Certifying Incremental SAT Solving

Katalin Fazekas [ID][1], Florian Pollitt [ID][2], Mathias Fleury [ID][2], and Armin Biere [ID][2]

[1] TU Wien, Vienna, Austria
katalin.fazekas@tuwien.ac.at
[2] University Freiburg, Freiburg, Germany.
pollittf@uni-freiburg.de, fleury@cs.uni-freiburg.de, biere@cs.uni-freiburg.de

Abstract

Certifying results by checking proofs and models is an essential feature of modern SAT solving. While incremental solving with assumptions and core extraction is crucial for many applications, support for incremental proof certificates remains lacking. We propose a proof format and corresponding checkers for incremental SAT solving. We further extend it to leverage resolution hints. Experiments on incremental SAT solving for Bounded Model Checking and Satisfiability Modulo Theories demonstrate the feasibility of our approach, further confirming that resolution hints substantially reduce checking time.

1 Introduction

Many automated reasoning tasks rely on propositional satisfiability (SAT) solvers, including hardware model checking [14, 38], planning [24] Satisfiability Modulo Theories (SMT) [9, 36], and theorem proving [39]. Moreover, most of the time, they use SAT in an incremental way by adding and querying formulas incrementally with the same SAT solver instance. The benefit of such an incremental approach [19, 40] is that information gained in one query can be reused to speed up subsequent ones. Reused information includes the result of simplification obtained through preprocessing steps, learned clauses from the search procedure and other internal state of the SAT solver such as variable scores used in the decision heuristics.

In our experiments we consider two such important applications. The first is bounded model checking (BMC) [10], where the distance k from an initial state to the conjectured bad state is increased one-by-one and the encoding of the reachability problem within k steps can reuse the encoding of all $k-1$ transitions of the previous $k-1$ reachability problem. The second application is (lazy) Satisfiability Modulo Theories [9, 36], where the SAT solver drives the search process but delegates theory reasoning to dedicated decision procedures which provide theory lemmas. In both cases incremental use of the SAT solver gives huge performance gains.

As SAT solvers are consistently improving [12] an important argument in favor of factoring out SAT solving from an automated reasoning task is that is allows simply updating the SAT solver to improve overall performance. This is also one of the motivations of the recently proposed generic incremental interface IPASIR-UP [21], which not only allows coarse-grained interaction with the SAT solver as in BMC but also fine-grained interaction as in SMT.

On the other hand one of the biggest achievements in SAT solving in the last decade was to provide certificates for the result of the SAT solver, which can be checked independently, even with verified checkers, in order to guarantee that SAT solver results can be completely trusted. For satisfiable queries a model (satisfying assignment) is such a certificate and for unsatisfiable SAT queries, polynomially machine checkable proofs are generated by the SAT solver [27, 28].

Since 2016, proofs have become mandatory in the main track of the international SAT solver competition [4], and have been adopted in industrial usage of SAT [15, 34]. Moreover, they have many applications in automated reasoning tasks itself, from core generation, over lifting proofs to interpolation. SMT proof generation is a highly sought feature. Ongoing work

including [6, 7, 29, 35] aims to lift the trustworthiness achieved in SAT solving to SMT. There is similar motivation behind recent certification approaches for model checking [41–44].

However, despite some recent work [30], which needs a second pass over the whole proof, we argue that a *principled approach* to proof generation in incremental SAT solving is still missing. In this paper we show what is needed to make clausal proof formats incremental, for SAT and potentially other formalisms, describe checking algorithms and provide experimental evidence that also for incremental solving we can produce and check proofs efficiently. A major new feature is that proof checking is incremental too. This means we can check proofs online, without the need to store and post-process them on disk, in contrast to [30].

On top of our preliminary study presented at the MBMV'24 workshop [22] we extend our incremental proof framework to fine-grained interactions between the SAT solver and the user, as it is common in SMT. We further introduce a second linear proof format with clause and antecedent identifiers as resolution hints, which, as our experiments show, reduces checking time substantially. Besides evaluating hard bounded model checking problems, we also provide experimental results on generating and checking proofs for SMT on the SAT solver side. Finally, we have extended MiniSat, Glucose and CryptoMiniSat to produce incremental proofs, in order to show that our approach generalizes beyond CaDiCaL, which remains our main focus.

This paper is structured as follows. The next section (Sect. 2) gives preliminaries and briefly presents the IDRUP format. In Sect. 3 we introduce our linear proof format LIDRUP with clause identifiers and hints before we briefly go over how to support fine-grained interactions. Section 4 describes our checking algorithms and the implementation of our proof checkers. We use a dedicated fuzzing approach discussed in Sect. 5 to develop and test proof production and checking. The experimental set-up in Sect. 6 is followed by the experimental results in the context of hard bounded model checking problems in Sect. 7 and SMT solving in Sect. 8.

2 Preliminaries

Given a set of variables $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, a propositional formula F over $\mathcal{V}$ in conjunctive normal form (CNF) is a conjunction of clauses ($F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$), where a clause is a disjunction of literals ($C_i = l_1 \vee l_2 \vee \cdots \vee l_{k_i}$) and a literal l_i is a Boolean variable (v_i) or its negation ($\bar{v}_i$). The set of literals is denoted $\mathcal{L} = \{l_1, \bar{l}_1, \ldots\}$. When it is clear from the context, we also interpret formulas as multi-sets of clauses and clauses as sets of literals.

We define a *truth assignment* to be a partial function assigning truth values *true* or *false* to variables. Literal $l = v_j$ (resp. $l = \bar{v}_j$) is *satisfied* by a truth assignment if it assigns *true* (resp. *false*) to v_j. A clause is satisfied by it if at least one of its literals is satisfied by it. A CNF formula is satisfied by a truth assignment if every clause of it is satisfied by it. If such a satisfying truth assignment exists, the formula is called *satisfiable*, otherwise it is *unsatisfiable*.

An incremental SAT problem is a sequence of *queries* where each query (Δ, A) consists of a set of clauses Δ and a set A of *assumptions*. A query poses the question whether the set of all clauses accumulated from all previous queries and the current set of clauses and current assumptions are together satisfiable or not. More formally, an incremental SAT problem P is $\langle Q_1, Q_2, \ldots, Q_n \rangle$ where for each $1 \leq i \leq n$, query $Q_i = (\Delta_i, A_i)$ is satisfiable if and only if $(\bigwedge_{j=1}^{i} \Delta_j) \wedge A_i$ is a satisfiable formula. The following example demonstrates this formalization and shows some further possible representations of incremental SAT problems.

Example 2.1 (Incremental SAT Problem P). Consider clauses with $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$:

$$C_1 = (\bar{v}_1 \vee v_3 \vee v_4), \quad C_3 = (\bar{v}_2 \vee \bar{v}_3 \vee v_4), \quad C_5 = (\bar{v}_1 \vee v_2),$$
$$C_2 = (\bar{v}_1 \vee v_3 \vee \bar{v}_4), \quad C_4 = (\bar{v}_2 \vee \bar{v}_3 \vee \bar{v}_4), \quad C_6 = (v_1 \vee \bar{v}_2), \quad C_7 = (v_1 \vee v_2).$$

```
s.clause(-1, 3, 4);
s.clause(-1, 3,-4);
s.clause(-2,-3, 4);
s.clause(-2,-3,-4);
s.assume(1);
s.assume(2);
assert(s.solve() == 20);
assert(s.failed(1));
assert(s.failed(2));
s.clause(-1, 2);
s.clause( 1,-2);
assert(s.solve() == 10);
assert(s.val(1) < 0);
assert(s.val(2) < 0);
s.clause( 1, 2);
assert(s.solve() == 20);
```

```
p icnf
i -1 3 4 0
i -1 3 -4 0
i -2 -3 4 0
i -2 -3 -4 0
q 1 2 0
s UNSATISFIABLE
f 1 2 0
i -1 2 0
i 1 -2 0
q 0
s SATISFIABLE
v -1 -2 -3 -4 0
i 1 2 0
q 0
s UNSATISFIABLE
f 0
```

(a) The incremental SAT problem P in C++. The code follows the interface of CaDiCaL [11], but the standard C API of incremental SAT solvers (Ipasir [3]) has similar functionalities.

