Efficient Proof Checking with LRAT in CaDiCaL (Work in Progress)

Florian Pollitt, Mathias Fleury, and Armin Biere
pollittf@informatik.uni-freiburg.de, fleury@cs.uni-freiburg.de, biere@cs.uni-freiburg.de
University of Freiburg, Germany

Abstract

The proof format DRAT used in the SAT competition is rather inefficient to check, often even slower to check than it takes the SAT solver to solve the instance and generate the proof. Therefore we implemented within the SAT solver CaDiCaL the possibility to generate LRAT proofs directly, where LRAT on the one hand is much easier and way more efficient to check, but on the other hand much harder to generate. Unlike previous approaches our implementation generates LRAT directly in the solver without intermediate translations. We further propose the tool LRAT-Trim, which can trim redundant proof steps from LRAT proofs, which not only reduces proof size but also leads to much faster proof checking.

1 Introduction

Proof checking is an important part of SAT solving, e.g., unsatisfiable problems do not count as solved in the SAT Competition unless a proof is provided which passes a proof checker. To increase trust even further proof checkers are used which are entirely verified [5, 7].

The currently only allowed proof format in the SAT Competition is DRAT [8].[1] The main issue with DRAT is that checking can take several times the amount of solving time. The reason is that the DRAT proof certificate format favors ease of generation and is not detailed enough to avoid searching during checking. Therefore both the solver and the checker have to propagate clauses (actually using similar data structures). To reduce this overhead (and simplify verification) all verified checkers follow the same principle. First the DRAT proof is converted by an (untrusted) external program to a more detailed proof format such as LRAT [5] or GRAT [7]. The resulting proof in an enriched format – containing enough details to avoid search – is checked by the verified program instead.

Note, however, that neither our SAT solver CaDiCaL [3] nor the winners of the SAT Competition of the last 2 years need the full power of RAT. They provide proofs in the upward-compatible but less powerful DRUP proof format. On the other hand CaDiCaL contains many different inprocessing techniques, which makes it a good candidate to implement direct generation of LRAT proofs – even though some of these techniques are not activated by default.

A similar attempt to resolve the performance issues with DRAT [1] lead to a new proof format, FRAT, that sits between LRAT (because it allows for justifications) and DRAT (because it still allows steps without justification). Their aim was to fill out most *gaps* and leave the *"harder"* cases as black box to be filled in by the proof checker. Therefore, they still use a tool Frat-Rs to convert the FRAT proof to a proper LRAT proof – trimming the proof on the way, to reduce the number of proof steps to check.

In this work-in-progress, we have extended our SAT solver CaDiCaL [3] to generate the richer LRAT format directly. The focus of our work is on three different aspects:

C1. directly produce correct LRAT proofs

C2. without slowing down the solver and

C3. without changing its search space.

Our first goal **C1** lead us to reimplement LRAT generation during conflict analysis and as part of all inprocessing techniques of CaDiCaL, some of which were not covered in the FRAT implementation, such as equivalent literal substitution (Section 3).

As it is common, our implementation generates a vast number of proof steps on-the-fly, from which however at the end a significant fraction turns out to be unnecessary to derive the empty clause $\perp$. This applies to most tools that process DRAT or FRAT which accordingly can benefit from some form of trimming. Therefore, we also implemented a tool called Lrat-Trim to trim proofs down and improve performance of checking proofs with the verified checker Cake_Lpr (Section 4).

At this point we can not report on extensive experiments yet, and therefore we focus in this work-in-progress report on making our proof generation robust and will discuss preliminary results for a problem with a very large large proof from the SAT Competition 2022. (Section 5).

Our implementation is publicly available[2] and is going to be merged into the main CaDiCaL version.

2 Preliminaries

For a detailed introduction to SAT solving, we refer to the *Handbook of Satisfiability* [4]. For the purpose of this paper, it is sufficient to know that SAT solvers build a partial

[1] A change was announced in the SAT Competition 2023, as different proof checkers (and therefore different proof format) could be allowed.

