Bridging the Gap Between Easy Generation and Efficient Verification of Unsatisfiability Proofs

Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler *

The University of Texas at Austin

Abstract. Several proof formats have been used to verify refutations produced by satisfiability (SAT) solvers. Existing formats are either costly to check or hard to implement. This paper presents a practical approach that facilitates checking of unsatisfiability results in a time similar to proof discovery by embedding clause deletion information into clausal proofs. By exploiting this information, the proof-checking time is reduced by an order of magnitude on medium-to-hard benchmarks as compared to checking proofs using similar clausal formats. Proofs in a new format can be produced by making only minor changes to existing conflict-driven clause-learning solvers and their preprocessors, and the runtime overhead is negligible. This approach can easily be integrated into `Glucose` 2.1, the SAT 2012 challenge winner, and `SatELite`, a popular SAT-problem preprocessor.

1 Introduction

Satisfiability (SAT) solvers act as the core search engine in many tools used for bounded model checking and the verification of hardware and software. It is incumbent upon these solvers to produce the correct results. The correctness of a satisfiability model is easy to check, but most SAT solvers do not emit an unsatisfiability proof. This paper shows that with minor modifications to contemporary SAT solvers and preprocessors a proof log can be emitted that can be checked in a time similar to the solving time. Such assurance should be built into all SAT solvers.

For nearly a decade, researchers have been proposing ways to check refutations produced by SAT solvers [1,2,3,4,5,6]. Currently, the dominant SAT solver architecture is based on the *conflict-driven clause-learning* (CDCL) paradigm [7]. CDCL solvers operate by adding conflict clauses that are logically implied by the input formula. An input formula is *refuted* as soon as the empty clause (a clause with no literals) can be added. Several proof formats have been designed to express how to check that each conflict clause is logically implied by the input clauses, but limitations prevent regular use.

Ideally, a proof format should have four properties. First, the proof should be *compact*. Second, it should be possible to verify the proofs *efficiently*. Checking

* The authors are supported by DARPA contract number N66001-10-2-4087.

a proof should not be more costly than constructing a proof. Third, the implementation of the proof checker should be *simple*, and if possible, one should be able to formally verify the checker. Fourth, it should be *easy* for CDCL solvers to output refutations in the chosen proof format. Presently, only a few solvers output proofs, and there is no agreed-upon representation or checking algorithm. To increase acceptance of proof checking, complicated modifications to the code should not be required.

A new proof format is proposed in this paper to bridge the gap between the apparent trade-offs of proof size and proof complexity. Proofs in formats for which efficient and simple checking tools exist tend to be huge in size and hard to emit from CDCL solvers. Conversely, the checkers for formats that facilitate compact proofs are slow and more complicated. The crucial difference between the proposed format and existing formats is the use of *clause deletion* during proof checking. Redundant clauses, i.e., those clauses that can be removed while preserving logical equivalence, put a burden on memory usage and inference speed. This approach supports aggressive elimination of redundant clauses, analogous to clause deletion in CDCL solvers.

Clause deletion significantly decreases computational costs to check *clausal proofs*. Clausal proofs are known to be compact and easy to obtain from CDCL solvers, but have been historically costly to check. The inclusion of clausal timestamps improves the efficiency with which proofs can be checked by eliminating clauses that are no longer needed. On medium-to-hard benchmarks, the proof-checking time for proofs in the new format is an order of magnitude smaller than existing clausal proof-checking algorithms and is comparable to, though somewhat slower than, the solving time. Proofs in the new format can be emitted by `Glucose` 2.1 and its preprocessor `SatELite` with minor modifications. A mechanically verified proof-checking algorithm, written it in the `ACL2` language, demonstrates a soundness property for this proof system, and the correctness of this algorithm has been verified by the `ACL2` theorem-proving system [8].

The new proof format combined with the presented checker makes it practical to verify refutations produced by CDCL solvers. In addition, such checking is useful when debugging CDCL solver implementations and validating competition results. CDCL solvers can include this proof-checking capability, thereby increasing user confidence in SMT solvers and theorem provers that use SAT technology.

This paper proceeds by presenting some preliminary information in Section 2. Motivation for this work is provided in Section 3. In Section 4, resolution proofs and clausal proofs are shown as sound methods to add clauses that are logically implied by a formula. The process of clause deletion is discussed in Section 5, and a new proof format is proposed in Section 6. Implementation details are provided in Section 7, along with performance results in Section 8. Related work is described in Section 9 and conclusions follow in Section 10. Two appendices document the changes necessary to alter a CDCL solver to emit a proofs in the proposed format (Appendix A) and the mechanical verification of the proof-checking algorithm (Appendix B).

