordered carefully, they can be used as assumptions in subsequent questions. Each question can be submitted to a solver independently, and combining the four resulting proofs will give a proof of the original claim that ψ is unsatisfiable.

Our work translates this general schema to the realm of SAT by (1) determining which learned clauses from a SAT solver are most useful and should be stored in a proof skeleton; (2) carefully grouping solver calls to prevent repeated work when producing partial proofs from a proof skeleton; and (3) stitching the partial proofs together to generate a complete proof.

Determining which clauses are stored in a proof skeleton. We co-opt the clause-importance metrics used by CDCL solvers. We give a brief overview of these metrics in the following. CDCL solvers make progress by continuously learning new clauses that help them prune the search space of possible truth assignments. To limit memory usage, they occasionally perform a clause database *reduction*, removing a large portion of learned clauses based on some usefulness heuristics. Most solvers keep clauses that are short, have low *glue* value, are *reason clauses*, or have been used recently. The glue of a clause (also known as its *literal block distance*, or LBD) is a positive integer that estimates the usefulness of a clause. Intuitively, a low glue value means that few decisions are required to falsify the clause, which is considered good. For a more extensive discussion of glue, we refer to the respective literature [1]. A *reason clause* is a clause that was used by the solver when performing unit propagation, meaning that the clause became a unit clause under a partial assignment. The number of times a reason clause is *used* during conflict analysis is considered the clause’s *activity*.

Grouping solver calls for partial proofs. We leverage incremental SAT to construct partial proofs. An incremental SAT solver solves a problem with several related steps, with the solver retaining state (e.g., learned clauses and heuristics) between steps; it also allows solving under so-called *assumptions*, which are

literals assumed to be true in a step. Solving a sequence of related steps incrementally is often much faster than solving them independently of each other (for more details on incremental SAT see, e.g., [6]).

Given a formula ψ and a sequence C_1, \dots, C_n of clauses, we want to produce a DRUP proof of $\psi \models C_i$ for each $i \in 1, \dots, n$. We use an incremental solver to produce partial proofs, with each solving step corresponding to a clause C_i . For the first step, $\psi \models C_1$, we pass the assumptions $\bar{C}_1 = \bar{l}_1 \wedge \dots \wedge \bar{l}_k$ to the incremental solver. Given the formula ψ , the solver assigns the literals in the assumptions, then runs the CDCL algorithm until it derives the empty clause. During solving, CDCL guarantees that all learned clauses are RUPs for the input formula ψ . Let ϕ_1 denote the sequence of clauses learned by the solver. Then, since unit propagation under the assumptions $\bar{l}_1 \wedge \dots \wedge \bar{l}_k$ derived the empty clause, C_1 is by definition a RUP for $\psi \wedge \phi_1$. This means that C_1 can be appended to the corresponding proof of the solver (which derives all clauses in ϕ_1) to obtain a valid DRUP derivation of C_1 from ψ .

In the next step, the clause C_2 is handled similarly, except the solver retains the learned clauses $\phi_1 \wedge C_1$ when proving that C_2 is a RUP clause. This continues until all $n + 1$ steps corresponding to the n clauses of the proof skeleton are completed (step $n + 1$ corresponds to the derivation of the empty clause).

To parallelize this reasoning, we use an approach akin to *divide-and-conquer* techniques established in parallel SAT solving [13]. Divide-and-conquer solvers first partition a problem into multiple subproblems and then solve the subproblems in parallel. Similarly, we divide the incremental solver steps into so-called *chunks*, which are independent groups of subsequent solver steps. For example, we can split the solver steps into one chunk containing the first half of steps and another chunk containing the second half of steps. Both chunks can then be solved in parallel by two independent incremental SAT solvers.

Stitching partial proofs together. Once we have partial proofs for all $n + 1$ solving steps, a full proof of unsatisfiability can be constructed as the sequence of clause additions arising from $\phi_1, C_1, \phi_2, C_2, \dots, C_n, \phi_{n+1}, \perp$, where ϕ_i is the sequence of learned clauses by the i -th solver step, as explained above. In general, clauses are added and deleted during solving, so the proof can be augmented with the deletion information contained in the proofs emitted by a solver. But, we need to ensure clauses are not deleted in the proof and then implicitly reintroduced into a solver, which can occur when inprocessing techniques touch variables in the assumptions. We use *variable freezing* [7] to freeze all variables occurring in C_1, \dots, C_n ; this avoids any unsound inprocessing [8], and is required to ensure correctness of the proofs.

