Incremental Proofs for Bounded Model Checking

Katalin Fazekas ‡, Florian Pollitt †, Mathias Fleury †, Armin Biere †
†University Freiburg, Germany and ‡TU Wien, Austria
{fleury,biere}@cs.uni-freiburg.de, pollittf@informatik.uni-freiburg.de, katalin.fazekas@tuwien.ac.at

Abstract

Bounded model checkers show the validity of a property of a hardware or software system to hold up to a certain bound by solving a sequence of related Boolean satisfiability (SAT) problems. An incremental SAT solver, a tool often used by such model checkers, can exploit similarities between these consecutive SAT problems. By avoiding repeated work incremental solving is much more efficient. To increase the trustworthiness of a model checker, it is however important to provide assurance about the correctness of its underlying solving engine. Though modern SAT solvers are expected to produce an independently verifiable certificate when a formula is unsatisfiable, incremental SAT solving involves multiple formulae under different temporary assumptions. In this paper we propose a new proof format for SAT solvers applicable to incremental use cases and demonstrate the viability of it in the context of bounded hardware model checking.

1 Introduction

In a wide range of applications, such as Hardware Model Checking (HWMC) [1, 2, 3], planning [4] or solving Satisfiability modulo Theories (SMT) problems [5] a sequence of related SAT problems need to be solved. For these applications incremental solving using a single solver instance for the whole sequence is greatly beneficial.

Bounded model checking (BMC) [6], for instance, encodes into propositional logic only those parts of the system under verification that are reachable up to a given bound k, starting with $k = 0$ for the initial reset state, and then increasing k, by adding appropriate next-state logic. If the encoded formula was found to be unsatisfiable by a SAT solver, the encoding is extended with further possible steps of the system. This extension-evaluation interplay is repeated until either the property is violated, or a high enough bound (e.g. $k = 1000$ in the hardware model checking competition [7, 8]) of unrolling steps is reached.

Extending an already built and solved SAT formula with further constraints, instead of constructing and solving new formulas in each step, allows to reuse incrementally the same SAT solver instance. The goal of incremental SAT solvers is to exploit the shared constraints between consecutive SAT queries of such use cases and thereby avoid repeated work and reduce solving time [9, 10].

State-of-the-art SAT (and SMT) solvers are quite complex software, already for stand-alone non-incremental usage. This complexity stems from their logically involved algorithms, sophisticated data-structures and further low-level optimizations. In order to trust their results, the SAT community has adopted a certification approach, and since 2016 solvers participating in the main track of the annual competition have to produce certificates [11].

For satisfiable queries a satisfying assignment produced by the SAT solver acts as certificate and easily allows to check correctness. For unsatisfiable queries the SAT solver is required to produce a machine checkable proof certificate instead. However, for incremental usage there is no standardized proof format, even though incremental solving adds another level of complexity and thus can be considered to be even more error-prone.

This lack of trust in incremental solvers lifts of course to applications relying on the efficiency of incremental solving. Hence, it is essential to gain more assurance that an employed solver is indeed correct. In related work on certifying hardware model checking [12, 13, 14, 15] monolithic certificates are checked by non-incremental SAT solvers. This will become a severe bottle-neck when the actual model checking makes use of incremental solving. One of our main motivations of this paper is to explore as alternatives either using incremental proofs in this context or making such incremental proofs accessible to theorem provers.

In this paper we propose a way to extend the proof generation method of SAT solvers such that it can provide *incremental proofs* where unsatisfiability under a set of assumptions is explicitly shown and justified. Beyond generating incremental proofs, we also present how to extend standard proof checking techniques of SAT solvers to be able to efficiently verify such incremental certificates. At the end, we illustrate and evaluate the costs and benefits of the proposed approach in our experiments in the context of bounded hardware model checking with CaMiCaL.

Our results show that the proposed techniques are scalable and has acceptable overhead in solving time, while allows users of model checkers to gain a higher trust in its results.

Overview

After preliminaries in Section 2, we introduce in Section 3 two new formats to capture incremental SAT queries and their incremental proofs in succinct ways. Section 4 presents the semantics of our proposed proof format and describes the implementation of our prototype checker for that. After our experiments (Section 5), we conclude by comparing to related work (Section 6) and briefly discussing future work (Section 7).

2 Preliminaries

Here we very briefly introduce the main concepts that are used and referred to throughout the paper.

Satisfiability Problems

Given a set of Boolean variables $V = \{x_0, x_1, \ldots, x_n\}$, a *literal* is a variable (x_i) or its negation ($\overline{x_i}$), a *clause* is a disjunction of literals, and a propositional formula in *conjunctive normal form* (CNF) is a conjunction of clauses. When it is convenient and clear from the context, we will refer to a formula as a multi-set of clauses, and to a clause as a set of literals. A *unit clause* is a clause with exactly one literal, while a clause without any literals is called the *empty clause*. A (partial) truth assignment is a function $V \to \{\top, \bot\}$, and a literal x_i (resp. $\overline{x_i}$) is satisfied by a truth assignments if it maps x_i to *true* (resp. *false*). We will often represent truth assignments by the set of those literals that are satisfied by it. A clause is said to be satisfied by a truth assignment if at least one of its literals is satisfied by it. A truth assignment that satisfies every clause of a formula F is called a *model* of F. The Boolean satisfiability problem (SAT) is to decide the existence of a model for a given multi-set of clauses. A formula is *satisfiable* if it has a model, otherwise it is *unsatisfiable*.

The empty clause is unsatisfiable, and a unit clause ℓ can be satisfied only by such assignments that satisfy ℓ and falsify $\overline{\ell}$. Thus, when a formula F contains a unit clause ℓ, satisfying it removes all clauses containing ℓ together with every occurrence of $\overline{\ell}$ from every clause of F. Satisfying all unit clauses in F is called unit propagation and can be repeated until either there is no more unit clause in F or one of the clauses become empty. Given clauses $C_1 = C \vee x$ and $C_2 = C' \vee \bar{x}$, applying *resolution* on C_1 and C_2 on variable x derives the *resolvent* clause $C \vee C'$.