(b) The incremental SAT problem P in ICNF format, an extension of DIMACS to capture incremental SAT queries in a single file (*cf.* [22] or Fig. 11).

Figure 1: Two further representations of P, the incremental SAT problem of Example 2.1.

Then the following three queries together form the incremental SAT problem $P = \langle Q_1, Q_2, Q_3 \rangle$:

$$
\begin{aligned}
Q_1 &= (\{C_1, C_2, C_3, C_4\}, \{v_1, v_2\}) &\mapsto& \quad C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge v_1 \wedge v_2 & \textit{satisfiable?} \\
Q_2 &= (\{C_1, C_2, C_3, C_4, C_5, C_6\}, \emptyset) &\mapsto& \quad C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6 & \textit{satisfiable?} \\
Q_3 &= (\{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}, \emptyset) &\mapsto& \quad C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6 \wedge C_7 & \textit{satisfiable?}
\end{aligned}
$$

The first query (Q_1) uses assumptions and asks if there is a solution to $C_1 \wedge \cdots \wedge C_4$ such that both v_1 and v_2 are true. Since there is no such satisfying truth assignment, the answer to that query is UNSATISFIABLE. The next query (Q_2) asks if strengthening the clauses of Q_1 with C_5 and C_6, i.e. if $C_1 \wedge \cdots \wedge C_4 \wedge C_5 \wedge C_6$ is satisfiable. Though Q_1 was unsatisfiable and Q_2 refines it with further clauses, the assumptions v_1 and v_2 of Q_1 are not considered anymore in Q_2, hence the answer to Q_2 is SATISFIABLE (consider the truth assignment $\{\bar{v}_1, \bar{v}_2, v_3, v_4\}$).

The third query further constrains the clause set of Q_2 by conjoining C_7 to it and, just as in Q_2, no assumptions are considered here. Since $C_1 \wedge \cdots \wedge C_4 \wedge C_5 \wedge C_6 \wedge C_7$ is unsatisfiable, the answer to Q_3 is UNSATISFIABLE. Now the accumulated clause set became unsatisfiable, independently from any assumptions, and thus no satisfiable queries are possible after Q_3.

Fig. 1 shows how such an incremental SAT problem can be formulated to be solved by an incremental SAT solver. The usual way is to define such problems step-by-step through the API of SAT solvers (Fig. 1a). On the right side (Fig. 1b) the complete interaction between SAT solver and user during solving P is described in the ICNF format(*cf.* [22] or Fig. 11).

Certifying the correctness of the answer of a SAT solver is a fundamental feature, expected to be supported by every modern tool. When a problem is satisfiable, a satisfying truth assignment,

also called *model*, suffices as witness. When a formula is unsatisfiable, however, the solver provides a formal *proof* which justifies the inconsistency of the given problem.

The most common clausal proof formats of SAT solvers capture contradictions found during solving as RUP (Reverse Unit Propagation) [23, 25] and RAT (Resolution Asymmetric Tautologies) [26] inference steps. Each RUP proof step contains a clause which is either an input clause or can be derived from previous clauses by a single RUP inference step as follows. A clause C can be derived from a formula F with a RUP inference step if unit propagation on the conjunction of F and the negation of C leads to a conflict.

Similarly, the RAT proof system allows the derivation of clauses by RAT inference steps, where a clause C with literal l can be derived from a formula F with a RAT inference step if every possible resolution with C on l in F derives a RUP resolvent clause.

Clauses derived from other clauses of a formula are redundant and can be deleted again without changing the satisfiability of the problem. Including such deletion steps in RUP and RAT proofs (called then DRUP and DRAT) can significantly speed up proof checking. Recent proof formats, such as FRAT [2] and LRAT [17], further enhance proof checking by including in each clause addition step some *hints*, a chain of antecedent clause identifiers.

One challenge in certification of incremental SAT queries is how to certify with existing SAT proof techniques when the set of clauses is satisfiable, but none of its solutions satisfies the assumptions (i.e., when $(\bigwedge_{j=1}^{i} \Delta_j) \wedge A_i$ is unsatisfiable, but $(\bigwedge_{j=1}^{i} \Delta_j)$ is satisfiable). Therefore we proposed at the MBMV'24 workshop [22] the IDRUP proof format. It extends DRUP by allowing to capture proofs of incremental SAT queries. In IDRUP each incremental query can be explicitly certified: (i) SAT queries are proven by a given (partial) satisfying truth assignment (ii) unsatisfiable queries are certified by deriving $\bar{A}_i$. Note that $\bar{A}_i$ is the empty clause if the query is unsatisfiable without or independent of any assumptions. The following example illustrates how DRUP and IDRUP proofs look like for incremental SAT problems.

Example 2.2 (cont. of Example 2.1)**.** Fig. 1a demonstrates how to solve problem P using an incremental SAT solver. To verify the correctness of the answer to a query, it generates a clausal proof derivation (for example by listing the derived RUP clauses). However, for existing proof formats (and SAT solvers) it remains unclear how the proof should look when the accumulated clauses are satisfiable, but not under a given set of assumptions. As illustrated in Fig. 2a, one option is to derive a conflict as if assumptions would have been original input unit clauses. This however intermixes several conflicts in one proof, making it challenging to certify. Another option (see Fig. 2b) is to derive the empty clause only when the clauses on their own are contradictory. This however has the risk that the produced proof will contain no conflict at all, for example when every query of the problem is only unsatisfiable due to assumptions. A third option is our IDRUP proof format (see Fig. 2c) which provides explicit certification for every *query*, i.e., both satisfiable and unsatisfiable queries. It keeps track of clause derivation and deletion steps done by the solver ('l', 'd' and 'w' lines), as well as the returned answer ('s' lines) and any further queried information, such as models ('m' lines) and unsatisfiable cores ('u' lines). For more details on IDRUP see [22] or Fig. 12.