2 Preliminaries

To make it easier to follow the presentation, some background concepts are first discussed: conjunctive normal form (CNF), resolution, and Boolean constraint propagation.

Conjunctive Normal Form. For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a finite disjunction of literals and a CNF formula is a finite conjunction of clauses. The set of literals occurring in a CNF formula F is denoted by $\text{LIT}(F)$. A truth assignment for a CNF formula F is a function τ that maps literals $l \in \text{LIT}(F)$ to $\{\mathbf{t}, \mathbf{f}\}$. If $\tau(l) = v$, then $\tau(\bar{l}) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. Furthermore:

- A clause C is *satisfied* by assignment τ if $\tau(l) = \mathbf{t}$ for some $l \in C$.
- A clause C is *falsified* by assignment τ if $\tau(l) = \mathbf{f}$ for all $l \in C$.
- A formula F is *satisfied* by assignment τ if all $C \in F$ are satisfied by τ.
- A formula F is *falsified* by assignment τ if some $C \in F$ is falsified by τ.

A CNF formula with no satisfying assignments is called *unsatisfiable*. A clause C is *logically implied* by formula F if adding C to F does not change the set of satisfying assignments of F.

Resolution. The resolution rule states that, given two clauses $C_1 = \{x, a_1, \ldots, a_n\}$ and $C_2 = \{\bar{x}, b_1, \ldots, b_m\}$, the clause $C = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, can be inferred by resolving on variable x. C is the *resolvent* of C_1 and C_2 and this is written as $C = C_1 \diamond C_2$. Furthermore, C is logically implied by any formula containing C_1 and C_2.

Boolean Constraint Propagation. For a CNF formula F, *Boolean constraint propagation* (BCP) (or *unit propagation*) simplifies F based on unit clauses; that is, it repeats the following until fixpoint: if there is a unit clause $\{l\} \in F$, remove all clauses that contain the literal l from the set $F \setminus \{\{l\}\}$ and remove the literal $\bar{l}$ from all clauses in F. The resulting formula is referred to as $\text{BCP}(F)$. If $\{l\} \in \text{BCP}(F)$ for some unit clause $\{l\} \notin F$, then BCP *assigns* the literal l to $\mathbf{t}$ (and the literal $\bar{l}$ to $\mathbf{f}$). If $\{l\}$ and $\{\bar{l}\}$ are in $\text{BCP}(F)$ for some literal $l \in \text{LIT}(F)$ (or, equivalently, $\emptyset \in \text{BCP}(F)$), then BCP *derives a conflict*.

3 Motivation

Over time, state-of-the-art SAT solvers are becoming more complex, and the importance of their correctness is becoming more critical as they are used to answer questions in engineering, design, manufacturing, and medicine. Therefore, it is important that there is a uniform method for representing the correctness of SAT solvers that is easy to obtain, efficient to check, and covers all practical techniques.

Modern SAT solvers use many different techniques, but the CDCL [7] approach is currently the leading paradigm. The most important aspect of CDCL solvers is the addition of learned clauses. As soon as a clause becomes falsified, the solver computes a corresponding learned clause which is used to avoid revisiting that part of the search space. These learned clauses are logically implied by the input clauses. A solver refutes the input, i.e., claims that no solution exists, when it adds the empty clause. Proof formats for CDCL solvers express *how to check* that each learned clause is logically implied by the input clauses. Furthermore, CDCL solvers aggressively delete lemmas to lower the computational costs of unit propagation.

State-of-the-art CDCL solvers also use preprocessing techniques to simplify input formulas before search, and some solvers even use *inprocessing*, where preprocessing techniques are applied during search. In general, preprocessing and inprocessing techniques cannot be verified using only resolution or clausal proofs, but most of the current preprocessing and inprocessing techniques can be translated to resolution and clausal proofs.

Much of the growing complexity of SAT solvers comes from various preprocessing and inprocessing techniques. Some techniques even go beyond resolution; for example, they perform blocked clause elimination (BCE) [9]. However, there is a clear difference between techniques that *weaken* the input formula (by removing clauses or adding literals to clauses) and those that *strengthen* the input formula (by adding clauses or removing literals from clauses). All weakening techniques, such as BCE, do not interfere with refutations. If one can prove the unsatisfiability of a weakened formula, the same proof applies to the original formula. All non-weakening techniques used by current state-of-the-art CDCL solvers are based on resolution, such as variable elimination and self-subsumption [10]. These techniques can be expressed using resolution and clausal proofs.