4 Creating Proof Skeletons

Given a clausal proof $P = \langle s_1, C_1 \rangle, \dots, \langle s_m, C_m \rangle$, we define a proof skeleton of P to be a sequence of clauses obtained from clause additions in P . Ideally, a skeleton is small but contains enough useful clauses to guide reasoning during proof

reconstruction. A proof skeleton can be constructed *online*, during the solver’s execution, by applying a filter to clauses as they are traced to a proof. Alternatively, a proof skeleton can be constructed *offline*, after solving, by processing the full proof and selecting important clauses.

4.1 Online Generation of Proof Skeletons

We create proof skeletons online by filtering clause additions as the solver traces them to a proof. Clauses that pass a usefulness threshold are added to the skeleton. As mentioned earlier, the filter applies usefulness heuristics from CDCL including *glue* and *clause activity*. Additionally, at certain intervals we add *reason clauses* to the skeleton. We implemented the filter within the solver CADICAL, giving us access to these values as well as to the reason clauses (through the *trail* of assignments). We also enabled logging, giving every clause a unique identifier, in order to sort the skeletons. We evaluate three different configurations:

- GLUE: Clauses with glue lower than 3.
- GLUE+TRAIL: Clauses with glue lower than 3, and all reason clauses on the trail before each clause-database reduction.
- DYNAMIC: Clauses with glue lower than some dynamically adjusted threshold $glue_d$, and all reason clauses on the trail every 50,000 learned clauses.

The first two configurations combine low-glue clauses with either no or some reason clauses. Increasing the glue value threshold often led to a compression of less than 1,000 times and slower reconstruction. Reason clauses are important because they are actively used by the solver whereas for low-glue clauses this is not guaranteed (although low glue is associated with high usage in general). Clause-database reductions are sparse, so reason clauses (which are added only during these reductions) will be added infrequently. We evaluate the impact of including reason clauses in the skeletons in Section 6.3.

In the first two configurations, all clauses passing the filter are accepted into the skeleton. For some formulas, a solver will produce many low-glue clauses and the skeleton will become too large, and for others too few low-glue clauses will lead to a small skeleton. Our third configuration accounts for the differences between formulas by adjusting heuristics dynamically to meet a desired compression ratio. The heuristics are updated based on the number of clauses added to the skeleton within some number of conflicts, denoted as $window_c$. For a compression ratio between 500 and 1,000, and a $window_c$ value of 5,000, we tuned the DYNAMIC configuration in the following way: every 5,000 conflicts, if more than 25 ($window_c/200$) lemmas passed the filter, the $glue_d$ value is decreased, and if less than 3 lemmas ($window_c/2,000$) passed the filter, the $glue_d$ value is increased. Reasons from the trail are added every 50,000 conflicts ($window_c \times 10$).

For configurations using reason clauses, the unique clause IDs are used to sort the skeleton. This is necessary because reason clauses are traced during reductions, so they may initially appear in the skeleton long after they were learned by the solver. During proof reconstruction it is important that clauses appear in the skeleton in an order that corresponds with a solver’s reasoning.

We implemented additional configurations using clause activities. For this, we incremented an *activity* field for each clause every time it was used during conflict analysis. An evaluation of these additional configurations is beyond the scope of this paper, but data can be found in the paper’s repository.

4.2 Offline Generation of Proof Skeletons

We create proof skeletons offline by processing a full proof and selecting the most active clauses. Given a DRAT proof, the tool DRAT-TRIM uses backwards checking to generate an optimized LRAT proof and, optionally, an UNSAT core (i.e., an unsatisfiable subset of the original formula). From the LRAT proof, we can estimate a clause’s *activity* by counting the number of times the clause appears in a hint of a clause-addition step. We then add the clauses with the highest activity to the skeleton until a target compression ratio is met. We found for most problems the target 1,000 provided optimal reconstruction performance. We sort the skeleton by each clause’s first use as a hint in the LRAT proof, signifying when a clause is actually used as opposed to when it is learned. We evaluate three configurations for offline generation:

- OFFLINE: Select 1,000 times fewer clauses than in the original DRAT proof.
- OFFLINE+UNITS: Additionally include all unit clauses from the proof.
- OFFLINE-OPT: Select 1,000 times fewer clauses than in the optimized LRAT proof.

The motivation for OFFLINE-OPT is that some optimized LRAT proofs have significantly fewer clauses than the DRAT proofs, resulting from many unused lemmas, which suggests that stronger compression is possible.

Offline construction requires expensive post-processing with DRAT-TRIM. However, during online construction we can only *guess* the future usefulness of clauses when they are derived, by relying on heuristics such as glue, but we cannot know how often a clause will actually be used. For instance, it may be that a clause has low glue (predicting high usefulness) but is learned and then never used in the rest of the proof, making it worthless in the skeleton. In contrast, when constructing a skeleton offline—after solving—we know already how often the clause was actually used in reasoning throughout the proof, and whether it was used to derive the empty clause. Also, we can use the UNSAT core instead of the original formula when reconstructing a proof for the original problem.