3   Supporting Clause Identifiers and Antecedent Hints

Our IDRUP proof format and checker, first proposed at the MBMV'24 workshop [22], has the goal to establish a certification standard for incremental use cases of SAT solvers. However, the original version comes with limitations in terms of flexibility and particularly scalability. We address these scalability shortcomings by extending our framework to LIDRUP, which supports
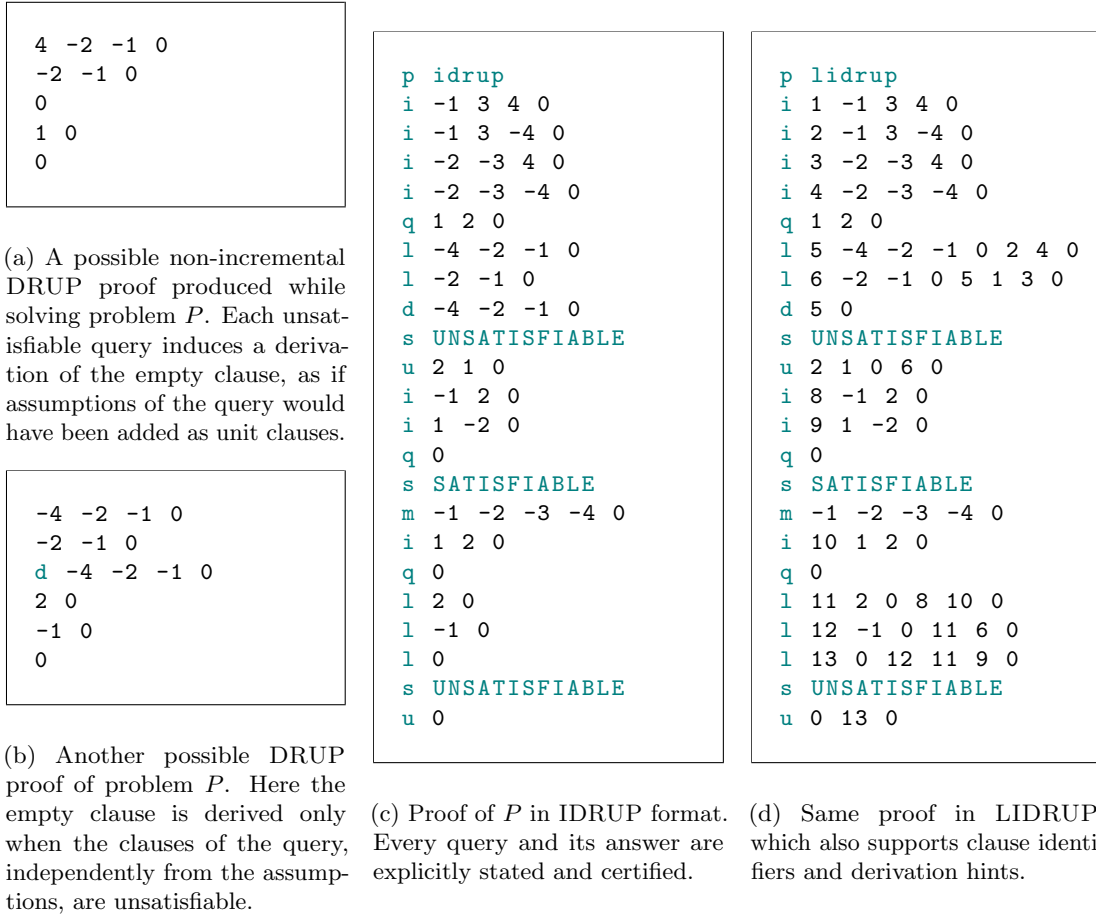
```
 4 -2 -1 0
 -2 -1 0
 0
 1 0
 0
```

(a) A possible non-incremental
DRUP proof produced while
solving problem P. Each unsat-
isfiable query induces a deriva-
tion of the empty clause, as if
assumptions of the query would
have been added as unit clauses.

```
 -4 -2 -1 0
 -2 -1 0
 d -4 -2 -1 0
 2 0
 -1 0
 0
```

(b) Another possible DRUP
proof of problem P. Here the
empty clause is derived only
when the clauses of the query,
independently from the assump-
tions, are unsatisfiable.

```
p idrup
i -1 3 4 0
i -1 3 -4 0
i -2 -3 4 0
i -2 -3 -4 0
q 1 2 0
l -4 -2 -1 0
l -2 -1 0
d -4 -2 -1 0
s UNSATISFIABLE
u 2 1 0
i -1 2 0
i 1 -2 0
q 0
s SATISFIABLE
m -1 -2 -3 -4 0
i 1 2 0
q 0
l 2 0
l -1 0
l 0
s UNSATISFIABLE
u 0
```

(c) Proof of P in IDRUP format.
Every query and its answer are
explicitly stated and certified.

```
p lidrup
i 1 -1 3 4 0
i 2 -1 3 -4 0
i 3 -2 -3 4 0
i 4 -2 -3 -4 0
q 1 2 0
l 5 -4 -2 -1 0 2 4 0
l 6 -2 -1 0 5 1 3 0
d 5 0
s UNSATISFIABLE
u 2 1 0 6 0
i 8 -1 2 0
i 9 1 -2 0
q 0
s SATISFIABLE
m -1 -2 -3 -4 0
i 10 1 2 0
q 0
l 11 2 0 8 10 0
l 12 -1 0 11 6 0
l 13 0 12 11 9 0
s UNSATISFIABLE
u 0 13 0
```

(d) Same proof in LIDRUP,
which also supports clause identi-
fiers and derivation hints.

Figure 2: Certificates for the incremental SAT problem of Ex. 2.1. On the left we show possible
but inadequate DRUP proofs generated for such a problem. These short-comings are addressed
by our new formats, with examples in the middle (IDRUP) and on the right (LIDRUP).

clause identifiers and antecedent hints. This reduces proof checking time substantially, similarly
to LRAT vs. DRAT in non-incremental certification [32, 33].

We first describe the (i) **proof steps** in our LIDRUP proof format informally, then give
precise but abstract (ii) **formal semantics**, before presenting (iii) **concrete syntax**. Readers
familiar with the standard proof formats of non-incremental SAT solvers maybe can skip to
Example 3.1 which allows to grasp the main features of IDRUP and LIDRUP compared to
DRUP. The list of possible LIDRUP proof steps is as follows:

- **Input clauses ('i'):** An input clause addition step has a unique clause identifier (ID)
  followed by its literals, encoded as in the DIMACS format (i.e., as integers). The clause IDs
  are not required to be in order and input clauses can occur during solving too (*cf.* Sect. 8).

- **Queries ('q'):** A query step indicates that the solver is asked to solve the accumulated
  formula, consisting of all clauses before this line (possibly weakened and restored as
  discussed below), under the additionally given (possibly empty) set of assumption literals.

5

- **Learned clauses ('l'):** This step introduces a unique clause ID for followed by the literals of a clause that has been derived by the solver (e.g. during conflict analysis [31] or preprocessing [13]). For LIDRUP in contrast to IDRUP the antecedent clause IDs have be to added as hints too (sufficient to derive the learned clause by resolution).

- **Deleted clauses ('d'):** This step indicates that a set of clauses (given by their IDs) is completely eliminated from the formula and can be ignored in subsequent proof steps.

- **Weakened clauses ('w'):** This step indicates that a set of clauses (given by their IDs) is temporarily removed (i.e. weakened [20]) from the formula and can therefore be ignored when checking subsequent proof steps. These clauses may be reintroduced (i.e. restored [20]) later in the proof. Thus their removal must be distinguished from deleted clauses.

- **Restored clauses ('r'):** This step indicates that a set of clauses (given by their IDs) have been reinserted (i.e., restored [20]) into the formula because some of the solver's previous weakening steps need to be reversed.

- **Status ('s'):** This step explicitly states the SAT solver's answer to the previous query (i.e., the last 'q' line), it can be SATISFIABLE, UNSATISFIABLE or UNKNOWN. The UNKNOWN answer indicates that the solver was terminated before reaching a conclusion.

- **Models ('m'):** This step gives the model produced by the SAT solver after receiving a SATISFIABLE answer. The model is supposed to satisfy all input clauses given before this step, including weakened and also input clauses added by a propagator (*cf.* Sect. 8).

- **Unsatisfiable cores ('u'):** After an UNSATISFIABLE answer, this step lists all failed assumptions of the last query, i.e., the solver claims that these literals form an unsatisfiable core (the input clauses are unsatisfiable if these literals are assumed). Again as for learned clauses, LIDRUP requires in contrast to IDRUP to also specify the antecedent clause IDs as hints (sufficient to derive the clause consisting of the negation of the core literals).

Note that in non-incremental SAT problems it is guaranteed that none of the previously weakened clauses will be restored, so in the proofs of these problems there was no need to distinguish between drop and weaken steps, nor between learned and restored clauses. For a more formal treatment of incremental solving see [20, 30].