[2] `https://github.com/florianpollitt/radical`

```
p cnf 2 4
1 2 0
1 -2 0
-1 2 0
-1 -2 0
```

(a) DIMACS input

```
1 0
d 1 2 0
d 1 -2 0
2 0
d -1 2 0
0
```

(b) DRAT proof

```
o 1 1 2 0
o 2 1 -2 0
o 3 -1 2 0
o 4 -1 -2 0
a 5 1 0 1 1 2 0
d 1 1 2 0
d 2 1 -2 0
a 6 2 0
d 3 -1 2 0
a 7 0 1 5 6 4 0
f 4 -1 -2 0
f 5 1 0
f 6 2 0
f 7 0
```

(c) FRAT proof

```
5 1 0 1 2 0
5 d 1 2 0
6 2 0 5 3 0
6 d 3 0
7 0 5 6 4 0
```

(d) LRAT proof

Figure 1 Example DRAT, FRAT, and LRAT proofs for the same CNF on the left.

model. Along the way, they learn new clauses preserving satisfiability until either the partial model becomes a total model (translating the model back to a model of the original clause set) or the empty clause $\bot$ is derived, meaning that the problem is unsatisfiable.

The DRAT proof format consists simply of all the clauses learned by the SAT solver. This design decision helps DRAT to easily capture all techniques currently used by SAT solvers without the need to provide justification. The LRAT proof format provides more detailed information: Each clause gets an identifier and each step is the result of resolving several clauses together. The list of clauses is given as justification for each step. The last derived clause is $\bot$ – showing that the problem is unsatisfiable.

In related work [1] a new proof format was proposed that sits in-between LRAT and FRAT: some justification can be left out. This reduces the amount of implementation work in the SAT solver, since certain types of functions can be left unchanged, particularly if they infrequently contribute to the proof. Confusingly, the option to activate this proof is called `--lrat`. Throughout the rest of the paper, we will call this implementation (CaDiCaL) FRAT, even when we talk specifically about the generation of LRAT steps, because we have nothing to say about the other steps that are simply unchanged DRAT steps.

Figure 1 illustrates these proof formats for a simple example. The shortest proof is obviously the DRAT proof 1b, but it is missing information on how clauses were derived (in dark blue). In FRAT we have to repeat all the input assumptions, starting with o. The justification steps are also optional (see `a 6 2 0` without justification).

3 Implementation

Most of the actual computation can be done alongside the generation of clauses, including clause learning, which by definition consists of resolving clauses in the order given by the partial model (Section 3.1). However, some techniques require a deeper change like equivalence literal substitution (Section 3.2).

3.1 Conflict Analysis

Most clauses derived by a SAT solver originate from conflict analysis. When the solver finds a mismatch between the current partial assignment and the clauses, one conflicting clause is analyzed and the partial assignment adjusted.

One recent addition to the conflict analysis is based on the concept of “shrinking” [6]. The idea in shrinking is to derive the first unique implication point [4] on each level without increasing the proof size. This is very useful for problems with many binary clauses, such as the planning instances from the SAT Competition 2020.

Unlike FRAT proof generation for minimization, our implementation perfoms a post-processing step direclty on the learned clause instead of repropagation. To this extent we identify literals that were removed or added and add the necessary reason clauses as needed:

```
C_old := Clause before shrinking
C_new := Clause after shrinking
Chain_old := LRAT chain for C_old
Chain_new := empty LRAT chain

calculate_lrat_chain (literal K)
  C := reason of K in the current assignment
  foreach literal L in C different from K
     if not (reason of L in Chain_new and
             reason of L in Chain_old) and
        L not in C_new)
       calculate_lrat_chain (L)

  add C to Chain_new

for each literal L in C_old
  if L is not in C_new
    calculate_lrat_chain (L)

Chain_new := Chain_new + Chain_old
```

This works both for the standard minimization which is actually used inside shrinking as well as shrinking.