5 Reconstructing Proofs from Skeletons

We reconstruct proofs by filling the gaps of a proof skeleton with a SAT solver. Once we have proofs for all gaps, we stitch them together with the clauses of the skeleton to create a complete proof. We can utilize information obtained during proof reconstruction to further shrink skeletons by removing less useful clauses. Finally, we can also use a skeleton to create an optimized LRAT proof.

Proof	Skeleton	Reconstruction	Incremental Reconstruction
C_1	C_2	$\psi \models C_2$	$\psi \models C_2 : \phi_1$
C_2	C_5	C_2	C_2
C_3	\vdots	$\psi \wedge C_2 \models C_5$	$\psi \wedge C_2 \wedge \phi_1 \models C_5 : \phi_2$
C_4	\vdots	C_5	C_5
C_5	\vdots	\vdots	\vdots
\vdots	\vdots	$\psi \wedge \textit{Skeleton} \models \perp$	$\psi \wedge \textit{Skeleton} \wedge \phi \models \perp$

Fig. 1. Proof reconstruction from a proof skeleton and a formula ϕ by filling in the gaps between skeleton clauses. This can be done with independent SAT calls or with an incremental SAT solver that keeps learned clauses (ϕ_i) between steps.

5.1 Filling Skeletons Using Incremental Solvers

We consider two ways of filling a proof skeleton’s gaps—*reconstruction* and *incremental reconstruction*; both are illustrated in Fig. 1. Given a formula ϕ and a skeleton C_1, \dots, C_n , reconstruction fills each gap $\psi \wedge C_1 \wedge \dots \wedge C_{i-1} \models C_i$ using independent SAT solver calls, with $\psi_1 \wedge C_1 \wedge \dots \wedge C_n \models \perp$ as the final call. Filling a gap for $C_i = (l_1 \vee \dots \vee l_k)$ involves assuming $\bar{l}_1 \wedge \dots \wedge \bar{l}_k$ and deriving the empty clause with proof ϕ , which proves that C_i is a RUP for $\psi \wedge C_1 \wedge \dots \wedge C_{i-1} \wedge \phi$. Each gap has an associated DRUP proof ϕ_i emitted by the solver. Since RUP is a monotonic property, the clauses added in ϕ_i will not affect the validity of ϕ_j for $i < j$. However, clause deletions could make the proof $\phi_1, \langle \mathbf{a}, C_1 \rangle, \phi_2, \langle \mathbf{a}, C_2 \rangle, \dots, \langle \mathbf{a}, C_n \rangle, \phi_{n+1}, \perp$ incorrect. For example, if a skeleton clause C_1 is deleted in ϕ_2 , then ϕ_3 (stemming from $\psi \wedge C_1 \wedge C_2 \models C_3$) may use C_2 —a clause already deleted in the proof. The same problem could occur if formula clauses are deleted. Therefore, we must remove any deletion steps for clauses of the skeleton or of the formula clauses from each ϕ_i .

The second approach, *incremental reconstruction*, uses an incremental SAT solver, which allows the use of learned clauses when filling subsequent gaps. Specifically, we create an incremental problem with the steps $\textit{assume}(\bar{C}_1), \dots, \textit{assume}(\bar{C}_n), \textit{assume}(\emptyset)$, where each step $\textit{assume}(\bar{C}_i)$, with $C_i = (l_1 \vee \dots \vee l_k)$, involves assuming $\bar{l}_1 \wedge \dots \wedge \bar{l}_k$ and deriving the empty clause. Each step produces a proof ϕ_i , and the complete proof $\phi_1, \langle \mathbf{a}, C_1 \rangle, \phi_2, \langle \mathbf{a}, C_2 \rangle, \dots, \langle \mathbf{a}, C_n \rangle, \phi_{n+1}, \langle \mathbf{a}, \perp \rangle$ is correct as long as variables occurring in skeleton clauses are frozen (as described in Section 3). With this approach, we no longer need to worry about deletions of skeleton clauses or formula clauses because the solver fills each gap using the current clause database, i.e., each gap is proved without clauses formerly deleted by the solver.

To parallelize incremental reconstruction, we partition the incremental problem into several independent incremental problems, which we call *chunks*. We assign k clauses C_l, \dots, C_{l+k-1} from the skeleton to each chunk, and we then use an incremental solver to compute partial proofs for each of the clauses, starting

from the formula $\psi \wedge C_1 \wedge \dots \wedge C_{l-1}$. For each partial proof corresponding to a clause C_i , we call the solver with the assumptions negating the clause, i.e., with $assume(\bar{C}_i)$. Again, we must remove any deletion steps of skeleton clauses or formula clauses since they may be used in later chunks. All added clauses are then RUPs, and so the concatenation of chunk proofs is a complete proof.