Modern SAT solvers are expecting SAT problems to be formulated in the so-called DIMACS file format [16]. DIMACS files start with a header "`p cnf |V| |F|`" (where $|V|$ is the number of variables and $|F|$ is the number of clauses in formula F) followed by a list of each clause of the formula. Clauses are represented as a list of their literals ended with a "`0`", while literals are encoded as non-zero integers. Figure 1 illustrates this format on a formula.

```
p cnf 4 4
-1 3 4 0
-1 3 -4 0
-2 -3 4 0
-2 -3 -4 0
```

Figure 1 This is the CNF formula $(\bar{x}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4)$ in DIMACS representation. It is unsatisfiable if we further assume x_1 and x_2 to be true ($x_1 \wedge x_2$), or they are equal $(\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$ and one of them is true $(x_1 \vee x_2)$, which are the queries we are going to explore incrementally later on.

Queries of Incremental SAT Problems

An incremental SAT problem $P = \langle Q_0, Q_1, \ldots, Q_n \rangle$ is a sequence of SAT *queries* where each query $Q_i = (\Delta_i, A_i)$ consists of a potentially empty set of clauses Δ_i, and a potentially empty set of assumptions A_i, where each assumption is an arbitrary literal. The first SAT query $Q_0 = (\Delta_0, A_0)$ is satisfiable if and only if $\Delta_0 \wedge A_0$ is a satisfiable formula, while for $i > 0$, Q_i is satisfiable if and only if $(\bigwedge_{j=0}^{i} \Delta_j) \wedge A_i$ is a satisfiable formula. In principle, we define incremental problems as a not decreasing set of clauses solved under different sets of assumptions. From that follows that a clause once added to the problem can not be removed later. Thus, if $(\bigwedge_{j=0}^{i} \Delta_j)$ is unsatisfiable, every later query (Q_k with $k > i$) is unsatisfiable as well, independently from the literals in A_k. Note however that clause deletion can be simulated with the help of assumptions (as it is described for example in [9]).

Proofs of SAT Solvers

Modern SAT solvers are able to provide further output and thereby let users to validate the correctness of the reported satisfiability answers. When a SAT solver claims that a formula is satisfiable, the found model serves as a certificate of the claim. When a solver finds a formula unsatisfiable, it can emit a *proof* during solving this formula. This proof then describes every relevant formula modification steps of the solver (e.g., learning new clauses, or deleting unnecessary clauses) and thereby shows how the empty clause is derivable from the given problem. Deriving the empty clause from a set of clauses (in a sound and complete proof system of propositional logic) proves that the clauses are unsatisfiable together. Though the main purpose of proofs is to certify unsatisfiable answers, it is not known upfront if a given formula is satisfiable or unsatisfiable. Thus, SAT solvers actually produce proofs in both cases. However, if the problem is satisfiable, the empty clause is not derived in the produced proof, that is, it is not a refutation.

The most common proof formats of SAT solvers are based on RUP (Reverse Unit Propagation) [17, 18] and RAT (Resolution Asymmetric Tautologies) [19] inference steps. RUP proofs are a sequence of clauses where each clause C_i is either an input clause from the user or can be derived from $\wedge_{j=0}^{i-1} C_j$ with a single RUP inference step. A clause C has RUP w.r.t. a formula F (i.e., can be derived from F with a RUP inference step) if unit propagation on the conjunction of F and the negation of C leads to a conflict. RAT proofs, beyond RUP steps, allow the derivation of clauses by RAT inference steps. A clause C has RAT w.r.t. a formula F if there is a literal $l \in C$ such that every possible resolution step on l in F derive RUP clauses. Inference steps as RUP and RAT can be used not just to add, but also to remove clauses from formulas. When a clause of a formula ($C \in F$) can be derived from the other clauses (i.e., has RUP or RAT w.r.t. $F \setminus C$), it can be removed from the problem without changing the satisfiability of F. When SAT solvers include such deletion steps in their produced RUP (resp. RAT) proofs, the proof is called DRUP (resp. DRAT).

In this paper we focus on proofs produced during incremental SAT solving. Clause learning and clause deletion steps in incremental use cases must fulfil certain additional conditions. Without going more into the details of these conditions (see [20] for these details), we will assume here that the produced proofs contain only RUP based clause addition steps and RUP or RAT inference based clause deletion steps.

Checking SAT Proofs

Checking each clause addition step in a SAT proof by an independent, simple, maybe even verified, *proof checker* provides the assurance that the solver gave a correct UNSAT answer. A RUP clause addition step can be checked based on the definition of RUP, namely, the negation of the clause is propagated over all the present clauses. If that propagation falsifies one of the clauses, the addition of the clause was correct. In case the propagation does not lead to a conflict, proof checking fails and stops with an error. Note that clause deletion steps are not checked in that process, since removing a clause can never turn an originally satisfiable problem into an unsatisfiable one. Nevertheless, clause deletion information can speed up proof checking substantially since it reduces the number of clauses that must be iterated through during unit propagation in RUP checks.

Incremental inprocessing SAT solvers actually can employ two kinds of clause deletion steps: temporary deletion and permanent deletion (also called "weaken" and "drop", for more details see again [20]). In the existing standard clausal proof formats of SAT solvers these steps can not be distinguished. Though there are possible workarounds [21], as we will see it later, our proposed proof format provides a way to address this problem too.

In general, since the size of SAT proofs can be extremely large (see [22] for an extreme example), having an efficient proof checker is a complex but important challenge. Proofs can be checked either *forward* or *backward* [17, 23]. Forward checking verifies each clause addition step in the same order as they were done by the solver, hence can be done even on-line, during solving. Backward checking verifies clause addition steps in a reverse order, starting from the empty clause. Therefore, it can start only once solving is finished, but it allows the checker to check only those clauses that are actually used in the refutation.