The state of a LIDRUP proof can be captured on an abstract level by a triplet $(\sigma, \mathcal{A}, \mathcal{P})$, where σ is a partial function $\mathbb{N} \to \mathbb{P}(\mathcal{L})$ mapping clause identifiers to sets of literals (i.e., representing clauses), $\mathcal{A} \subset \mathbb{N}$ is the set of IDs of *active* clauses and $\mathcal{P} \subset \mathbb{N}$ is the set of clause IDs of the *passive* clauses. Initially the domain of σ, and the sets $\mathcal{A}$ and $\mathcal{P}$ are all empty. Then, given a LIDRUP proof, the following function describes how a step S updates the state of the proof. Note that in Sect. 4 we will provide further details on how to validate if a proof step is correct, while here we only show how a correct step changes the state of the proof.

$$(\sigma', \mathcal{A}', \mathcal{P}') = \begin{cases} (\sigma \cup \{id \mapsto C\}, \mathcal{A} \cup \{id\}, \mathcal{P}) & \text{if S is 'i' line with } id \in \mathbb{N}, \text{ clause } C \\ (\sigma \cup \{id \mapsto C\}, \mathcal{A} \cup \{id\}, \mathcal{P}) & \text{if S is 'l' line with } id \in \mathbb{N}, \text{ clause } C \\ (\sigma \setminus \mathcal{I}, \quad \mathcal{A} \setminus \mathcal{I}, \quad \mathcal{P}) & \text{if S is 'd' line with } \mathcal{I} \subset \mathbb{N} \text{ clause IDs} \\ (\sigma, \quad \mathcal{A} \setminus \mathcal{I}, \quad \mathcal{P} \cup \mathcal{I}) & \text{if S is 'w' line with } \mathcal{I} \subset \mathbb{N} \text{ clause IDs} \\ (\sigma, \quad \mathcal{A} \cup \mathcal{I}, \quad \mathcal{P} \setminus \mathcal{I}) & \text{if S is 'r' line with } \mathcal{I} \subset \mathbb{N} \text{ clause IDs} \\ (\sigma, \quad \mathcal{A}, \quad \mathcal{P}) & \text{otherwise.} \end{cases}$$

where $\sigma' = \sigma \setminus \mathcal{I}$ means that for all $n \in \mathcal{I}$: $\sigma'(n)$ is undefined, with other values unchanged.

```
<lidrup>   =  { <comment> "\n" } <header> "\n" { <line> "\n" }
<comment>  =  "c " <any-character-but-new-line>
<header>   =  "p lidrup"
<line>     =  <comment> | <input>   | <query>  | <lemma> | <delete> |
              <weaken>  | <restore> | <status> | <model> | <core>
<input>    =  "i " <id> { " " <literal> } " 0"
<query>    =  "q" { " " <literal> } " 0"
<lemma>    =  "l " <id> { " " <literal> } " 0" { " " <id> } " 0"
<delete>   =  "d" { " " <id> } " 0"
<weaken>   =  "w" { " " <id> } " 0"
<restore>  =  "r" { " " <id> } " 0"
<status>   =  "s SATISFIABLE"  | "s UNSATISFIABLE" | "s UNKNOWN"
<model>    =  "m" { " " <literal> } " 0"
<core>     =  "u" { " " <literal> } " 0" { " " <id> } " 0"
<id>       =  <pos>
<literal>  =  <pos> | <neg>
<pos>      =  "1" | "2" | ... | <INT_MAX>
<neg>      =  "-" <pos>
```

Figure 3: Syntax of the LIDRUP proof format. Choices are separated by vertical bars, while application of the Kleene's star operator to denote finite abitrary repetition is marked with curly braces. Compared to the IDRUP format LIDRUP requires clause identifiers for input and lemma lines and hints in form of antecedent clause identifier lists for lemmas and unsatisfiable core lines. For deleting, weakening and restore lines the IDRUP format only expects a single clause given explicitly by its literals instead of clause identifiers. Thus IDRUP lines uniformly only consist of literals except for comments, the header and status lines.

An important difference compared to the semantics of IDRUP proofs is that since each clause has a unique identifier, there is no need to consider multi-sets of clauses any more. The clause IDs also allow to delete, restore or weaken multiple clauses in a single step. Further, as we will see it in Sect. 4, they also simplify the implementation details of proof checking.

Regarding syntax, LIDRUP builds on IDRUP as LRAT extends DRAT. Our proposed LIDRUP format is indeed very similar to IDRUP, with only a few very important differences: (i) a unique identifier for each clause is explicitly introduced, (ii) learned clauses and cores are annotated with clause identifier antecedent chain hints of resolution steps and (iii) batched restore, delete and weaken steps. The syntax of the LIDRUP format is depicted in Fig. 3. Note that the proof header does not require the specification of the number of clauses nor variables because we are considering incremental problems where this is usually not known in advance.

Example 3.1. Continuing our running example, Fig. 2d depicts how a certificate for the incremental problem P looks in the LIDRUP proof format. In that example the number of steps is the same as in IDRUP, but this time every new (input or derived) clause starts with an identifier. Further, each clause addition step has a second component providing resolution hints, i.e., identifiers of the antecedent clauses. Clause deletion, weakening, and restore steps ('d', 'w' and 'r') can also use these clause identifiers, thereby batching multiple steps into one.

There is recent interest in the context of incremental SAT reasoning to precisely describe more fine-grained interactions between the user and the SAT solver during SAT solving. Target applications are for instance lazy SMT [36], or Satisfiability Modulo Symmetries [45].
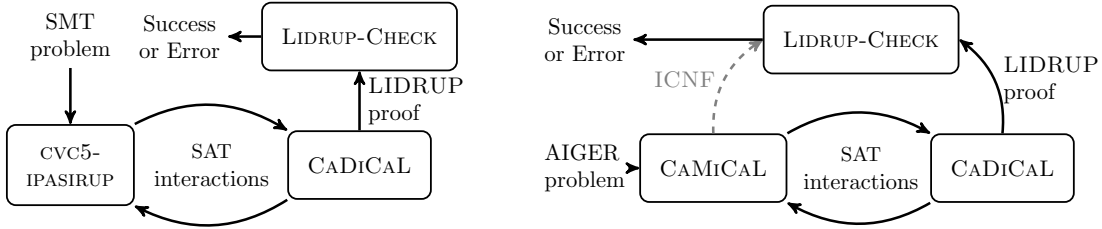
Figure 4: Workflow of proof generation in two of our experiments.

In this regard, the recently proposed Ipasir-Up interface [21] notifies users about internal assignments done by the SAT solver. At the same time, it allows users to add new constraints to the problem, while the solver is solving the query (is within the CDCL loop). Hence, use cases of the Ipasir-Up interface heavily rely on the incremental reasoning features of the underlying SAT solver and so producing certificates of them is essential but challenging.

However, our new proof formats LIDRUP and the new version of IDRUP in this paper easily allow to capture not just the solver reasoning steps, but user interactions as well, including Ipasir-Up use cases. The only additional requirement is that in such use cases input clauses are allowed to be added *between* the query ('q') and answer status lines ('s'). As a consequence the proof contains all clauses added by the external propagator during solving too.

Repeating input and external clauses in the certificate allows proof checkers to validate that the problem solved by the solver is indeed the same problem that the user intended to solve, i.e., the interactions between the solver and the user are correct. However, this feature can quickly become cumbersome in applications where the interaction is relatively simple but potentially introduces an enormous amount of constraints. For this reason, we have defined LIDRUP so that it can be checked even in the absence of the corresponding ICNF interaction file.

4   Checking Proofs with Clause Identifiers and Hints

Our LIDRUP proof checker Lidrup-Check[1] started as a fork of the Idrup-Check[2] proof checker for IDRUP [22]. It uses hints of antecedent clause identifiers as in the LRAT proof format [17], as supported by CaDiCaL [33], for faster checking of derived lemmas and cores.

Both Idrup-Check and Lidrup-Check provide as a new feature a weaker mode where only the proof file is checked as on the left of Fig. 4. The control-flow described as an automaton of this proof-only and single-file mode is shown in Fig. 5. It also serves as a simple view on how incremental proof checking actually works. Note, that this mode requires the user to trust (or check independently) that the interactions recorded in the proof match those actually occurring.