Each chunk can be solved independently in parallel. The more skeleton clauses in each chunk, the more clauses the incremental solver can learn and reuse in subsequent steps. However, gaps might differ in hardness, meaning that some gaps can be filled quickly while others require a significant amount of solving time. A chunk can thus become a bottleneck during parallelization if it includes many difficult gaps. In our evaluation, we partitioned the skeleton into chunks of equal size, one for each core. For instance, on a single core, one incremental problem spanning the entire skeleton was given to a solver instance whereas for 24 cores, the skeleton was partitioned into 24 chunks. In principle, we could partition a skeleton into more chunks than cores, but this would require an intermediary level of problem scheduling that we leave for future work.

5.2 Shrinking Skeletons

The runtimes for filling each gap of a proof skeleton could provide insight into the usefulness of the skeleton clauses. For example, if the solver can quickly fill a gap, the corresponding skeleton clause may be trivially implied, and if the solver takes long, the clause may be useful since its derivation requires a lot of reasoning. Alternatively, the difference in runtime might not be explained by clause usefulness. Take, for example, the two gaps $\psi \models C_2$ and $\psi \wedge C_2 \models C_5$ from Fig. 1, and assume that the solver fills the first gap in a millisecond and the second gap in ten seconds. If the difference is a result of C_2 being trivially implied, it makes sense to remove C_2 from the skeleton; otherwise, if the difference is due to factors unrelated to usefulness, it is better to remove C_5 . Based on this observation, we try to shrink a given skeleton by sorting gap reconstruction times and removing a certain share of the slowest or fastest clauses.

Our empirical evaluation in Section 6 indicates that removing the fastest clauses is the right approach for improving compression and (sometimes) reducing reconstruction time. Even though gap runtime and clause usefulness are correlated, the correlation is not perfect. For instance, sometimes the incremental solver is able to quickly fill a gap because of learning from previous steps of the incremental problem. Even if it takes a long time to fill a gap, there is no guarantee that the corresponding skeleton clause is useful for filling future gaps. We examine in detail how shrinking skeletons affects reconstruction time.

5.3 Reconstructing LRAT Proofs from Skeletons

The proof reconstruction described above will produce DRAT proofs. Formally verified checkers typically require LRAT proofs, forcing a conversion via a proof checker such as DRAT-TRIM, which can take much longer than the original

solving time. Instead, we can reconstruct DRAT proofs for each chunk, then convert the DRAT proofs to LRAT in parallel, and finally concatenate them.

We use DRAT-TRIM to convert chunk DRAT proofs to LRAT. This required us to modify DRAT-TRIM (e.g., by changing the way it performs backwards checking, and how it handles unit clauses). By default, DRAT-TRIM starts backwards checking at the empty clause. But, only the last chunk will derive the empty clause, and further, we must ensure all skeleton clauses are included in the backwards check, as they may be used in later chunks. To account for this, we mark each skeleton clause in the DRAT proof before performing the backwards check. The backwards check verifies that each marked clause is RAT (or RUP, in our case), including the clauses in the LRAT proof. When combining the chunk LRAT proofs, we map the skeleton clauses in each chunk to the index of the LRAT step where they were initially added. Finally, we remove all deletions from the LRAT proof, but this will not affect proof-checking time, mainly since LRAT checkers perform unit propagation in linear time using hints. While the following evaluation focuses on DRAT proof reconstruction from skeletons, we tested our implementation of parallel LRAT proof reconstruction on 24 cores, and verified several proofs with CAKE-LPR [19].

6 Experimental Evaluation

We evaluated our approach on SAT competition 2021 Main Track benchmarks, using all (65) unsatisfiable formulas that were solved between 500 and 5,000 seconds by the solver CADICAL [2]. By requiring at least 500 seconds of solving time, we ensured that proofs are of reasonable size (around 1 GB) and therefore good candidates for compression. We ran experiments on an AWS EC2 m5d.metal instance, with 96 virtual CPUs and 500 GB of memory, running at most 24 parallel processes at a time. We used a timeout of 5,000 seconds for solving a problem and constructing a DRAT proof. For proof reconstruction on a single core we used a single incremental problem spanning the entire skeleton. For proof reconstruction on 24 cores, we evenly divided the proof skeleton into 24 incremental problems (chunks) passed to 24 instances of CADICAL. We report real time for proof reconstruction, not including skeleton extraction.