The CDCL paradigm and current preprocessing and inprocessing techniques can be expressed as either performing *clause addition* and/or *clause deletion* operations. In Section 4, different methods for clause addition with respect to proofs of unsatisfiability are presented. In Section 5, the effects of clause deletion on a proof are discussed.

4 Clause Addition

Concerning the addition of clauses, two approaches are presented: one for resolution proofs and one for clausal proofs. Recall that *lemmas* are often used to construct a proof of a theorem in mathematics. Here, lemmas are represented as learned clauses and the "theorem" is that the formula is unsatisfiable. From now on, the term *clause* is used to refer to an input clause, while *lemma* will refer to a learned clause.

4.1 Resolution proofs

Early approaches to verify refutations produced by SAT solvers were based on resolution [1]. The lemmas computed by CDCL solvers can be simulated by a sequence of resolutions [11]. Let L be a lemma and $\{C_1, \ldots, C_m\}$ be the input clauses. For each lemma L there exists a sequence of resolutions such that $L = (((C_i \diamond C_j) \ldots) \diamond C_k)$. In practice, this sequence may also use previously added lemmas to construct a new lemma.

Resolution proofs can be checked efficiently, and since resolution is such an elementary operation, simple checking algorithms exist [1,3,12]. There are clear disadvantages, however. Resolution proofs can be huge (see Section 4.3). It is also hard to modify a SAT solver to emit a resolution refutation; for instance, each clause and lemma needs a unique identifier for the entire search. In some cases, clauses or lemmas may not have a unique identifier, and in others, identifiers may change during garbage collection. Since resolution is not associative, the order of the clauses in the resolution sequence is crucial. The difficulty to compute this order lies in how the generation of lemmas is implemented.

It is hard to overemphasize the seriousness of this last problem. At the moment, there exist only four SAT solvers that output resolution proofs: `zChaff` [1], `Minisat` [13], `Picosat` [14], and `Booleforce` [15]. These solvers all use different formats, and the proof-logging versions for the first two solvers are outdated. Even for the author(s) of a SAT solver, modifying the solver to emit resolution proofs is not an easy task. After the integration of a portfolio of SAT solvers into an SMT solver or into a theorem prover, it would be a daunting task to enhance them all to emit resolution proofs.

4.2 Clausal proofs

An alternative approach was proposed by Goldberg and Novikov [2]. They introduced *clausal proofs*, also known as Reverse Unit Propagation (RUP) proofs [3]. The key insight regarding clausal proofs is that each lemma L learned by SAT solvers can be checked using BCP. Lemmas, like clauses, are disjunctions of literals. Let $\overline{L}$ denote the set of unit clauses that falsify all literals in a lemma L. If BCP$(F \cup \overline{L})$ results in a conflict, i.e., produces the empty clause $\emptyset$, then L is implied by F.

Clausal proofs are represented as a queue of lemmas $(L_1, \ldots, L_m)$ such that $L_m = \emptyset$. Given a CNF formula F, a clausal (or RUP) proof of F consists of lemmas L_i that are logically implied by F. Let $F_0 := F$ and $F_i := F_{i-1} \cup \{L_i\}$. To check that L_i is logically implied by F, it should hold that if all literals $l \in L_i$ are assigned to $\mathbf{f}$, then BCP on F_{i-1} results in a conflict.

The elegance of clausal proofs is that they can be expressed in conjunctive normal form, although the order of the lemmas matters. Clausal proofs are significantly smaller, as compared to resolution proofs, and only minor modifications of a SAT solver are required to output these proofs. However, because of unit propagation, checking of clausal proofs can be quite expensive, especially for large proofs. Checking algorithms for clausal proofs are typically more complex

than those for resolution proofs, making it harder to trust or prove correctness of the algorithm.

```
RUPchecker (CNF formula F, CNF queue Q)
1   while Q is not empty
2     L := Q.pop()
3     F' := BCP(F ∪ ¬L)
4     if ∅ ∉ F' then return "checking failed"
5     F := BCP(F ∪ L)
6     if ∅ ∈ F then return "unsatisfiable"
7   return "all lemmas validated"

BCP (CNF formula F)
11  while ∃ {l} ∈ F do
12    for C ∈ F with l ∈ C do
13      F := F \ {C}
14    for C ∈ F with ¬l ∈ C do
15      C := C \ {¬l}
16  return F
```

Fig. 1. Pseudo-code to check (clausal) Reverse Unit Propagation (RUP) proofs.