In general it is recommended that the checker is given both the ICNF interaction file and the proof file (either IDRUP for Idrup-Check or LIDRUP for Lidrup-Check) as on the right of Fig. 4. This makes the control-flow of the checking algorithm much more complex, as it is also supposed to work in an online-mode (as in our experiments through pipes), and we refer to Fig. 6 for more details. One might notice, that the automaton of the proof-only single-file mode of Fig. 5 just omits the interaction states from Fig. 6 which read the ICNF file.

Finally, note that LIDRUP requires clause identifiers for input clauses, while we currently do not require that for ICNF as currently SAT solvers do not provide that information (the
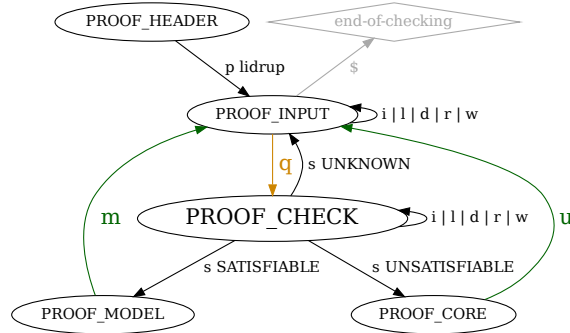
---

[1]https://github.com/arminbiere/lidrup-check
[2]https://github.com/arminbiere/idrup-check

Figure 5: This automaton describes the control-flow of Lidrup-Check (and Idrup-Check if 'p lidrup' is replaced by 'p idrup') in single-file mode (left workflow in Fig. 4). After the header, input clauses in 'i' lines are added, and optionally simplified to lemmas from 'l' lines, which are checked and added too. These clauses can be deleted, weakened and restored according to the 'd', 'w' and 'r' lines. A new incremental query is started with a 'q' line followed by again reading input clauses, lemmas, deleting, weakening and restoring them. An 's' status line ends this part which corresponds to the SAT solver running a CDCL loop interleaved with inprocessing and potential interaction with a user propagator. For instance 'i' lines read in 'PROOF_CHECK' allow to capture theory lemmas provided by an SMT solver. One query is concluded by a model in an 'm' line in the satisfiable case and by an unsatisfiable core of failed assumptions (of the query) in a 'u' line. Early termination or hitting a limit is modelled by an unknown status line.

internal clause identifier) to the user. Therefore we suggest to extend the common Ipasir API along this line, which is also necessary in order to support more advanced usages of a proof tracing SAT solver anyhow (such as in-memory interpolation).

As a consequence we actually allow LIDRUP proofs to introduce clause identifiers in an arbitrary order (in contrast to LRAT) and treat them in essence symbolically. This is facilitated by both storing the active and inactive (weakened) clauses in separate hash tables indexed by the clause identifier. For online checking this is guaranteed not to lead to cyclic proofs. We consider proof trimming as future work, which will then require an additional acyclicity check.

5   Fuzzing and Generating Incremental Proofs

To validate functionality of our IDRUP and LIDRUP implementations in the considered SAT solvers (CaDiCaL, MiniSat, Glucose and CryptoMiniSat), we developed two simple new custom fuzzers Idrup-Fuzz and Lidrup-Fuzz (available in the checker repositories). Adding incremental proof generation, checking and delta-debugging support to CaDiCaL's dedicated model-based fuzzer, Mobical, is complex and left to future work. These fuzzers go through either the generic Ipasir API or directly through the C API of CaDiCaL, generate interaction files and trigger production of an IDRUP or LIDRUP proof from the SAT solver. Subsequently, our checkers Idrup-Check or Lidrup-Check verify the correctness of generated proofs.

Basically, these grammar-based white-box fuzzers produce sequences of incremental SAT problems, consisting of random clauses, assumptions and queries. As for non-incremental CNF fuzzing [16], we pick a random number of variables and then determine the number of generated clauses based on a random clauses-per-variable ratio sampled around the satisfiability threshold of 4.26. This avoids any bias towards satisfiable or unsatisfiable queries. As novel feature we
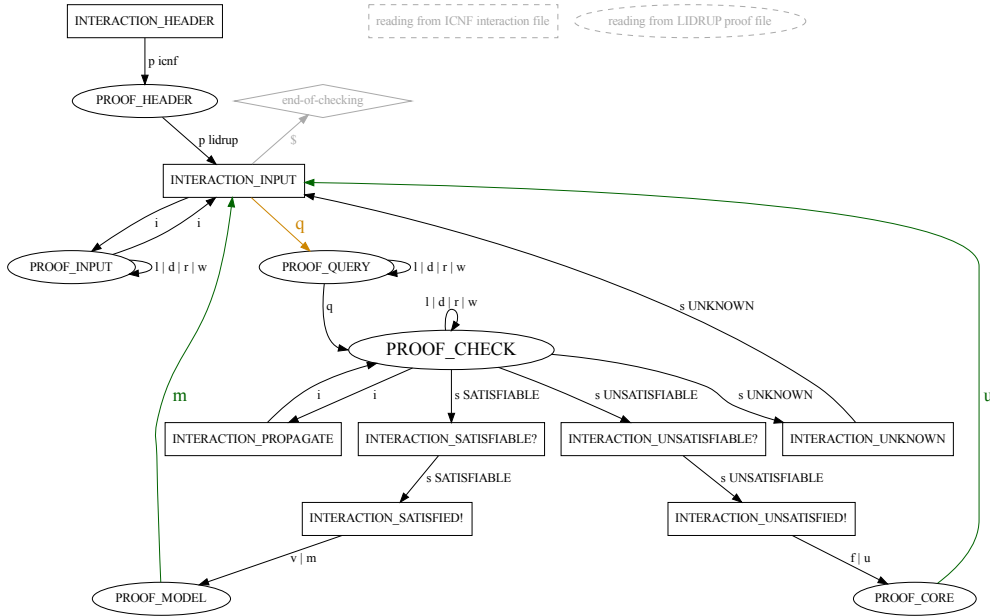
Figure 6: This is the control-flow of LIDRUP-CHECK when parsing and checking in parallel a linear incremental LIDRUP proof and the ICNF interaction file as in the right workflow of Fig. 4. The oval boxes denote states in which the checker reads from the LIDRUP proof file alternating with rectangular states in which the checker reads from the ICNF file. The labels of the states (redundantly) are prefixed with 'INTERACTION_' and 'PROOF_' accordingly. For the edges we only show single-letter tags from Fig. 3 omitting literals and identifiers (but do include the status for 's' lines). Consecutive lines read from the two files with the same tag, i.e., 'i', 'q', 'm' and 'u', are checked to have the same literals. An incremental query starts with the (large orange) 'q' edge leaving the 'INTERACTION_' state and ends with one of the edges labelled with (a large green) 'm' or 'u'. The former concludes a satisfiable query with a model, which is checked to satisfy all input clauses. The latter concludes an unsatisfiable query and gives the list of failing assumptions, i.e., the unsatisfiable core, which is a subset of the assumptions given as argument to the query and also checked do be derivable. On the interaction side we also allow (partial) models through 'v' value lines and (partial) unsatisfiable cores through failed assumption 'f' lines. Those values and literals are then checked to form a subset of the corresponding 'm' and 'u' lines and are not required to match them precisely. The checker run is successful if the end-of-file indicator '$' is read in the 'INTERACTION_INPUT" and thus all of the (claimed) interactions in the ICNF file are checked. The self-loops around 'PROOF_INPUT' and 'PROOF_QUERY' allow to capture the common practice in SAT solvers to simplify clauses and thus starting to omit proof steps while parsing input clauses. Note that these edges are not necessarily followed and checked in the same temporal order as they originally happened, e.g., the 'q' line in the ICNF file might be read after the last input clause is read from the LIDRUP file but before it is simplified as next step following the self-loop in 'PROOF_QUERY'. Similarly the 'i' edge leaving the 'PROOF_CHECK' state observes a clause added by the user propagator using the IPASIR-UP interface during solving and is read from the LIDRUP file and then checked, even though the following 'i' edge reading the corresponding clause from the ICNF file, could be interpreted to have happened before. In any case the checker makes sure queries and cores match, lemmas and cores can be derived, only active clauses can be deleted and weakened, only weakened clauses can be restored, models satisfy input clauses and finally cores match queries.

however split these generated clauses across the randomly generated incremental queries.