6.1 Single-Core Proof Reconstruction

Fig. 2 shows the best configurations on each formula using online skeletons (left) and offline skeletons (right), for the single-core experiments (i.e., the entire skeleton on a single core). Almost all proofs were reconstructed faster than the original solving time (below the red dotted line), and in some cases more than five times faster (below the blue dotted line). Each configuration was the best for some formulas. The GLUE configuration led the online skeletons. With a single incremental problem, learned clauses from earlier incremental calls can be kept for the entire execution, meaning that clauses that occur later in large skeletons (e.g., GLUE+TRAIL) may be trivially implied by previously learned clauses.

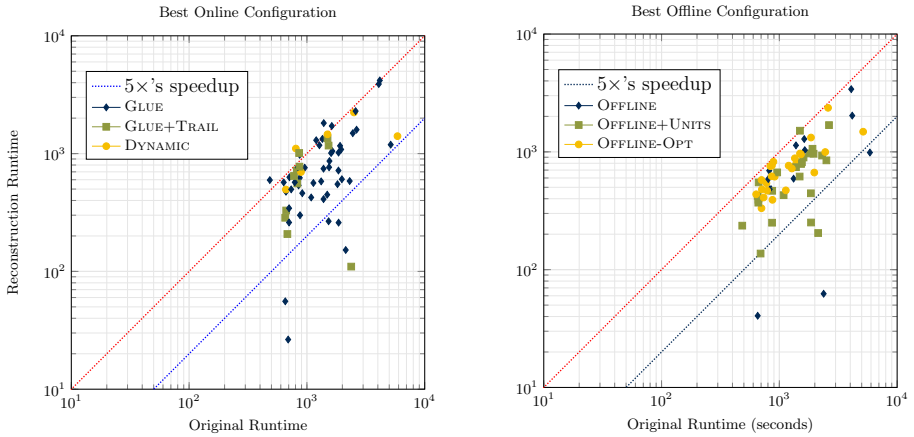


Fig. 2. Runtimes (in seconds) of best online (left) and offline (right) configurations for proof reconstruction using a proof skeleton and a single core.

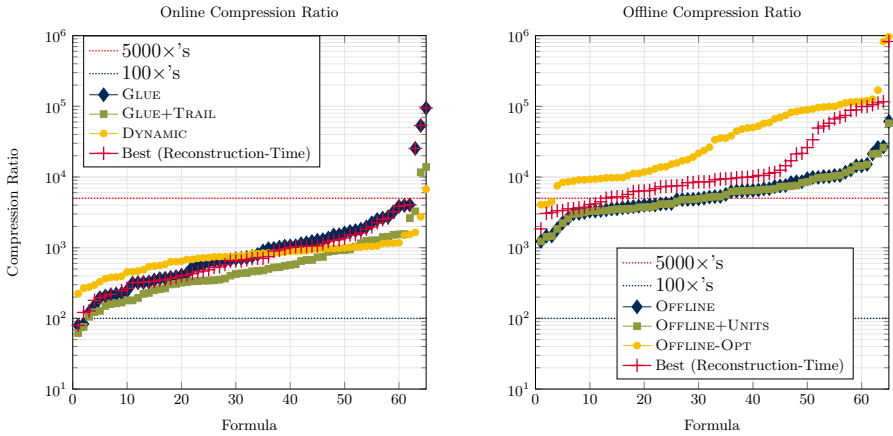


Fig. 3. Proof skeleton compression ratio for online (left) and offline (right).

6.2 Skeleton Compression Ratio

Fig. 3 shows the sorted compression ratios (w.r.t. file size) between proof skeletons and the original DRAT proofs for each configuration as well as the compression ratios for the configuration with the fastest reconstruction time on each formula (Best). For online configurations (left), the DYNAMIC skeletons have the most consistent compression ratios, with a tradeoff in reconstruction times. In some cases, skeletons can have higher compression (10,000 times) without a loss in performance, witnessed by the right-hand-side tail of the plot.

For offline configurations (right), OFFLINE selects 1/1,000 of the clauses from the original DRAT proof. The ratios are much greater than 1,000 because skele-