Fig. 1 shows the pseudo-code of a clausal proof checking algorithm. The input is a CNF formula F and a CNF queue Q of lemmas representing a refutation of F. Lemmas are sorted in chronological order as they are learned by a SAT solver. While Q is not empty (line 1), its front lemma L is popped (line 2). If unit propagation on F using $\overline{L}$ does not derive a conflict, then terminate because L is not logically implied by F and (lines 3 and 4). Otherwise, L is added to F. In the case that L was unit, the new F is simplified using BCP (line 5). If that simplification results in a conflict, a top level contradiction is found meaning that the formula is unsatisfiable (line 6). If the algorithm reaches the end (line 7), all lemmas in Q are validated but no top-level conflict was encountered.

Goldberg and Novikov [2] suggested checking lemmas in reverse order so that some lemmas may be skipped, but this process uses conflict analysis which complicates the proof checker. It is possible to implement a clausal checker that validates lemmas in reverse order [16]. A tool called `DRUPtrim` was developed in order to validate proofs as efficiently as possible at the cost of being larger and more complicated. The `DRUPtrim` tool uses the main contributions described in this paper to realize fast performance and was used to validate the results of the SAT 2013 Competition. However, in the remainder of the paper, the presentation concerns checking lemmas in the order they are learned because the resulting algorithm is simpler.

4.3 Comparison

For medium-sized and larger problems, resolution proofs can be quite large and it is hard to modify solvers to emit them, but the resulting proofs can be checked with simple, efficient algorithms. Such proofs can require hundreds of gigabytes of disk space, and the creation of such large proof files often dramatically increases the solving time because of the sheer amount of data that needs to be written to and read from disk. However, checking resolution proofs is relatively fast. In fact, most of the checking time is spent reading the proof. A critical disadvantage

of resolution proofs is the difficulty to extract them from state-of-the-art SAT solvers. As solvers have become more complex, these difficulties continue to increase.

Clausal proofs are compact and easy to obtain, but they are computationally more expensive and complex to check. For these proofs, the lack of speed in existing checking algorithms has traditionally been the major drawback. However, this paper presents a fast, clausal, proof-checking algorithm in less than 200 lines of C code (see Section 7.1). Alternatively, one could transform a clausal proof into a resolution proof, although this further increases the checking costs [3].

Experience has shown that clausal proofs are easier to emit than resolution proofs. To illustrate the dynamics between the two approaches consider the *clause minimization* [17] technique. After a SAT solver computes a lemma L, clause minimization tries to eliminate literals from L. This technique results in shorter lemmas which decreases the number of lemmas necessary for a clausal proof while increasing the number of unit propagations during clausal proof checking. However, eliminating literals requires additional (up to the number of variables) resolution steps. Hence, the length of lemmas in resolution proofs increases when clause minimization is used [18]. Additionally, to emit resolution proofs, it is necessary to identify the clauses and lemmas on which to resolve and compute the order in which to apply resolution steps.

Resolution-style proof checkers are generally considered straightforward to implement, but their code size is typically several hundreds of lines of code or larger. In contrast, a clausal proof-checking algorithm might be considered somewhat more complex, but can be implemented in less than 200 lines. One difficulty in claiming that a clausal proof checker is "simple" concerns the use of a watch-pointer data structure, which is critical for unit-propagation performance; this code increases the complexity of the proof-checking algorithms and, therefore, may reduce confidence in the soundness of the checker. Mechanically-verified proof checkers exist for the resolution proof formats [19,20,12,21]. Through use of the `ACL2` theorem-proving system, it is possible to mechanically verify a RUP checking algorithm (see Appendix B). It is also possible to mechanically verify a generalization of the RUP proof format based on satisfiability equivalence [22].

5 Clause deletion

One of the most crucial techniques in SAT solvers is the 2-watch-literal, or *watch-pointer*, data structure [23]. Due to this data structure, only a fraction of clauses are examined during propagation.

When checking clausal proofs, performance increases when both watch-pointer and clause deletion techniques are used. The watch-pointer data structures can be integrated relatively easily, although it makes the checking algorithm more complex. However, clause deletion is not currently used by any proof format. Clausal proofs contain only a few hints about which clauses are redundant. Persistence of lemmas is the most important reason why larger proofs are expensive to check.