One limitation we had to overcome when going through the IPASIR interface was that IPASIR does not allow to set options (we expect this feature to become available in IPASIR 2). We extended IPASIR to allow specifying the file to which the IDRUP proof is written.

To promote the use of the IDRUP format among solvers, we incorporated it into our CaDiCaL SAT solver. We also added it to three other top-tier incremental SAT solvers: MiniSat 2.2.0 [18], Glucose 4.2.1 [1], and CryptoMiniSat 5.11.21 [37]. The implementation in MiniSat and Glucose was straightforward. This is because these solvers handle the original problem's literals without much simplification. We only appropriate printing of header and status lines, as clause tracing is already part of the previously existing DRUP proof generation.

Modifying CryptoMiniSat as a sophisticated inprocessing solver with features like XOR-reasoning is clearly more complex. It creates proofs in DRAT and FRAT [2] (partial LRAT where some hints are missing) formats for non-inprocessing methods. Therefore, we chose to implement only IDRUP, as altering the entire solver to support LIDRUP with hints looked daunting. We also observed that originally IDRUP-Fuzz did not trigger any simplifications (inprocessing) in CryptoMiniSat due to the small size of the generated incremental SAT problems. Thus we further extended the IPASIR API with a `simplify` function, which resulted in much more interesting tests. For instance, we had to modify CryptoMiniSat to preserve clauses for justifying equivalences, which were removed too early before.

The challenge was to achieve proper proof production during importing and simplifying input clauses, which in CryptoMiniSat consists of three steps: The original imported clause (e.g. $a \lor b \lor c$) is logged into the IDRUP proof file with '`i`'. Then, equivalent literals are substituted (e.g., $\neg d \lor c \lor c$ if $b = c$ and $a = \neg d$). In non-incremental DRAT mode these two clauses are interchangeable but for IDRUP the two '`i`' lines in the ICNF interaction file and the IDRUP proof file have to match. Finally duplicated literals are removed ($c \lor c$) and then this version is kept internally. To keep the number of modifications small, we opted to print the intermediate clause $\neg d \lor c \lor c$ in the proof – even if we delete the original clause (e.g., $a \lor b \lor c$) immediately in the next step and the substituted one again immediately if the final clause is different – leading to a higher number of proof steps.

6   Experimental Set-Up

Our experiments were conducted on the bwForCluster Helix with AMD Milan EPYC 7513 CPUs using a memory limit of 16 GB RAM. We evaluated the feasibility of our approach in two typical use cases for incremental SAT reasoning: (i) bounded model checking with the hardware model checker CaMiCaL [20] (1 h timeout) and (ii) solving Boolean abstractions for lazy SMT in cvc5 [5] (5 min timeout). Both applications rely on an incremental SAT solver, although the way in which they interact with that solver differs significantly (cf. Fig. 4).

The coarse-grained interaction of CaMiCaL with the SAT solver during hardware model checking is captured by IPASIR, while SMT solving requires fine-grained interaction with the SAT solver through IPASIR-Up. In both cases, CaDiCaL can produce an IDRUP or LIDRUP certificate recording every interaction and reasoning step it performs. This certificate is then forwarded to LIDRUP-Check to verify its correctness (either through files or pipes). Our experiments aim to show that proof production and online checking using our proposed LIDRUP format and LIDRUP-Check is practical, has only limited overhead, and much less than the IDRUP format.

7   Bounded Model Checking Experiments

To evaluate LIDRUP and the proof checker IDRUP-CHECK, we enhanced CAMICAL, a bounded hardware model checker for the AIGER format, which unrolls a model checking problem up to a specified bound ($k = 1000$ in our experiments) incrementally. Our enhancement records every interaction between CAMICAL and its SAT solver in an ICNF and an IDRUP file. In previous work, we only supported CADICAL (through its API), while we now also support MINISAT and GLUCOSE (via the IPASIR interface, utilizing an additional method for specifying the proof file), as well as CRYPTOMINISAT (through its API).

We use the 300 AIGER models from the hardware model checking competition 2017 (HWMCC'17), as in previous work in [20], and check them until bound $k = 1000$. As performance metric we use the distribution of the individual wall-clock times it took to answer each query (not the whole incremental SAT problem) or the times it took to check individual queries with the proof checker. This information is logged by CAMICAL and the proof checkers. On the illustrating figures the x-axis corresponds to bounds solved over all models, while the y-axis shows the time needed to answer or check the individual SAT queries.

In Fig. 7 we explore various variants of using CADICAL as backend SAT solver for CAMICAL: non-incremental (no proof), solve-only (no proof), writing proof to '/dev/null', writing proof to the file system (only produce proof), storing the proof and subsequent verification (offline verification), and using named pipes for immediate proof consumption by the checker (online verification). The non-incremental approach emerged as significantly slower than its counterparts, emphasizing the critical role of incremental processing. Offline verification outpaced the online version, albeit with a more substantial discrepancy observed in IDRUP compared to LIDRUP.

Utilizing named pipes presents two key advantages: first, it eliminates the need for storing proofs (which can be large, *cf.* Fig. 10), and second, it facilitates simultaneous solver-checker operation through synchronization at the pipe level. However, a full pipe means that the solver and the checker wait on each other. For instance, while experimenting with CRYPTOMINISAT, we discovered "hanging proofs" tied to insufficiently frequent output flushing during proof generation–the solver awaited content in the pipe, and the checker waited for content receipt. To rectify this issue, we had to adjust the default buffer size for CRYPTOMINISAT proof-writing.

In the remaining experiments, we use online-checking through pipes only, as it prevents generation of extensive files and is only slightly slower. We first examine proof sizes of completed model and proof checking runs (*cf.* Fig. 10). It is evident that LIDRUP proofs can be substantially larger than those generated by IDRUP. This disparity persists even when steps such as delete, weaken, and restore are combined in LIDRUP; the extra chains offset the reduced cost. Our subsequent investigation compares IDRUP and LIDRUP (*cf.* Fig. 9): despite the increased expense of parsing, the proof verification process is considerably faster for LIDRUP due to the simplified nature of checking all operations ('l' but also 'w', 'd', 'r', 'l' and 'u'). The IDs provide a simple way to identify clauses and focusing resolution on the antecedent clauses is much faster than propagation through reverse unit propagation.

Beyond using CADICAL as SAT solver backend we have also experimented with our modified versions of MINISAT, GLUCOSE, and CRYPTOMINISAT (*cf.* Fig. 8). Even though there is a clear slow-down due to proof checking, using the solver non-incrementally (*cf.* Fig. 7) is even slower (even without proof checking). All calls were checked and we did not find any discrepancy.