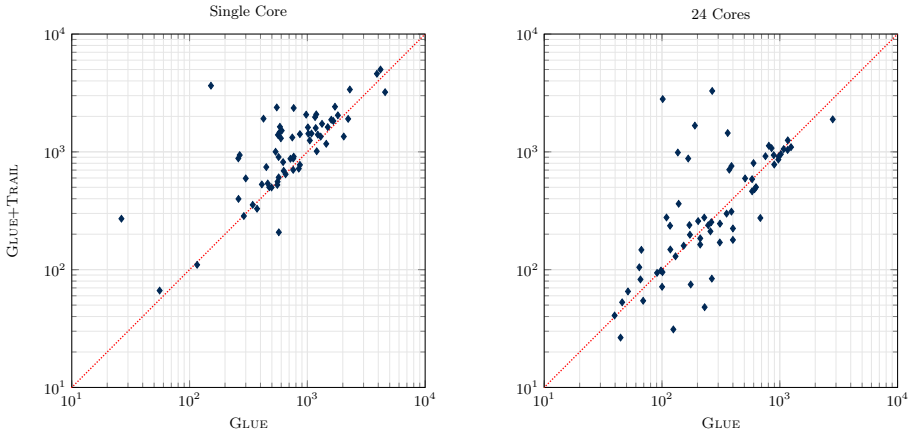


Fig. 4. Runtimes (in seconds) for proof reconstruction of multiple online configurations with a single core (left) and 24 cores (right).

tons have no deletion information and the most active clauses are typically much shorter than the average clause. OFFLINE-OPT provides around a factor 10 more compression, and these smaller skeletons provide faster reconstruction for about half of the formulas. In general, the compression is much better when using *clause activity* as a measure for clause importance as opposed to online heuristics (such as *glue*), with similar reconstruction times seen in Fig. 2.

6.3 Impact of Reason Clauses in Online Skeletons

Fig. 4 shows a comparison of reconstruction times between the GLUE and the GLUE+TRAIL online configurations, both on a single core (left) and on 24 cores (right). On a single core, creating skeletons with only low-glue clauses performs better than creating skeletons with low-glue clauses *and* reasons from the trail. On multiple cores, however, the reason clauses are beneficial for many reconstructions. This may be because for parallel reconstruction, each individual chunk only has access to lemmas earlier in the skeleton during solving. Therefore, having more clauses in the skeleton will aid the later chunks. In contrast, for a single chunk on one core, learned clauses are kept throughout solving, and these learned clauses supplement the smaller skeletons.

6.4 Impact of the UNSAT Core on Offline Skeletons

Fig. 5 shows the effect of using an UNSAT core during reconstruction for offline skeletons on a single core (left) and on 24 cores (right). For the experiments using an UNSAT core, we remove formula clauses that are not in the UNSAT core before passing the formula to the solver during the incremental SAT call for the chunk proof. Using the UNSAT core greatly improves performance during

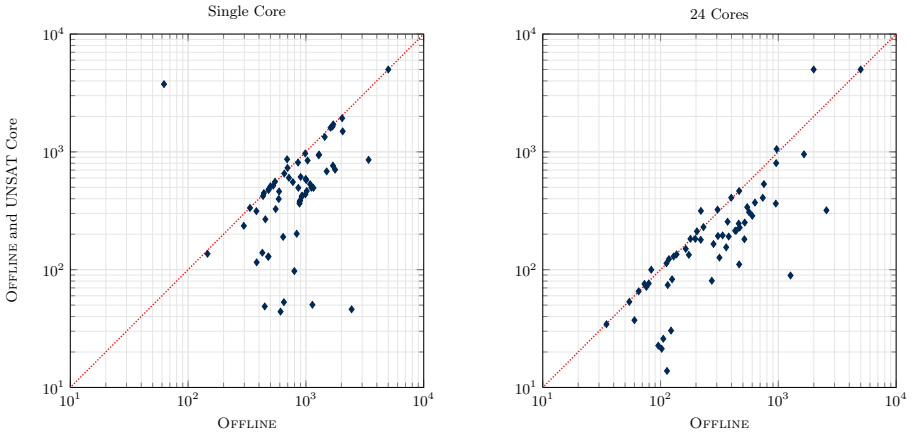


Fig. 5. Runtimes (in seconds) for OFFLINE proof reconstruction with and without an UNSAT core with a single core (left) and 24 cores (right).

reconstruction on a single core. This may be because the skeleton is built from reasoning based on the UNSAT core, so focusing the solver on these specific formula clauses makes filling the gaps in the skeleton easier. The UNSAT core is useful in parallel reconstruction as well, producing the overall best configuration between online and offline skeletons. To give an idea, it takes approximately 125 KB to store an UNSAT core as a bit vector (each bit indicating whether or not a clause is part of the core) for a formula with one million clauses. For most formulas, this data would be dominated by the size of the proof skeleton.

6.5 Skeleton Shrinking after Reconstruction

We discussed in Section 5.2 that it might make sense to shrink a skeleton by removing some amount of the fastest or of the slowest skeleton clauses. Fig. 6 shows results for reconstruction on 24 cores using the online skeleton, removing either the fastest 90% or the slowest 10% of clauses. To perform the shrinking, we performed proof reconstruction from the skeleton and measured the solve times for the incremental calls, with each call corresponding to a skeleton clause. Removing the fastest 90% has a small impact on reconstruction time, performing slower for the majority of formulas. In some cases, shrinking the skeleton even improves performance because redundant or unnecessary clauses are removed from the skeleton. Removing the slowest solved clauses causes a wider variation in reconstruction time. This might be because these clauses are important for guiding the solver during reconstruction, and sometimes they lead the solver into unprofitable search regions that waste time. This shows two things: (1) For some formulas, removing only a fraction of clauses from the skeleton can lead to a big or small improvement, and (2) skeleton clauses are mostly nontrivial and cannot be added or removed randomly without a potentially consequential impact.