3 Interactions and their Proofs

The interaction between a user and a SAT solver is very limited in non-incremental use cases: the user gives a set of input (original) clauses to the solver and the solver decides if it is SAT or UNSAT. This one-time input and single query of the user is perfectly captured by DIMACS (see again Fig. 1). Further, a DIMACS file together with the proof produced by the SAT solver contains sufficient information to verify the correctness of an UNSAT answer. In incremental use cases, however, more than one satisfiability problem is asked to be solved by the same SAT solver instance. One could use DIMACS to describe each SAT query independently from each other, by repeating all the reoccurring clauses and adding assumptions as unit clauses. But doing so would transform an incremental problem sequence into a set of independent SAT problems, and thus would prevent the exploitation of incremental SAT solvers. In general, note that the total sum of the number of clauses in these independent problems grows quadratically. Further, an UNSAT answer in incremental use cases most often means unsatisfiability w.r.t. a given set of assumptions. In this situation, the produced proof of the incremental SAT solver is actually not a refutation, because the empty clause is not derived. Thus, the standard SAT proof checking techniques are not applicable on these proofs directly.

From that follows that current standard formats and techniques of SAT solvers and proof checkers are not adequate for complete proof checking in incremental use cases. Therefore, we first introduce ICNF, a new input format that exactly captures the *interaction* between user and solver that we would like to verify by proof checking. Then, we introduce IDRUP as an extension to DRUP, a commonly used SAT proof format, so that it is suitable to capture proofs of incremental SAT problems. Both our proposed input and proof format extends already established standard formats of SAT solvers and thus can be produced with minimal modifications to already existing solvers and tools.

```
<icnf>     = <comments> "p icnf\n" <lines>
<comments> = { <comment> "\n" }
<lines>    = { <comment> "\n" | <line> "\n" }
<comment>  = "c"   " " <anything-but-new-line>
<line>     = <tag> " " { <literal> " " } "0"
           | "s"   " " <status>
<tag>      = "i" | "q" | "u" | "m"
<status>   = "SATISFIABLE"
           | "UNSATISFIABLE"
           | "UNKNOWN"
<literal>  = <pos> | <neg>
<pos>      = "1" | "2" | ... | <INT_MAX>
<neg>      = "-" <pos>
```

Figure 2 Syntax of the ICNF file format. Choices are separated by vertical bars, while the application of the Kleene star operation is marked with curly brackets.

3.1 The ICNF Format

The authors in [24] introduced a file format called ICNF, where one could describe in a single file a fixed set of input clauses and multiple sets of assumptions. We refine and extend this format so that it can describe every formula-related interaction between user and solver and thereby can capture arbitrary incremental problem sequences.

The syntax of a sequence of incremental SAT queries (i.e., *interactions*) in our ICNF format is presented in Figure 2. The header line of ICNF files consists of `p icnf`, notably without the exact number of variables or clauses of the problem. Beyond the header line and comments, the file captures every relevant interaction

```
s.clause(-1, 3, 4);
s.clause(-1, 3,-4);
s.clause(-2,-3, 4);
s.clause(-2,-3,-4);
s.assume(1);              // 1 assumed
s.assume(2);              // 2 assumed
assert(s.solve() == 20);// UNSAT
assert(s.failed(1));      // 1 in core
assert(s.failed(2));      // 2 in core
s.clause(-1, 2)           // additional
s.clause( 1,-2)           // clauses
assert(s.solve() == 10);// SAT
assert(s.val(1) < 0);     // 1 is false
assert(s.val(2) < 0);     // 2 is false
s.clause( 1, 2);          // add clause
assert(s.solve() == 20);// UNSAT
```

```
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
u  1  2  0
i -1  2  0
i  1 -2  0
q  0
s SATISFIABLE
m -1 -2  0
i  1  2  0
q  0
s UNSATISFIABLE
u  0
```

Figure 3 Interaction with a SAT solver through its API (above) and the corresponding ICNF file (below). The C++ code follows the interface of CaDiCaL [25], but similar functionalities are available in the standard C API of incremental solvers (IPASIR [26]) as well.

between the user and the incremental SAT solver:

Input clauses (`"i"`): A clause addition step is expressed as in the DIMACS format (i.e., literals are encoded as integers and the end of the clause is specified by 0).

Queries (`"q"`): A query command indicates that the solver is asked to solve the current formula (all clauses that were given before this line) under the here defined (possibly empty) set of assumption literals.

Status (`"s"`): Records the answer of the SAT solver for the previous query (i.e., the last `"q"` line).

Models (`"m"`): The line represents the model given by the SAT solver after receiving a *satisfiable* answer. The model is meant to satisfy all input clauses.

Unsatisfiable cores (`"u"`): After the user receives an *unsatisfiable* answer this line lists all the failed assumptions of the last query. The SAT solver claims they form an unsatisfiable core (the input clauses are unsatisfiable when assuming these literals).

Note that the format allows to interleave clause additions and query lines (in contrast to the ICNF use in [27]). Since ICNF is meant to describe the interaction between users and SAT solvers, the order of the commands is very important and, as we will see it later, forms the base of synchronization with the produced proof of the solver. Figure 3 illustrates how different API calls of a SAT solver are expressed in the proposed ICNF format.

```
<idrup> = <comments> "p idrup\n" <lines>
...
<tag> =    "i" | "q" | "u" | "m"
         | "l" | "d" | "w" | "r"
...
```

Figure 4 Syntax of the incremental IDRUP format. Only differences to the ICNF syntax in Fig. 2 are shown.

```
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
```

Figure 5 An IDRUP proof matching the ICNF SAT solver interactions in Fig. 3. The main difference is in the header and the additional formula manipulation steps (`"l"` and `"d"` lines).