An exploration of what information in clausal proofs can be used to support deletion of clauses, without including additional information in the proof, suggested that efficiency could be improved by deleting subsumed clauses and lemmas. After the addition of a new lemma, one can checked which clauses and/or lemmas this new lemma subsumes (i.e., clauses or lemmas which are a superset of the new lemma) and later delete them (when they are no longer needed). Unit and binary lemmas frequently subsume several clauses and lemmas. However, most lemmas of length three or more rarely subsume other lemmas. Subsumption information speeds up verification; preliminary results indicated a 20% improvement.

In order to substantially lower the checking costs, the clause deletion information of SAT solvers needs to be included in the proof. The main advantage of clausal proofs is that SAT solvers can output them with only a few additional lines of code. That also holds for clause deletion information. There are several approaches to implement this. A straight-forward way is to interleave the addition and the deletion of lemmas (both as a set of literals) by using a flag to denote addition or deletion. The order in the emitted proof provides the information about when lemmas can be deleted.

Several alternative deletion-oriented implementations are possible, but they may depend on information about lemmas that is not available. When each clause and lemma has a unique identifier, this identifier could be combined with added lemmas. In that case, clause deletion can be expressed using these identifiers. Such an alternative does not require matching add and delete entries in the proof. The above straight-forward proposal is somewhat more costly, but is at most twice the size of conventional clausal proofs.

6 New Proof Formats

The structure of clausal proof formats can effect the ease of implementation and the size of the proof to be checked. Two new proof formats are presented in this section. The first format is designed to minimize the modifications to SAT solvers, while exporting all information required to ensure efficient checking. The second format is a generally more compact variant designed to reduce the complexity needed to check a proof. With a small program, it is possible to convert the first format to the second one. Fig. 2 shows the details of the conventional clausal proof format RUP [3] and two new formats for a small unsatisfiable formula.

The first new format is called *delete-reverse-unit-propagation* (DRUP). This format is designed to make it easy to output added and deleted lemmas into a standardized form. Proofs in DRUP simply list the literals of each lemma (both for addition and deletion entries). Lemma deletion entries are preceded with a "`d`" identifier. This information can easily be generated from any modern SAT solver. Appendix A shows the minor modifications required to output the DRUP format for the `Glucose` solver and the `SatELite` preprocessor.

CNF formula

```
p cnf 3 5
 1  2 0
-1  2 0
 1 -2 0
-1  3 0
-2 -3 0
```

RUP proof

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

DRUP proof

```
   -1 -2 0
d  -1  3 0
d  -2 -3 0
    1  0
d   1  2 0
d   1 -2 0
    0
```

IORUP proof

```
iorup 3 5
1 7    1  2 0
2 8   -1  2 0
3 7    1 -2 0
4 6   -1  3 0
5 6   -2 -3 0
6 8   -1 -2 0
7 8    1  0
8 8    0
```

Fig. 2. An example of a CNF problem in the typical DIMACS format (left) as well as equivalent clausal proofs in the RUP (middle left), DRUP (middle right) and IORUP (right) formats. For the CNF problem and IORUP format, the numbers in the first line denote the number of variables (first) and the number of clauses (second). Spaces between numbers can be arbitrary long. Spacing in the example is to improve readability. A `0` marks the end of clauses and lemmas. For each lemma line in RUP, there is a corresponding line in DRUP. Additionally, the DRUP format contains some `d` (delete) lines. Compared to the RUP format, the IORUP format has two additional numbers on each line denoting the 'in' and 'out' time-stamps of the corresponding clause or lemma. The RUP and DRUP formats are appended to the input formula, but the IORUP format modifies the input formula to contain time-stamps.

The *in-out-reverse-unit-propagation* or IORUP format contains two time-stamps for each clause and lemma. The 'in' time-stamp denotes the time at which the lemma becomes active (all clauses are active before the first lemma). These 'in' time-stamps are strictly increasing and are used to determine which other clauses become inactive. The checker maintains a global time-stamp that is set to the 'in' time-stamp of the last processed lemma. The 'out' time-stamp of a clause or lemma denotes when it becomes inactive. A clause or lemma will be inactive when its 'out' time-stamp is smaller than the global time-stamp. Inactive clauses and lemmas can be ignored for the remainder of the proof.

Clausal proofs in the DRUP format can cheaply be converted into the IORUP format. Solvers can first emit the refutation in the DRUP format, and then convert that into the IORUP format, and finally check the latter.

There is little difference between proof validation in RUP and IORUP formats. Two changes are required to exploit the information in the IORUP format. The 'out' time-stamp has to be stored with each clause and lemma. And, while examining clauses and lemmas during unit propagation, the corresponding 'out' time-stamps need to be checked. If the 'out' time-stamp is larger than the global time-stamp, then the corresponding watch-pointers are deleted (and the literals of the clause or lemma will not be checked).