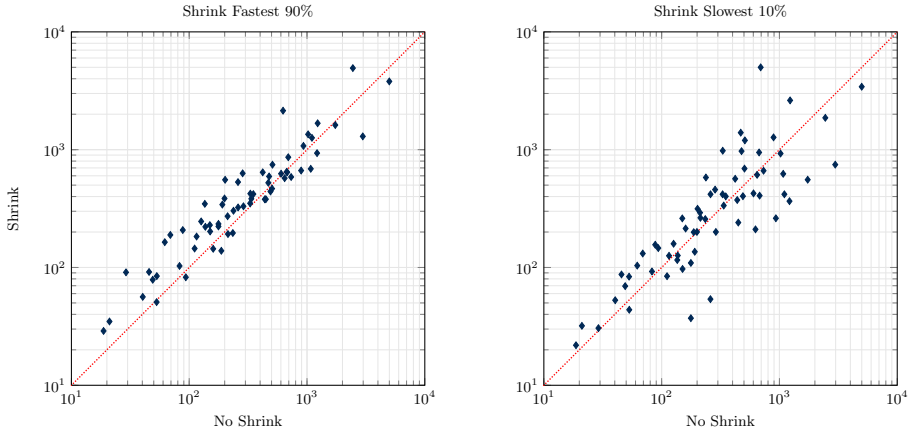


Fig. 6. Runtimes (in seconds) of proof reconstruction on 24 cores after skeleton shrinking for the DYNAMIC online configuration, removing the fastest 90% (left) or the slowest 10% (right) of clauses from the skeleton.

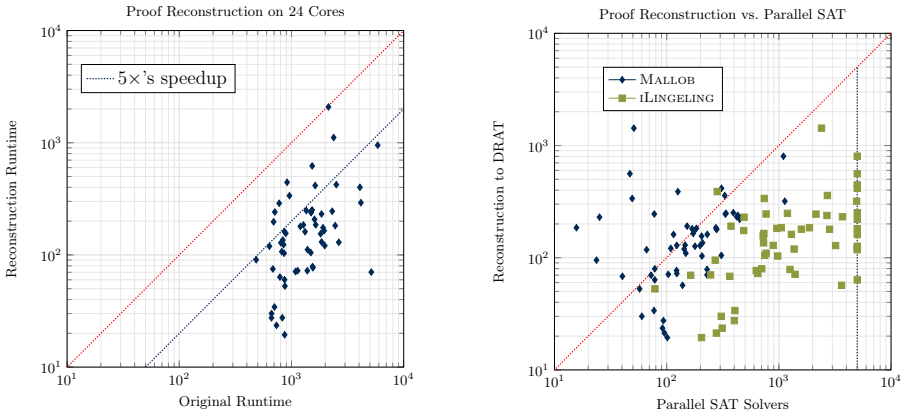


Fig. 7. Left: Runtimes (in seconds) of original solver on a single core against proof reconstruction on 24 cores with the best offline-skeleton configurations OFFLINE+UNITS using UNSAT cores. Right: Runtimes (in seconds) of parallel SAT solvers MALLOB and LINGELING without proof logging against proof reconstruction with the best offline skeleton configurations using an UNSAT core, each using 24 cores.

6.6 Comparison With Sequential and Parallel SAT Solvers

Alternatives to our proof reconstruction could be to compute a proof on demand by solving a formula from scratch (either with a sequential or with a parallel SAT solver) or to run a parallel incremental solver that fills the gaps of a skeleton.

The left plot of Fig. 7 shows the difference between running a sequential solver on a single core versus running our parallel proof reconstruction on 24 cores. For

the majority of formulas, parallel proof reconstruction is over five times faster, and in some cases closer to ten times faster. One formula had little improvement for reconstruction (on the red dotted line). For this formula, the final chunk took around 2,000 seconds to solve, and the next slowest chunk took only 24 seconds, meaning the hardest gaps were all clustered in the final chunk. For these sorts of problems, a smaller chunk size could break up the hard gaps, therefore improving utilization across cores and reducing the reconstruction time.

To our knowledge, there exist no portfolio solvers or parallel incremental solvers that produce proofs. However, it might be possible to add proof support to solvers like MALLOB (a clause-sharing portfolio solver) or ILINGELING (a parallel incremental solver); we thus compare our approach to these solvers in the right plot of Fig. 7.