3.2 The IDRUP Format

Our proposed incremental proof format IDRUP extends ICNF in the same way as the DRUP format extends DIMACS by listing learned clauses (also called "derived" or "lemmas") and deleted clauses. A major difference is to require all the original input clauses to be repeated in the proof. In this regard the ICNF interactions file forms a subsequence of the IDRUP proof.

The syntax of IDRUP, our proposed incremental DRUP proof format, is described in Figure 4. The header of this format is "`p idrup`", similar to ICNF. Input clauses, SAT queries, solver answers, models and unsatisfiable cores are represented the exact same way as in ICNF (with tags "`i`", "`q`", "`s`", "`m`" and "`u`"). Including these lines in a proof is essential to synchronize between the incremental SAT queries and the proof.

Beyond these input and query related commands, there are additional proof tags introduced to capture the formula manipulation steps of the solver. An incremental proof describes which clauses were added to and removed from the formula, just as the proofs of non-incremental solvers. However, in contrast to standard SAT proof formats, we are more precise regarding these clause addition and deletion steps. First of all, the format supports two kinds of clause deletions steps: *drop* (indicated by prefix "`d`") and *weaken* (tagged with "`w`"). Dropped clauses are completely eliminated from the formula (and thus can be ignored during checking in following steps of the proof).

Weakened clauses are removed from the active formula as well, but they might be reintroduced in later steps, thus their deletion must be seen as temporary. The clause addition steps are also refined compared to the non-incremental SAT proof formats. There are clauses, indicated by the prefix "`l`", that are added to the problem because they are derived somehow and then learned by the solver (e.g. during conflict analysis [28] or preprocessing [29]).

Then, there are those added clauses that are reintroduced by a *restore* step (marked with tag "`r`"), i.e., by a step that is undoing a previous weakening step of the solver. Note that in non-incremental SAT problems it is guaranteed that none of the previously weakened clauses will be restored, hence in the non-incremental case there was no need to distinguish drop from weaken steps, nor learned from restored clauses. For a more formal description of these incremental-specific solver steps see [20, 21].

Continuing our example started in Figure 3, Figure 5 illustrates the corresponding incremental proof of the presented SAT queries.

4 Checking Incremental Proofs

First we describe the main steps of proof checking on an abstract level, then we provide more details about how to implement such a method. To keep things simple, we only consider *forward checking* of proofs in this paper.

4.1 Semantics

The state of proof checking can be captured on an abstract level by two multi-sets of clauses:

the *active* clauses F_A and *passive* clauses F_P,

both initially assumed to be empty. Given an IDRUP file, the following function (where $t(L_i)$ is the tag of line L_i) describes how each line L_i of it updates the current state of the proof checker.

$$(F_A^{i+1}, F_P^{i+1}) = \begin{cases} (F_A^i \cup \{L_i\}, F_P^i) & \text{if } t(L_i) = \texttt{"i"} \\ (F_A^i \cup \{L_i\}, F_P^i) & \text{if } t(L_i) = \texttt{"l"} \\ (F_A^i \setminus \{L_i\}, F_P^i) & \text{if } t(L_i) = \texttt{"d"} \\ (F_A^i \setminus \{L_i\}, F_P^i \cup \{L_i\}) & \text{if } t(L_i) = \texttt{"w"} \\ (F_A^i \cup \{L_i\}, F_P^i \setminus \{L_i\}) & \text{if } t(L_i) = \texttt{"r"} \\ (F_A^i, \quad F_P^i) & \text{otherwise.} \end{cases}$$

Input clause addition steps by the user ("`i`" lines) add a clause to the active set and need to be validated that it is indeed the next added clause in the ICNF file. Lemma addition steps ("`l`" lines) on the other hand need to be checked to have RUP w.r.t. the current active formula (F_A^i). Clause deletion ("`d`") and weakening steps ("`w`" lines) are only allowed to remove a clause if the clause is currently present in F_A^i. In the case of weakening the clause is made passive while for deletion it is just dropped.

Similarly, a restore step ("`r`" lines) can be applied only on a clause if it is currently present in the passive formula (F_P^i). Though restore steps of SAT solvers must fulfil further constraints in order to ensure that solution reconstruction works properly (see [20]), we omit any checks in this direction as we require proofs and interactions to have models satisfying the original input clauses (and not just the remaining active clauses). In any case, it must be ensured that no arbitrary clauses are introduced to the formula by these restore steps, hence the checking of the existence of weakened and restored clauses is unavoidable in incremental proofs.

Further, note that both F_A^i and F_P^i are multi-sets, i.e., in theory, the same clause can be added, removed and restored multiple times and a proof checker must be able to handle multiplicity correctly. Status lines ("`s`" lines) do not modify the state of the proof checker but serve as synchronization points between the proof and the interaction files. In case the answer is *satisfiable*, the justification of it, i.e., the following "`m`" line, defines a satisfying assignment. This assignment is checked whether it indeed satisfies not just every input clause provided until that point in the ICNF file, but also every assumption of the current query (i.e. of the most recent "`q`" line). Unsatisfiable answers are justified by a following "`u`" line, that defines a subset of the assumptions of the current query that makes the problem inconsistent.

4.2 Implementation details

There are several technical challenges and questions regarding the implementation of a proof checker for our proposed IDRUP proof format. In order to demonstrate that our approach is practically viable, we implemented and describe the IDRUP proof checker IDRUP-CHECK.

The proof checker reads and checks both the ICNF and the IDRUP file in parallel according to the state-machine shown in Fig. 6. After checking headers in both files the input clauses are read from the ICNF interactions file and matched against input lines (in the same order) in the IDRUP proof file.