One might note that the 'in' time-stamp is not necessary during proof validation. A global time-stamp that is increased after each lemma will suffice. The addition of explicit 'in' time-stamps improves readability, which is important

when "debugging" proofs. Furthermore, the addition of 'in' time-stamps does not significantly increase the size of proofs. In the worst case, every lemma is binary and every line increases from three integers (two literals and a terminating zero) to five integers (the time-stamps, two literals, and a terminating zero) — roughly doubling the file-size. The general case is much better as an average conflict clause (for existing application benchmarks) contains 40 or more literals. Overall, the 'in' time-stamps only account for small increase in file size.

The remainder of this paper is focused on validating proofs in the IORUP format because this format facilitates a small and efficient checker implementation. However, there are situations in which the DRUP format is preferred. Recall that the set of original clauses (together with time-stamps) is present in IORUP proofs. The act of adding time-stamps for the original formula can introduce inconsistencies. In some settings, such as validating the results of the SAT 2013 Competition, this might be exploited by modifying the original formula. Modification of the input formula is not a problem for validating proofs in the DRUP format as the lemmas are expressed as a separate input.

6.1 Preprocessing validation

State-of-the-art CDCL solvers use input preprocessors, but there is little research on proof-checking of these solvers. Clausal proof checking can also be used to validate results produced by SAT preprocessors. Checking a proof log produced by preprocessing tools is a little different than checking a refutation. When checking a refutation using a RUP checker (recall Fig. 1), it is expected it to return "unsatisfiable". When checking a proof by a preprocessor, the expected outcome is "all lemmas validated", meaning that all added lemmas are logically implied by the input clauses.

As discussed in Section 3, the strengthening techniques used in preprocessors are based on resolution and can therefore be validated using clausal and resolution proofs. While SAT solving procedures only add and remove lemmas, some preprocessing techniques also remove literals from clauses and lemmas. Literal elimination can easily be simulated by adding and removing clauses: one first adds the shortened clause and then removes the original clause.

7 Implementation

The result of this work produced three tools for proofs for the RUP, DRUP, and IORUP formats.[1] The first program is a compact, fast implementation for checking RUP proofs. The second program converts DRUP proofs into IORUP proofs. The third program is a modified version of the first program and is used to check proofs in the IORUP format. At the time of submission, the only publicly-available tool to check RUP proofs first converts clausal proofs into resolution proofs and checks the resolution proof using a RES checker [3].

[1] The tools are available at `http://www.cs.utexas.edu/~marijn/IORUP/`.

7.1 RUP checker

The new proof checkers, implemented in C, for the RUP and the IORUP formats are compact (less than 200 lines of C code); this makes it relatively "easy" to visually inspect the code for correctness. The implementation of the RUP checker focuses on compactness while compromising as little efficiency and readability as possible.

To realize efficiency, the watch-pointer data structure [23] was used. This data structure only examines whether a clause or lemma is unit when one of its first two literals becomes falsified. In contrast to most CDCL solvers, the two watch-pointers are stored in memory next to the first two literals. Using this structure, no dynamic arrays are required to store the watch-pointers. Memory is only allocated during initialization and without the use of libraries (no malloc).

The core of the code, the BCP implementation, is just over 20 lines, and each line is documented. Most of the other lines are related to parsing. As discussed in Section 4.3, checking clausal proofs is considered more complex when compared to checking resolution proofs. Yet existing resolution proof checkers are several hundreds of lines of code. This is caused by applying some tricks to deal with huge files efficiently. Although the size of the code is not a very good measurement of its complexity, one could argue that the RUP checker is not that much harder to visually inspect for correctness than existing resolution proof checkers.

7.2 DRUP to IORUP converter

The second program implemented converts DRUP proofs into IORUP proofs. This small program merely parses and matches the 'add' and 'delete' entries. The DRUP format does not prescribe the order of the literals in clauses and lemmas. Hence, the order of the literals between matching 'add' and 'delete' entries might be different. In practice, the order is rarely the same because the watch-pointer data structure approach frequently shuffles literals within clauses and lemmas. One possible solution is to sort all entries, but this is rather expensive. Instead, a hash table was used to match 'add' and 'delete' entries. The hash function uses three statistics for each lemma that are independent of the order of the literals: the *sum*, the *product*, and the *xor* of its literals. For a given lemma L a hash function computes a value as follows (with *size* denoting the size of the table):

$$hash(L) := (1023 \cdot sum(L) + product(L) \oplus (31 \cdot xor(L))) \pmod{size} \quad (1)$$

Collisions are addressed by adding linked lists to implement separate-chaining collision resolution. In order to avoid collisions, the size of the table is 10 times the number of lines without `d` entries. Using this approach, the conversion costs are mostly caused by parsing the input file and writing the output file to disk. The conversion tool adds typically 5% overhead to checking a IORUP proof.