The CADICAL backend with proof checking is faster than any other solver without proof checking. The performance of CRYPTOMINISAT is surprisingly worse than expected, which we contribute to a potential performance regression compared to the 2020 version submitted to the incremental SAT Competition 2020. We are in the process of reaching out to the author for
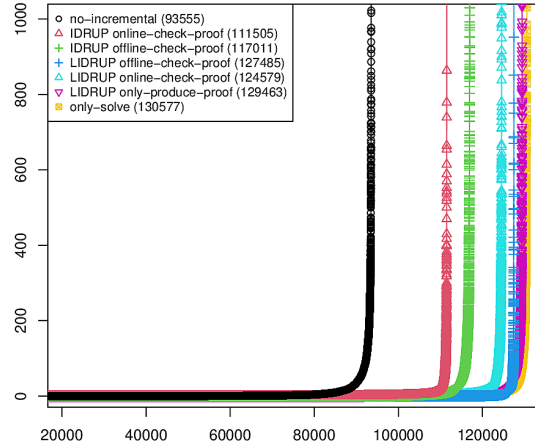
Figure 7: Comparing times it took to solve and check individual bounds (incrementally and non-incrementally) for different workflows of using CaDiCaL as SAT solver backend for CaMiCaL.
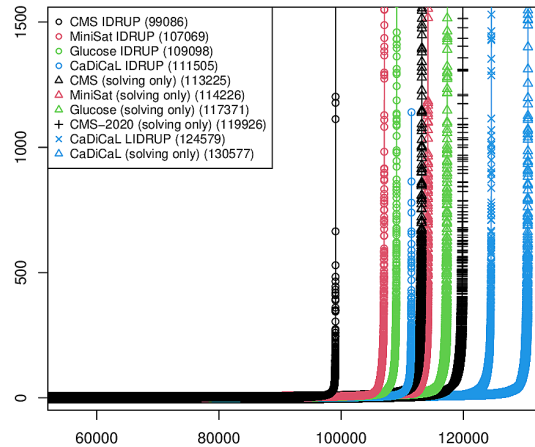

Figure 8: Comparing the time it took to solve and check individual model checking bounds incrementally using different SAT solvers with CaMiCaL, all with IDRUP production, except for CaDiCaL which supports both IDRUP and LIDRUP.

confirmation. Still this does not invalidate the conclusion that adding IDRUP support is easy and proof checking of incremental problems is achievable with a small overhead.

8 Satisfiability Modulo Theories Experiments

In this experiment we evaluated the feasibility of LIDRUP production and checking in the context of solving the Boolean abstraction of SMT problems. Our goal is to demonstrate that proof production of the SAT solver of an SMT solver is affordable in practice, rather than to evaluate the performance of the SMT solver. Thus, we focused on quantifier free problems with linear real arithmetic as a background theory (QF_LRA benchmarks of SMT-LIB [8]), a theory
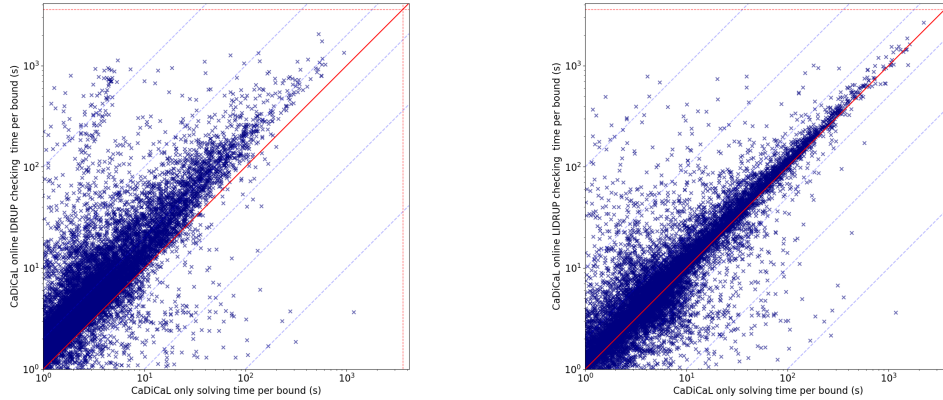
Figure 9: IDRUP and LIDRUP checking vs. solving time per individual model checking bound.
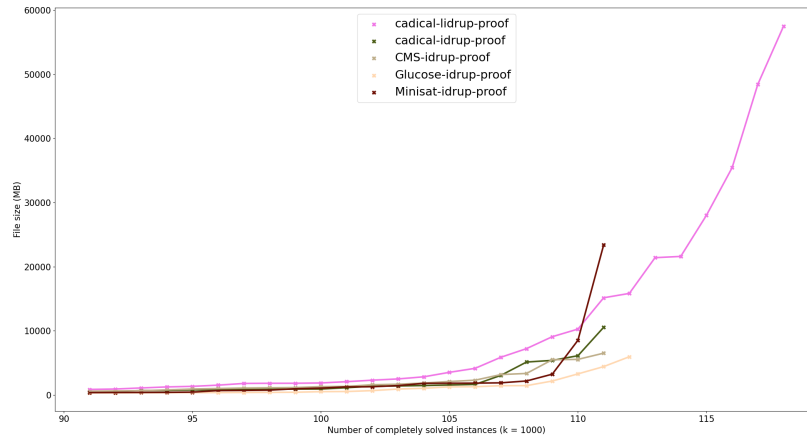

Figure 10: Sizes of completed proofs of incremental model checking instances where model checking either reached the bound $k = 1000$ or the query of the bound became satisfiable and further proof checking went through too.

that is relevant for many applications, but not the most technically challenging for SMT solvers.

We experimented with CVC5 [5], a state-of-the-art SMT solver that supports to switch out its Boolean engine with any CDCL SAT solver that implements the IPASIR-UP interface [21]. We used a recent version of CVC5, referred to as CVC5-IPASIRUP[3], that supports the most current version of IPASIR-UP. We plugged into CVC5-IPASIRUP our extended version of CADICAL[4] by adjusting some compilation parameters of CVC5, without modifying CVC5 or CADICAL any further. The workflow of the experiment is shown at the left side in Fig. 4.

Our results are shown in Table 1. The table is split into two rows: non-incremental and incremental QF_LRA SMT problems. Note that the problems solved by CADICAL are in both cases incremental: given an initial SAT problem (the Boolean abstraction of the input SMT problem), it is continuously refined with further theory lemmas during the CDCL loop.

---

[3]Though this update of CVC5 is not released yet, it is publicly available at https://github.com/aniemetz/cvc5/tree/cdclt-cadical-removable. Commit 6cdd2e6 in the fork https://github.com/kfazekas/cvc5-cadical-idrup-test shows the necessary modification.

[4]Available at https://github.com/arminbiere/cadical/tree/ipasirup-lidrup

Benchmark	All		Solved wo. Proof		Solved & Verified	
	Instances	Queries	Instances	Queries	Instances	Queries
QF_LRA	1754	1754	1671	1671	1650	1650
incr-QF_LRA	10	1515	2	730	2	725

Table 1: Number of solved and verified incremental SAT queries of CaDiCaL in cvc5-ipasirup on incremental and non-incremental SMT problems in QF_LRA (with 300 sec. time limit).

However, incremental SMT problems result in more incremental SAT queries and employ several assumptions. In contrast, non-incremental SMT problems induce only one SAT problem, which is incrementally extended with further clauses during solving, without any assumptions.

Without any proof production or checking, cvc5-ipasirup can solve 95% of the non-incremental SMT problems (see column 'Solved wo. Proof'). When the underlying CaDiCaL solver generates a LIDRUP proof and pipes it directly to Lidrup-Check while solving, 21 (~1%) fewer instances are solved. However, in that case the reasoning of the SAT solver is completely certified and verified (see column 'Solved & Verified'). Proof production does not affect the number of fully solved instances for incremental SMT problems. Only 2 out of the 10 problems are solved completely, as it is shown in the second row of Table 1. However, these problems generate several SAT queries (a total of 1515 queries), and proof production and checking results in 5 fewer queries to be solved ($< 1\%$).

The cost of LIDRUP proof production and checking was minimal in this experiment. This is not surprising, as in SMT the more complex reasoning typically happens in theory solvers. However, we argue that having LIDRUP proofs readily available in these use cases is still valuable. It can aid debugging and profiling of the interaction between the different reasoning engines. Additionally, it has the potential to serve as a component in more general SMT proofs.