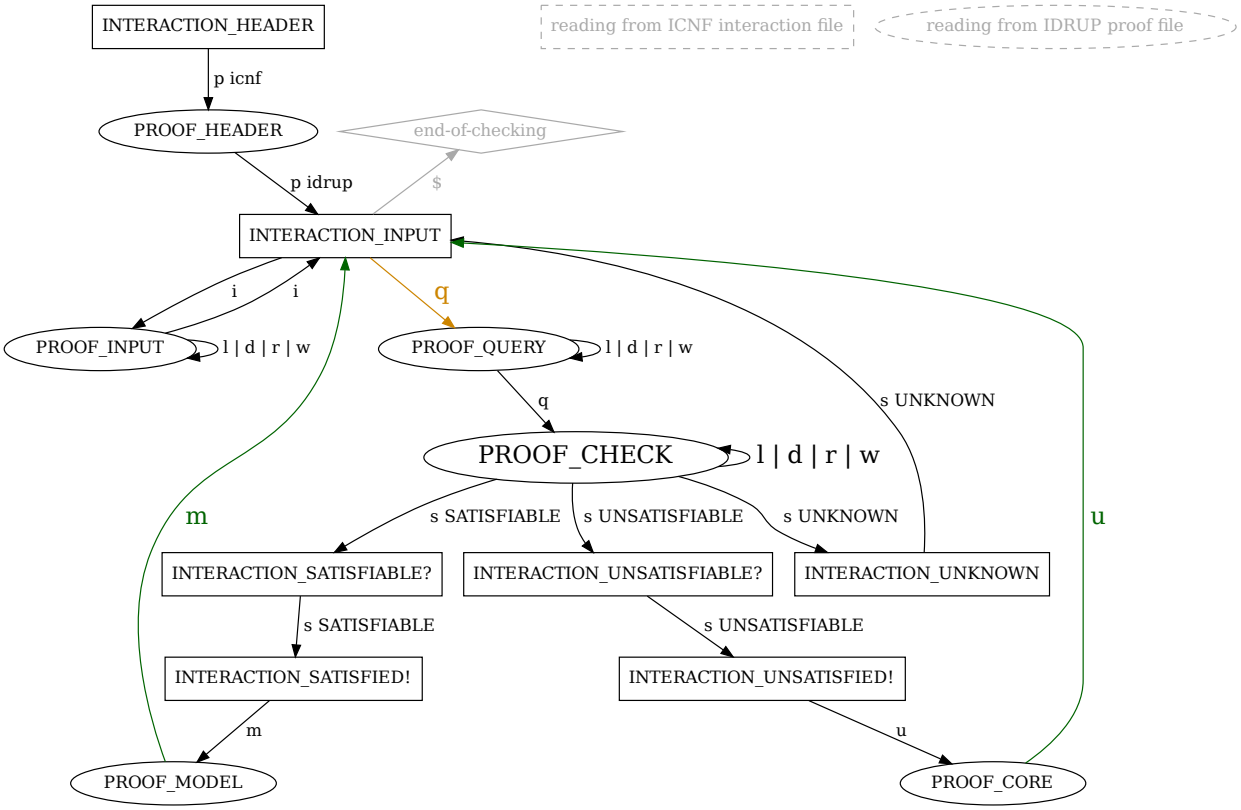

Figure 6 The state machine based implementation of the proof checker IDRUP-CHECK interleaves parsing and checking lines from the ICNF interactions file (rectangular states) and from the IDRUP proof file (oval states). If all queries in the ICNF file are justified and all implicit checks (lines match, lemmas are implied, deleted clauses exists etc.) also go through then the checker exists with success.

As input clauses can be simplified in the DIMACS parser of the SAT solver by for instance unit propagation, it is possible that the SAT solver starts producing lemmas before it has consumed all input clauses for the next query. This is taken care of by accepting for instance `"l"` lines in the `PROOF_INPUT` state. After a sequence of such `"l"`, `"d"`, `"w"` or `"r"` lines the saved input clause from the ICNF file has to occur in the IDRUP file too.

As soon a query is recorded in the ICNF file the same query with matching assumptions has to occur in the IDRUP file. This marks usually the starting point of a long sequence of lemmas and deletions generated in the CDCL loop [28] of the SAT solver interleaved with inprocessing steps [30], most prominently bounded variable elimination [31, 29], which on top of adding and deleting lemmas also weakens clauses. These weakened clauses are usually irredundant clauses which are subject to being restored later. They are moved from the active to the passive set as explained in the previous section.

In our current set-up, using CADICAL as SAT solver, restoring clauses only happens initially right after entering the `PROOF_CHECK` state. Then the restored clause is moved from the passive clause set to the active clause set. If the clause attempted to restore can not be found in the passive set IDRUP-CHECK aborts with an error message.

The active clauses are watched by a two-watched literal scheme [32] to facilitate implication checks by unit propagation. These watches are also used to look-up and find active clauses to be deleted and weakened. For passive clauses, which are not propagated, a one-watched literal scheme [33] is employed.

Implication checks are performed for lemmas as well as to justify unsatisfiable cores. These core literals are also checked to be a subset of the assumptions of the last query. Models on both side have to be consistent, i.e., not contain clashing literals, satisfy the assumptions of the last query and satisfy all the input clauses. Accordingly input clauses are saved and never deallocated even if deleted.

Proof checking is thus only successful if all lines match in both files, the implicit checks described above succeed and all queries observed in the ICNF file (the larger and golden `"q"`) have matching justifications (the larger and green `"m"` or `"u"` lines) in the IDRUP file.