The comparison to MALLOB suggests that some form of clause sharing between solvers that solve independent chunks may improve performance. This could be achieved with *forward clause sharing*, where learned clauses can only be sent to solvers running on subsequent chunks. Also, MALLOB has full core utilization by running each solver until one derives the empty clause, but our proof reconstruction does not since some chunks take longer than others. With smaller chunk sizes and good scheduling, proof reconstruction could get closer to full utilization.

ILINGELING, which is based on LINGELING [2], takes an incremental problem and greedily assigns steps to solver instances, terminating when one instance derives the empty clause. There is no clause sharing between solvers. We ran ILINGELING using the incremental problem derived from the proof skeleton. In proof reconstruction, chunks can use skeleton clauses from previous chunks, leading to consistently better performance than ILINGELING.

7 Conclusion

We presented a semantic approach for compressing propositional proofs by selecting important clauses that summarize the reasoning of a solver. We store these clauses in a so-called proof skeleton, from which we can reconstruct a complete proof in parallel by performing multiple incremental SAT solver calls. We implemented our approach on top of the SAT solver CADICAL and the proof checker DRAT-TRIM. In an empirical evaluation, we showed that our approach can produce skeletons that are 100 to 5,000 times smaller than the original proofs. On a single core, almost all proofs were reconstructed faster than the original solving time, and when using 24 cores, the majority of proofs was reconstructed around five times faster. This is significant since proof checking typically takes longer than solving, and since existing parallel solvers cannot produce proofs while maintaining strong performance. We observed that proof skeletons not only serve as a compression mechanism but also provide insight into a problem. In future work, we thus plan to explore the connection between skeletons, proofs, and solver performance.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 399–404 (2009), <http://ijcai.org/Proceedings/09/Papers/074.pdf>
2. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froykys, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
3. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. *J. Autom. Reason.* **51**(1), 109–128 (2013)
4. Boudou, J., Fellner, A., Paleo, B.W.: Skeptik: A proof compression system. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8562, pp. 374–380. Springer (2014), https://doi.org/10.1007/978-3-319-08587-6_29
5. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (2017), https://doi.org/10.1007/978-3-319-63046-5_14
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003), https://doi.org/10.1007/978-3-540-24605-3_37
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. *Electron. Notes Theor. Comput. Sci.* **89**(4), 543–560 (2003), [https://doi.org/10.1016/S1571-0661\(05\)82542-3](https://doi.org/10.1016/S1571-0661(05)82542-3)
8. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 136–154. Springer (2019), https://doi.org/10.1007/978-3-030-24258-9_9
9. Heule, M., Jr., W.A.H., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 269–284. Springer (2017), https://doi.org/10.1007/978-3-319-66107-0_18
10. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **abs/1610.06229** (2016), <http://arxiv.org/abs/1610.06229>
11. Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18). pp. 6598–6606. AAAI Press (2018), <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952>

12. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: Creignou, N., Le Berre, D. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2016*. pp. 228–245. Springer International Publishing, Cham (2016)
13. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) *Hardware and Software: Verification and Testing*. pp. 50–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
14. Lammich, P.: Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.* **64**(3), 513–532 (2020), <https://doi.org/10.1007/s10817-019-09525-z>
15. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers* **48**(5), 506–521 (1999), <https://doi.org/10.1109/12.769433>
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. pp. 530–535. ACM (2001), <https://doi.org/10.1145/378239.379017>
17. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) *Formal Methods in Computer-Aided Design - 22nd Conference, FMCAD 2022, Trento, Italy, October 17-21, 2022, Proceedings*. pp. 65–74. *Formal Methods in Computer-Aided Design*, TU Wien Academic Press (2022)
18. Rollini, S.F., Bruttomesso, R., Sharygina, N., Tsitovich, A.: Resolution proof transformation for compression and interpolation. *Formal Methods Syst. Des.* **45**(1), 1–41 (2014), <https://doi.org/10.1007/s10703-014-0208-x>
19. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Groote, J.F., Larsen, K.G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 12652, pp. 223–241. Springer (2021), https://doi.org/10.1007/978-3-030-72013-1_12
20. Van Gelder, A.: 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), http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf
21. Vyskocil, J., Stanovský, D., Urban, J.: Automated proof compression by invention of new definitions. In: Clarke, E.M., Voronkov, A. (eds.) *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 6355, pp. 447–462. Springer (2010), https://doi.org/10.1007/978-3-642-17511-4_25
22. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: *Theory and Applications of Satisfiability Testing (SAT)*. LNCS, vol. 8561, pp. 422–429 (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