7.3 IORUP checker

The third, and most important tool, is a checking algorithm for IORUP proofs. The algorithm closely follows the RUP checking algorithm and contains about

30 additional lines of C code. The IORUP checker stores in memory the 'out' time-stamp with the two watch-pointers for each clause and lemma.

The checker also maintains a global time-stamp which is equal to the 'in' time-stamp of the lemma currently being checked. The BCP algorithm is changed in such a way that for each examined clause or lemma, the corresponding 'out' time-stamp is first obtained. If this 'out' time-stamp is larger than the global time-stamp, the watch-pointers to that clause are permanently removed. These modifications represent about 15 lines of code.

Notice that only the watch-pointers to the clauses are removed, while the corresponding clauses are kept in memory. The checker becomes about 25% faster by periodically removing all 'out' clauses from the data structures — this includes clauses satisfied by top level unit clauses. This feature makes the checking algorithm somewhat more complex, so it can be argued whether the performance gain is worth it. No matter, this feature is used throughout all experiments.

8 Evaluation

In this section, the three tools presented above are evaluated. The `Glucose` 2.1 SAT solver was modified — the winner of the SAT 2012 challenge[2] — to emit proofs in the DRUP format (see Appendix A for the changes). Recall that the DRUP format can easily be converted to the RUP and IORUP formats. Four checking tools were used during the evaluation. There are two tools written by Van Gelder, which might be considered to be the official RUP checker. The `RUPtoRES` tool first converts a RUP proof into a RES proof. Then, the `Checker3` tool checks the resulting RES proof. Those tools are compared with the checkers presented here: one for RUP (see Section 7.1) and one for IORUP (see Section 7.3).

Two benchmark suites of unsatisfiable application instances were constructed. The first suite consists of medium-to-hard benchmarks from the SAT 2012 challenge. The second suite consists of benchmarks frequently used in papers that concern the checking of unsatisfiability proofs. Instances from the second suite are mostly easy benchmarks, but they are included to aid comparison with earlier work. Because `Glucose` 2.1 uses the `SatELite` [10] preprocessor, `SatELite` was first applied to all formulas to have a clear comparison with the SAT 2012 challenge results. Verification of the preprocessing is discarded while checking the SAT solving for two reasons: the preprocessing and its checking are only a small fraction of the total costs, and the RUP format does not support clause deletion. Therefore, verification of a combined preprocessing and solving run would require that all clauses removed by the preprocessor are present during the solving phase; this would significantly degrade the performance. Results on preprocessing are further discussed in Section 8.1. These experiments were performed on a 4-core Intel Xeon CPU E31280 3.50GHz, 32 Gb RAM machine running Ubuntu 10.04. Because the `RUPtoRES` tool created huge files and storage resources were limited, the size of the output files was restricted to 100 Gb.

[2] `http://baldur.iti.kit.edu/SAT-Challenge-2012/`

Table 1. Comparison of the runtime (in seconds) on application benchmarks: *solving* an instance with `Glucose` 2.1; converting a clausal proof using `RUPtoRES` by Van Gelder; verifying a RES proof using `Checker3` by Van Gelder; and verifying the RUP proof and corresponding IORUP proof using the checkers described in this paper. $|V|$, $|C|$, and $|L|$ denote the number (in thousands) of variables, clauses and lemmas, respectively. OoS (Out of Space) means that `RUPtoRES` hit the storage limit of 100 Gb. Consequently, some RES proofs could not be checked (denoted by —). The top section of the table shows SAT 2012 challenge instances, while the bottom section shows benchmarks frequently used in papers regarding proof checking.