5 BMC Experiments

To evaluate our proof of concept implementation of IDRUP-CHECK, we extended CAMICAL [20], a bounded hardware model checker for the AIGER format [34], which relies on incremental SAT solving, to produce ICNF interaction files. The extension prints every interaction that CAMICAL has with its SAT solver into a single ICNF file. Then, we extended the DRUP generation features of CADICAL [25], a state-of-the-art incremental SAT solver, to be able to produce IDRUP proofs. Employing this

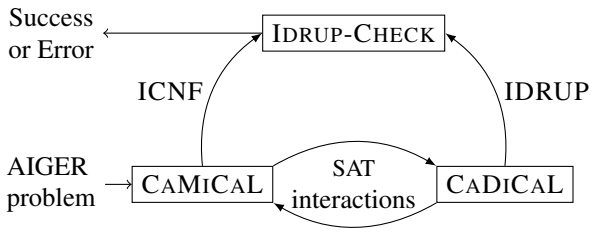

Figure 7 Our workflow to generate and check proofs during incremental SAT based bounded model checking.

extended CADICAL in the extended CAMICAL led to a workflow that is presented in Figure 7.

In our experiments we evaluated our approach on the 300 hardware model checking competition (HWMCC) benchmarks from 2017 following [20]. Our cluster made use of AMD Milan EPYC 7513 CPUs, with a space limit of 16 GB of main memory usage and a timeout of 1 000 s for both the model checker CAMICAL and separately for running the proof checker IDRUP-CHECK. We further limited the maximum checked bound to $k = 100$ steps.

Our experiments consist of at most $30\,300 = 300 \cdot 101$ incremental SAT solving calls. Some of the model checker runs however terminate early as soon a query becomes satisfiable or the time limit is reached, i.e., the running time for one instance while increasing the checked bound from $k = 0$ to a maximum of $k = 100$ reaches the time limit.

The plot on the left of Fig. 8 shows the time taken by each individual incremental calls to the SAT solver, sorted by the time taken. For each pair of benchmark and reached bound k there is one call to the SAT solver and thus one data point. The "only-solve" times are without any proof generation and "write-proof-to-dev-null" is with proof production but writing the proof to "`/dev/null`". Finally "write-proof-to-tmp" represents producing a proof and writing the interaction and proof to files in "`/tmp`" but again without proof checking. It turns out that proof production without and with writing the proof has a relatively small overhead compared to plain solving.

On the middle of Fig. 8 we compare the time for plain solving without proof generation to the time taken for proof checking, again for each bound individually. It takes around a factor of two for proof checking (actually with median 1.57) compared to plain solving, which we do consider reasonable as it is in the order expected for DRUP proofs (in the SAT competition solving time limit is 5 000 seconds while proof checking is 40 000 seconds).

Finally, the right of Fig. 8 plots the size of the produced ICNF and IDRUP files (in MB) for those instances where every step of the tool-chain successfully finished within its time limit, sorted by size of each file. While average file size of ICNF files in our experiment is 26 MB, the IDRUP files are, on average, $\sim$8.5 times larger (220 MB). Note that neither the input nor the proof files are in binary format.

6 Related Work

Existing proof formats of SAT solvers allow to capture when a clause is deleted or added, but distinguishing between deleted and weakened, or added and restored clauses is not possible. Recent work [21] addressed this problem by transforming proofs of incremental solvers with restore steps into standard DRAT proofs by *not* deleting those clauses that are later restored. This is a workable solution enabling the use of non-incremental proof checkers on proofs produced during incremental solving. But it only verifies the very last query of an incremental problem sequence and allows less clauses to be ignored during proof checking, which increases proof checking time. Our approach requires to modify the proof checker, but provides a precise set of the actually present clauses of the problem sequence at any point of time.

There is recent work on a more general proof format VeriPB [35, 36, 37, 38], which makes use of Pseudo-Boolean constraints and allows to capture reasoning steps of a very wide range of methods, including cardinality reasoning, symmetry breaking and Gaussian elimination. But since incremental SAT solvers, due to the continuous clause additions, usually do not use very advanced reasoning techniques, the benefits of such an expressive format in our context is limited. Even though VeriPB was used in an incremental setting for core-based MaxSAT solving [39] it does not support yet our generic form of incremental solving proposed in this paper. But we do believe that our solution can be applied to VeriPB too.

7 Discussion

In this paper we presented some essential first steps to establish proper standards tailored towards the incremental use cases of SAT solvers. Although preliminary results are encouraging, several further steps have to be taken to achieve a mature solution that is scalable and viable in general. Here we discuss some of the current shortcomings and limitations of our proposed approach that we are planning to address in the near future.

First of all, it is intriguing future work to evaluate the viability of our proposed certification approach in other application domains of incremental SAT solvers, such as, k-induction, interpolation or IC3 based model checkers, SMT solvers and planners. However, our goal here was to cover the most fundamental interactions between users and solvers. There are several further practical aspects of incremental solving that our approach needs to address before further evaluations can begin. For example, CADICAL allows users in incremental SAT queries to define *clause assumptions* [40] additionally to the considered literal assumptions in this paper. Such queries are convenient and practical for instance for IC3, thus relevant in solvers of model checkers.

Attaching an external *user propagator* [41] to the SAT solver is another feature of increasing interest. It allows to provide additional input constraints while solving a query without being forced to eagerly encode them up-front. In