9   Conclusion

We took first steps towards a principled approach to incremental proof production and checking. As two concrete examples of incremental proof formats we presented IDRUP and LIDRUP which capture both the interaction and logical reasoning of incremental solving. We have further implemented and evaluated proof production in various solvers. We introduced our new proof-checkers Idrup-Check and Lidrup-Check which can be used to validate both coarse- and fine-grained interaction with a SAT solver. In our experiments we show that proof production is feasible with a negligible overhead, particularly for the new LIDRUP format with clause identifiers and antecedent hints. As future work we propose to investigate whether our incremental proofs have other usage beside proof checking, such as interpolation or diagnosis and what implications our approach has on the design of incremental SAT solver interfaces.

Acknowledgement

References

[1] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018.

[2] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. *Log. Methods Comput. Sci.*, 18(2), 2022.

[3] Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016.

[4] Tomás Balyo, Marijn J. H. Heule, and Matti Järvisalo. SAT Competition 2016: Recent developments. In Satinder Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 5061–5063. AAAI Press, 2017.

[5] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[6] Haniel Barbosa, Clark W. Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. Generating and exploiting automated reasoning proof certificates. *Commun. ACM*, 66(10):86–95, 2023.

[7] Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. *J. Autom. Reason.*, 64(3):485–510, 2020.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2023.

[9] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[11] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[12] Armin Biere, Mathias Fleury, Nils Froleyks, and Marijn J. H. Heule. The SAT museum. In Matti Järvisalo and Daniel Le Berre, editors, *Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023*, volume 3545 of *CEUR Workshop Proceedings*, pages 72–87. CEUR-WS.org, 2023.

[13] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS

Press, 2021.

[14] Armin Biere and Daniel Kröning. SAT-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018.

[15] Nicolas Breton and Yoann Fonteneau. S3: proving the safety of critical systems. In Thierry Lecomte, Ralf Pinger, and Alexander B. Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*, volume 9707 of *Lecture Notes in Computer Science*, pages 231–242. Springer, 2016.

[16] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

[17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo 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.

[18] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[19] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003.

[20] Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019.

[21] Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: user propagators for CDCL. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[22] Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Incremental proofs for bounded model checking. In Wolfgang Kunz, editor, *27th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV'24), Kaiserslautern, Germany*, volume 314 of *ITG-Fachberichte*, pages 133–143. VDE Verlag, 2024.

[23] Allen Van 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.

[24] Malik Ghallab, Dana S. Nau, and Paolo Traverso. Chapter 7: Propositional satisfiability techniques. In *Automated planning - theory and practice*. Elsevier, 2004.

[25] Evguenii I. Goldberg and Yakov 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*, pages 10886–10891. IEEE Computer Society, 2003.

[26] Marijn J. H. Heule. The DRAT format and DRAT-trim checker. *CoRR*, abs/1610.06229, 2016.

[27] Marijn J. H. Heule. Proofs of unsatisfiability. In Armin Biere, Marijn Heule, Hans van Maaren,

and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 635–668. IOS Press, 2021.

[28] Marijn J. H. Heule and Armin Biere. Proofs for satisfiability problems. In *All about Proofs, Proofs for All (APPA)*, volume 55 of *Math. Logic and Foundations*. College Pub., 2015.

[29] Jochen Hoenicke and Tanja Schindler. A simple proof format for SMT. In David Déharbe and Antti E. J. Hyvärinen, editors, *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70. CEUR-WS.org, 2022.

[30] Benjamin Kiesl-Reiter and Michael W. Whalen. Proofs for incremental SAT with inprocessing. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, pages 132–140. IEEE, 2023.

[31] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021.

[32] Florian Pollitt, Mathias Fleury, and Armin Biere. Efficient proof checking with LRAT in CaDiCaL (work in progress). In Armin Biere and Daniel Große, editors, *Proceedings 26th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-24, 2023*, pages 64–67. VDE, 2023. Accepted.

[33] Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 21:1–21:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[34] Neha Rungta. A billion SMT queries a day (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2022.

[35] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.

[36] Roberto Sebastiani. Lazy satisability modulo theories. *J. Satisf. Boolean Model. Comput.*, 3(3-4):141–224, 2007.

[37] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[38] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE*, 103(11):2021–2035, 2015.

[39] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014.

[40] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 542–545. ACM, 2001.

[41] Emily Yu, Armin Biere, and Keijo Heljanko. Progress in certifying hardware model checking results. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume

12760 of *Lecture Notes in Computer Science*, pages 363–386. Springer, 2021.

[42] Emily Yu, Nils Froleyks, Armin Biere, and Keijo Heljanko. Stratified certification for k-induction. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 59–64. IEEE, 2022.

[43] Emily Yu, Nils Froleyks, Armin Biere, and Keijo Heljanko. Stratified certification for k-induction (extended abstract). In Armin Biere and Daniel Große, editors, *Proceedings 26th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-24, 2023*, pages 68–69. VDE, 2023.

[44] Emily Yu, Nils Froleyks, Armin Biere, and Keijo Heljanko. Towards compositional hardware model checking certification. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Proceedings 23rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'23)*, volume 4, pages 44–54. TU Vienna Academic Press, 2023.

[45] Tianwei Zhang and Stefan Szeider. Searching for smallest universal graphs and tournaments with SAT. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 39:1–39:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

Appendix

The syntax of the ICNF interactions format is shown in Fig. 11, adapted from [22], followed by the syntax of the IDRUP proof format in Fig. 12 without clause identifiers and antecedent identifier hints. Both are closely related to the syntax of LIDRUP (*cf.* Fig. 3).

```
<icnf>    =  { <comment> "\n" } <header> "\n" { <line> "\n" }
<comment> =  "c " <any-character-but-new-line>
<header>  =  "p icnf"
<line>    =  <comment> | <input>  | <query> | <status> | <model> | <core>
             <value>   | <failed>
<input>   =  "i" { " " <literal> } " 0"
<query>   =  "q" { " " <literal> } " 0"
<status>  =  "s SATISFIABLE"  | "s UNSATISFIABLE" | "s UNKNOWN"
<model>   =  "m" { " " <literal> } " 0"
<core>    =  "u" { " " <literal> } " 0"
<value>   =  "v" { " " <literal> } " 0"
<failed>  =  "f" { " " <literal> } " 0"
<id>      =  <pos>
<literal> =  <pos> | <neg>
<pos>     =  "1" | "2" | ... | <INT_MAX>
<neg>     =  "-" <pos>
```

Figure 11: Syntax of the common ICNF interaction format.

```
<idrup>   =  { <comment> "\n" } <header> "\n" { <line> "\n" }
<comment> =  "c " <any-character-but-new-line>
<header>  =  "p idrup"
<line>    =  <comment> | <input>   | <query>  | <lemma> | <delete> |
             <weaken>  | <restore> | <status> | <model> | <core>
<input>   =  "i" { " " <literal> } " 0"
<query>   =  "q" { " " <literal> } " 0"
<lemma>   =  "l" { " " <literal> } " 0"
<delete>  =  "d" { " " <literal> } " 0"
<weaken>  =  "w" { " " <literal> } " 0"
<restore> =  "r" { " " <literal> } " 0"
<status>  =  "s SATISFIABLE"  | "s UNSATISFIABLE" | "s UNKNOWN"
<model>   =  "m" { " " <literal> } " 0"
<core>    =  "u" { " " <literal> } " 0"
<id>      =  <pos>
<literal> =  <pos> | <neg>
<pos>     =  "1" | "2" | ... | <INT_MAX>
<neg>     =  "-" <pos>
```

Figure 12: Syntax of the IDRUP proof format without clause identifiers.