3.2 Equivalence Literal Substitution

Equivalent literal substitution is a procedure that detects and replaces equivalent literals by a chosen representative. For example, if the problem includes the clauses $\neg A \vee B$ and $A \vee \neg B$, we know that A and B are equivalent and we can replace all occurrences of either literal by the other.

We use Tarjan's algorithm to detect cycles in the graph spanned by the binary clauses and then fix a representative for each cycle. In the DRAT proof we simply dump all changed clauses and delete the old ones.

For LRAT we have to produce the resolution chain. This can only be calculated after fixing the representative and is done for each replacement in every clause separately, similarly to the process described in Section 3.1.

Fixing the representative is a rather arbitrary choice (smallest absolute value). We considered changing this to first visited by Tarjan's algorithm. This change would allow us to reuse some computation possibly making the generation of LRAT more efficient. In the end we decided against it to keep the behavior of CADICAL the same.

4 Trimming LRAT proofs

In early experiments we observed that the FRAT tool chain produced significantly smaller proofs, allowing for much more efficient proof checking. This is because clauses which are not required to derive the empty clause are trimmed from the proof and do not have to be checked nor at the end verified. An important feature of proof checking is the ability to trim down the proof which helps to reduce this redundant checking.

Even though trimming is very effective it is not obvious how to achieve this reduction in DRAT because dependencies between proof steps are missing. In LRAT these dependencies are listed explicitly and we implemented a proof trimmer called LRAT-TRIM to make use of this fact. It allows us to regularly achieve a reduction by a factor of 2 to 3 also again emphasizing how many useless clauses a SAT solver actually derives during search.

In essence, trimming is about doing a backward liveness analysis skipping clauses which are not useful. However, it is not possible to write a file backwards, so we only iterate over the graph starting from the $\perp$ clause at the end.

```
mark_antecedents(clause C)
   if C is marked return
   if C is an original clause return
   mark C as used
   for each antecedent D of C
     mark_antecedents (D)
```

Once the algorithm has identified all the useful proof steps, we can dump the proofs back to a file. One step we have not experimented with is the deletion of clauses: Studying whether it is better to immediately delete clauses or wait and delete several clauses at once is left as future work. However, we observed that eagerly deallocating removed clauses unfortunately does not improve performance, but it does reduce maximum memory usage substantially.

5 Early Experiments

After implementing LRAT production in our SAT solver CADICAL we first identified a minor necessary change (Section 5.1) that has no major impact. Besides fuzzing we have also tested our approach on input files containing unit clauses, which was not supported by FRAT (Section 5.2). Finally we report on the performance difference for a single problem with a very large proof (Section 5.3).

5.1 No Behavior Difference

During our experiments to validate goal **C3**, we realized that we had to change solver behavior in two ways. First, scheduling of garbage collection during bounded-variable elimination depends on the size of the clauses, which however changed with LRAT proof generation, as clauses became larger due to the additional required clause identifier `id` field. Our new version of CADICAL thus is always forced to use clause identifiers which however we do not consider to have a substantial impact on performance no memory usage. The second change became necessary due to the way how conflicts were derived in equivalence literal detection: instead of stopping on detection of such a conflict, we now simply continue and later propagate the literal in order to produce a proper LRAT proof.

5.2 Robustness by Fuzzing

Our goal **C1** of always being able to generate proofs was achieved by intensive fuzzing of our solver, proof generation and proof checking. We first attempted to do the same with the old implementation, but immediately experienced failing proofs, due to several reasons, including handling of unit clauses in the input proof file.

We also observed that resolution chains often listed the same clause several times. Reducing these occurrences can lead to a polynomial speedup, since justifying one literal can pull in several more clauses (e.g., if some of the literals have been removed by minimization).