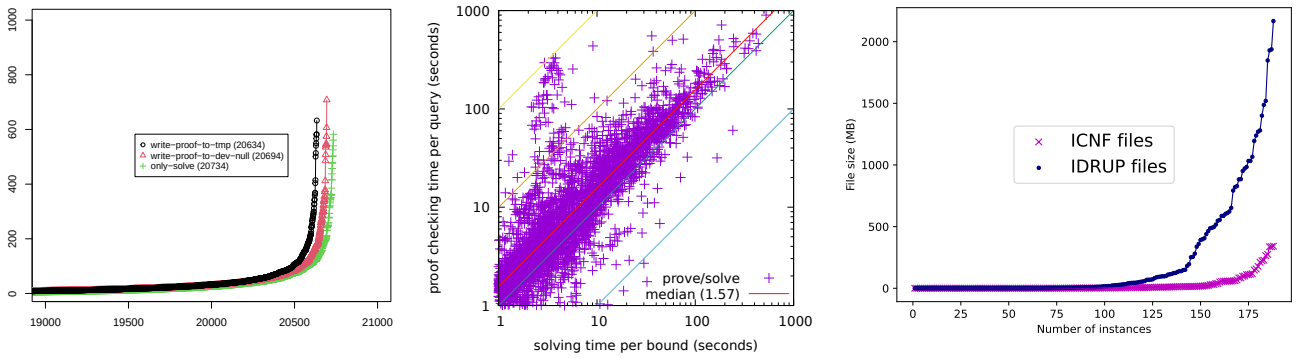

Figure 8 On the left, proof generation and writing overhead for CaMiCaL on the HWMCC 2017 benchmarks. In the middle, proof checking versus plain solving time. On the right, size of the generated ICNF and IDRUP files.

principle we could allow to have clause addition steps (`"i"` lines) between SAT queries (`"q"` lines) and their corresponding answers (`"s"` lines) and thus ICNF can capture the formula affecting interactions with an external user propagator. However, it remains future work to extend the proof checker to support these use cases.

Further, our interaction and proof representation allows to express partial models and unsatisfiable cores (depending on the status of the solver) as atomic steps, while IPASIR [26], the standard C interface of incremental SAT solvers, does not provide functions to access the model or core of a problem in a single call. Instead of requesting models and cores as a whole, IPASIR only allows to query the value of a single variable or whether it is a failed assumption. For the next upcoming IPASIR-2 version it is however already proposed to support such usage too.

Another interesting question is where to draw the line between traced and untraced user interactions. Currently we focus only on the formula to be solved, but the interface of incremental solvers usually provide further interactions, for example, freezing or melting variables or setting default phases for some of them. These details for proof checking are irrelevant, but optionally ICNF could store them. A standardized API for setting options is for instance also discussed for IPASIR-2. With such extensions, ICNF could function as a tracing language that can be used for example for debugging solvers.

As Figure 8 shows, the size of the produced incremental proofs can be substantial, which is not completely unexpected, as each incremental proof consists of a sequence of SAT proofs, certifying both SAT and UNSAT queries of the solver. Still, there is a large room for improvements in this regard in our prototype implementation, with extending IDRUP-CHECK to support proofs in binary format as an obvious first step.

Our current implementation is rather strict about the format of the ICNF and IDRUP files. More flexibility could allow to treat many details (e.g. found models or cores) as optional. Another extension would be to move from forward to backward proof checking. This opens up the possibility to trim produced proofs too. However, one potential benefit of forward checking could be the possibility to send proofs directly to an on-line checker *during* SAT solving avoiding to store the proof on disk.

Going beyond improving implementation details of IDRUP-CHECK, in [42, 43] it was shown that native support in the SAT solver of the more detailed LRAT [44] proof format reduces non-incremental proof checking time substantially. An incremental version of LRAT is expected to provide the same benefits.

The way how weakened, deleted, and restored clauses are handled during proof checking makes it evident that IDRUP-CHECK only verifies the provided answer of the solver, but not the actual steps that led to that answer. As a SAT proof checker, it is the expected functionality. An intriguing future work is to investigate how expensive would it be to extend our format and checker with more details, such that it can justify and verify all the individual steps of the solver. Such proofs (but not necessarily the refutations) could be beneficial in use cases where the SAT solver must maintain certain properties of the models of the formulas (e.g., in model counting or MaxSAT).

However, the more complicated the proof checker gets, the harder it is to trust its correctness. During the development of our workflow, we also implemented a prototypical fuzzer for IDRUP. This tool already helped us to discover several potential issues (not just in IDRUP-CHECK but even in CaDiCaL), but it requires further improvements as the implementation of IDRUP-CHECK progresses. In the very long run, we aim to have a formally verified proof checker for incremental DRUP or DRAT proofs.

8 Conclusion

Standard proof formats of SAT solvers, such as DRAT, focus on certifying a single refutation of a single unsatisfiable formula. Incremental SAT solvers, however, solve multiple satisfiable or unsatisfiable formulas and in order to increase the trust in the provided results beside checking models one needs to make sure that sources of inconsistency, i.e., provided unsatisfiable cores and failed assumptions, are justified too. In this paper we introduced a new SAT solver interaction and proof format that captures incremental SAT queries and their corresponding proof segments in a succinct way.

To demonstrate the viability and scalability properties of our proposed approach, we developed a prototype

proof checker and evaluated it in the context of bounded hardware model checking, a prime application domain of incremental SAT solvers.

Our experiments show that the format is practical and has the potential to support efficient certification, and thereby verification, of incremental use cases of modern SAT solvers. In future work our primary goals are to expand the experimental evaluation, make proof checking more efficient and general and in particular apply the approach to other contexts including certification of hardware verification problems beyond bounded model checking.

Acknowledgement

This work was supported in part by the Austrian Science Fund (FWF) under project No. T-1306, the state of Baden-Württemberg through bwHPC, the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG, and by a gift from Intel Corporation.

9 Literature

[1] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proc. IEEE*, vol. 103, no. 11, pp. 2021–2035, 2015.

[2] A. Biere and D. Kröning, "Sat-based model checking," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 277–303.

[3] S. Wieringa, "On incremental satisfiability and bounded model checking," in *Proceedings of the First International Workshop on Design and Implementation of Formal Tools and Systems, Austin, USA, November 3, 2011*, ser. CEUR Workshop Proceedings, M. K. Ganai and A. Biere, Eds., vol. 832. CEUR-WS.org, 2011.

[4] M. Ghallab, D. S. Nau, and P. Traverso, "Chapter 7: Propositional satisfiability techniques," in *Automated planning - theory and practice*. Elsevier, 2004.

[5] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 1267–1329.

[6] A. Biere, "Bounded model checking," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 739–764.

[7] A. Biere, T. van Dijk, and K. Heljanko, "Hardware model checking competition 2017," in *Formal Methods in Computer-Aided Design, FMCAD 2017, Vienna, Austria, October 02-06, 2017.*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, p. 9.

[8] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, and K. Heljanko, "Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 135–172, 2014.

[9] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," in *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, ser. Electronic Notes in Theoretical Computer Science, O. Strichman and A. Biere, Eds., vol. 89, no. 4. Elsevier, 2003, pp. 543–560.

[10] J. Whittemore, J. Kim, and K. 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*. ACM, 2001, pp. 542–545.

[11] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT competition 2016: Recent developments," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 5061–5063.

[12] E. Yu, N. Froleyks, A. Biere, and K. Heljanko, "Towards compositional hardware model checking certification," in *Proceedings 23rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'23)*, A. Nadel and K. Y. Rozier, Eds., vol. 4. TU Vienna Academic Press, 2023, pp. 44–54.

[13] ——, "Stratified certification for k-induction (extended abstract)," in *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*, A. Biere and D. Große, Eds. VDE, 2023, pp. 68–69.

[14] ——, "Stratified certification for k-induction," in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 59–64.

[15] E. Yu, A. Biere, and K. Heljanko, "Progress in certifying hardware model checking results," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12760. Springer, 2021, pp. 363–386.

[16] "SAT competition 2009: Benchmark submission guidelines," https://web.archive.org/web/20190325181937/https://www.satcompetition.org/2009/format-benchmarks2009.html, accessed: 2023-12-01.

[17] E. I. Goldberg and Y. Novikov, "Verification of proofs

of unsatisfiability for CNF formulas," in *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 2003, pp. 10 886–10 891.

[18] A. V. Gelder, "Verifying RUP proofs of propositional unsatisfiability," in *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008.

[19] M. J. H. Heule, "The DRAT format and DRAT-trim checker," *CoRR*, vol. abs/1610.06229, 2016.

[20] K. Fazekas, A. Biere, and C. Scholl, "Incremental inprocessing in SAT solving," in *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 136–154.

[21] B. Kiesl-Reiter and M. W. Whalen, "Proofs for incremental sat with inprocessing," in *FMCAD*. IEEE, 2023, pp. 132–140.

[22] M. J. H. Heule, O. Kullmann, and V. W. Marek, "Solving and verifying the boolean pythagorean triples problem via cube-and-conquer," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 228–245.

[23] M. Heule, W. A. H. Jr., and N. Wetzler, "Trimming while checking clausal proofs," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 181–188.

[24] S. Wieringa, M. Niemenmaa, and K. Heljanko, "Tarmo: A framework for parallelized bounded model checking," in *Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009*, ser. EPTCS, L. Brim and J. van de Pol, Eds., vol. 14, 2009, pp. 62–76.

[25] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[26] T. Balyo, A. Biere, M. Iser, and C. Sinz, "SAT race 2015," *Artif. Intell.*, vol. 241, pp. 45–65, 2016.

[27] M. J. H. Heule, O. Kullmann, and A. Biere, "Cube-and-conquer for satisfiability," in *Handbook of Parallel Constraint Reasoning*, Y. Hamadi and L. Sais, Eds. Springer, 2018, pp. 31–59.

[28] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 133–182.

[29] A. Biere, M. Järvisalo, and B. Kiesl, "Preprocessing in SAT solving," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 391–435.

[30] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370.

[31] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.

[32] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient conflict driven learning in boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, R. Ernst, Ed. IEEE Computer Society, 2001, pp. 279–285.

[33] L. Zhang, "On subsumption removal and on-the-fly CNF simplification," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 482–489.

[34] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 11/2, 2011.

[35] S. Gocht, R. Martins, J. Nordström, and A. Oertel, "Certified CNF translations for pseudo-boolean solving," in *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, ser. LIPIcs, K. S. Meel and O. Strichman, Eds., vol. 236. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 16:1–16:25.

[36] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström, "Certified symmetry and dominance breaking for combinatorial optimisation," in *Thirty-*

Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022. AAAI Press, 2022, pp. 3698–3707.

[37] S. Gocht, C. McCreesh, and J. Nordström, "An auditable constraint programming solver," in *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, ser. LIPIcs, C. Solnon, Ed., vol. 235. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 25:1–25:18.

[38] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström, "Certified dominance and symmetry breaking for combinatorial optimisation," *J. Artif. Intell. Res.*, vol. 77, pp. 1539–1589, 2023.

[39] J. Berg, B. Bogaerts, J. Nordström, A. Oertel, and D. Vandesande, "Certified core-guided maxsat solving," in *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, ser. Lecture Notes in Computer Science, B. Pientka and C. Tinelli, Eds., vol. 14132. Springer, 2023, pp. 1–22.

[40] N. Froleyks and A. Biere, "Single clause assumption without activation literals to speed-up IC3," in *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 72–76.

[41] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, and A. Biere, "IPASIR-UP: user propagators for CDCL," in *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, ser. LIPIcs, M. Mahajan and F. Slivovsky, Eds., vol. 271. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 8:1–8:13.

[42] F. Pollitt, M. Fleury, and A. Biere, "Efficient proof checking with LRAT in CaDiCaL (work in progress)," in *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*, A. Biere and D. Große, Eds. VDE, 2023, pp. 64–67, accepted.

[43] ——, "Faster LRAT checking than solving with cadical," in *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, ser. LIPIcs, M. Mahajan and F. Slivovsky, Eds., vol. 271. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 21:1–21:12.

[44] S. Baek, M. Carneiro, and M. J. H. Heule, "A flexible proof format for SAT solver-elaborator communication," *Log. Methods Comput. Sci.*, vol. 18, no. 2, 2022.