5.3 Performance Loss

We have not yet studied the goal **C2** much. Early experiments indicate that our solver is slightly slower but that solving and proof checking is significantly faster. As example we consider the problem `sudoku-N30-10` from the SAT Competition 2022 [2]. To make the times comparable, we activated clause identifiers in the basic CADICAL version, ported the modification from the FRAT version to the newest CADICAL 1.5.2. The only real algorithmic difference is our shrinking of learned clauses [6], which we deactivated for the comparison instead of fixing the FRAT proof generation. All runs are without shrinking (as it is not supported by the FRAT version). Under these restrictions the comparison is fair, as runs produce exactly the same search behavior, including the same exact number of conflicts. The experiments were run on a desktop computer with an Intel i9-12900 with 128 GB RAM and hyperthreading on, except for the GRAT generation which was run on a machine with 2 TB because 128 GB were not enough.

CaDiCaL	Solving time	Proof size	Conversion tool	Conversion+ trimming	Trimmed proof size	Checking tool	Verified checking	Total
no proofs	4 770 s	-	-	-	-	-	-	4 770 s
DRAT	4 801 s	21 GB	DRAT-TRIM	5 639 s	13 GB	CAKE_LPR	812 s	11 252 s
DRAT	4 801 s	21 GB	GRAT (64 threads)	916 s	13 GB	GRATCHK	326 s	6 043 s
FRAT	5 349 s	78 GB	FRAT-RS	1 907 s	23 GB	CAKE_LPR	900 s	8 156 s
LRAT*	5 100 s	70 GB	-	-	-	CAKE_LPR	3 819 s	8 919 s
LRAT*	5 100 s	70 GB	LRAT-TRIM*	263 s	18 GB	CAKE_LPR	900 s	6 263 s

Table 1 Timing with different workflows on the `sudoku-N30-10` from the SAT Competition 2022. The $\star$ symbol indicates that the tool is a contribution of this work.

Table 1 provides the detailed timings. The DRAT proof conversion needs slightly more time than the proof production. LRAT proof generation has a cost of around 7%, but the proof checking is heavily reduced. Trimming the proof has a cost of 263 s, but reduces the checking time by a factor 3 and the proof size by a factor 4.

One reason explaining that directly produced LRAT proofs are larger than translated proofs comes from a heuristic of the translation tools: If they have the choice between two clauses, they will pick the clause that has already been used, while our proof production will pick the one used internally to derive a clause.

6 Conclusion

We have implemented LRAT proof production in our SAT solver CaDiCaL. Early experiments show that performance is slightly reduced, but the full workflow of producing *and* checking the proof becomes much faster thanks to our other tool LRAT-TRIM.

Future work includes a proper evaluation of the implementation on the entire set of problems of the SAT competition. Another interesting idea is to check the proofs online, directly while generated.

7 Literatur

[1] S. Baek, M. Carneiro, and M. J. H. Heule. A flexible proof format for SAT solver-elaborator communication. *Log. Methods Comput. Sci.*, 18(2), 2022. doi: 10.46298/lmcs-18(2:3)2022.

[2] T. Balyo, M. Heule, M. Järvisalo, M. Iser, and M. Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, Finland, 2022. URL `https://helda.helsinki.fi/bitstream/handle/10138/347211/sc2022_proceedings.pdf`.

[3] A. Biere, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In M. Heule, M. Järvisalo, and M. Suda, editors, *SAT Competition 2021*, 2021.

[4] A. Biere, M. Järvisalo, and B. Kiesl. Preprocessing in SAT solving. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391 – 435. IOS Press, 2nd edition edition, 2021.

[5] L. Cruz-Filipe, M. J. H. Heule, J. Hunt, Warren A., M. Kaufmann, and P. Schneider-Kamp. Efficient certified RAT verification. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi: 10.1007/978-3-319-63046-5_14.

[6] M. Fleury and A. Biere. Efficient All-UIP learned clause minimization. In C. Li and F. Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2021. doi: 10.1007/978-3-030-80223-3_12.

[7] P. Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020. doi: 10.1007/s10817-019-09525-z.

[8] N. Wetzler, M. Heule, and J. Hunt, Warren A. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *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*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi: 10.1007/978-3-319-09284-3_31.