benchmark	$\|V\|$	$\|C\|$	$\|L\|$	*solving*	`RUPtoRES`	RES	RUP	IORUP
`aes-bottom12`	6.7	37	479	37.12	573.36	100.52	242.54	67.75
`aes-bottom13`	7.8	43	3,403	402.53	15,588.20	1,422.71	9,613.47	875.28
`AProVE07-02`	2.3	13	3,189	231.41	OoS	—	5,618.84	497.41
`AProVE07-08`	2.5	13	1,469	147.21	3,231.11	671.26	2,587.83	308.38
`AProVE07-27`	3.6	20	2,741	401.26	OoS	—	12,098.36	1,174.86
`eq.atree.09`	0.6	3	412	28.13	432.34	46.25	375.24	53.24
`eq.atree.10`	0.8	3	1,349	133.45	4,393.31	173.10	4,772.71	334.61
`eq.atree.11`	1.0	4	3,289	467.47	OoS	—	26,452.72	1,411.61
`gus-md5-08`	28.5	163	102	177.41	OoS	—	157.38	126.62
`gus-md5-09`	28.5	163	309	441.01	OoS	—	591.42	361.71
`gus-md5-10`	28.5	163	1,406	1,860.24	OoS	—	6,421.71	1,503.25
`maxor064`	18.6	86	858	125.49	2,184.15	1,033.26	512.07	169.23
`maxxor032`	4.7	22	924	97.35	2,069.98	550.59	815.86	158.00
`maxxororand032`	6.6	35	1,262	231.68	OoS	—	3,086.81	395.51
`q_query_3_l44`	7.5	65	1,009	74.93	1,537.70	42.17	1,282.81	148.79
`q_query_3_l45`	7.7	67	999	75.84	1,576.52	39.23	1,344.34	148.58
`q_query_3_l46`	7.9	69	985	72.88	1,426.51	36.07	1,208.62	141.70
`rbcl_xits_07`	0.6	57	1,068	100.52	3,053.81	40.83	2,270.79	216.37
`c5315`	1.5	7	17	0.32	2.74	0.33	0.20	0.21
`c7552`	1.7	8	26	0.51	1.58	0.62	0.44	0.43
`fifo8_200`	14.6	68	39	1.29	1.98	0.46	0.55	0.56
`fifo8_300`	22.5	105	123	7.34	16.78	6.58	4.40	4.09
`fifo8_400`	30.5	141	134	7.92	11.56	2.36	3.41	3.29
`longmult13`	1.4	8	304	8.25	112.78	75.64	27.09	15.54
`longmult14`	1.6	9	129	6.67	106.95	77.62	19.54	11.56
`longmult15`	1.8	10	111	5.78	68.12	44.60	15.38	10.16
`w10_45`	4.4	23	2	0.09	0.30	0.05	0.06	0.06
`w10_60`	7.2	38	8	0.43	2.05	1.40	0.38	0.39
`w10_70`	9.0	48	26	1.01	7.17	6.12	0.93	0.95

Table 1 shows the result of running the tools on both suites. For all tools the real time in seconds is listed. While `RUPtoRES` and the RUP and IORUP tools were using 100% of the CPU, the `Checker3` tool for the RES proof was slowed by file I/O and used, on average, 20% of the CPU. The top section shows the first suite. On nearly half of those instances, the `RUPtoRES` hit the space limit

of 100 Gb, demonstrating one of the main weaknesses of resolution proofs. On the other instances, the performance of RUPtoRES is comparable with the RUP checker. Notice that for the approach from RUP to RES, the cost is the sum of the RUPtoRES and RES columns.

For medium-to-hard benchmarks, IORUP clearly outperforms RUP. The extent of the performance gain on an instance is related to the size of the proof: the larger the proof, the larger the speed-up. For most medium-to-hard benchmarks, checking the IORUP proofs is an order of magnitude faster when compared to checking the RUP format. Moreover, the cost of checking IORUP proofs remains comparable with the solving time, although about a factor two slower on average. Comparing the solving time and the checking time for RUP proofs shows a growing gap as the solving time increases. It is impractical to use RUP proofs for hard benchmarks — in contrast to IORUP proofs. Checking proofs in the RUP format requires that all lemmas be in memory during the validation; therefore, the cost of proof checking increases.

The bottom section of Table 1 shows the results on benchmarks frequently used in papers [1,2,3,20] regarding proof checking. These benchmarks are easy for modern SAT solvers as Glucose 2.1 can solve each of them within 10 seconds. Papers on RUP checking [2,3] report solving times and RUP checking times on several of these instances of over 1,000 seconds. Notice that the number of variables and clauses may differ from other papers, because all benchmarks have been preprocessed by SatELite.

A comparison of the size of refutations in different proof formats is shown in Table 2. The smallest proof format is RUP. Proofs in the DRUP format are generally twice as large as compared to RUP, while IORUP proofs are only 10% larger. Clausal proofs are significantly smaller, typically by two orders of magnitude, than resolution proofs.

Table 2. Comparison between the proof sizes (in Mb) on some application benchmarks solved by Glucose 2.1. The solver emits DRUP proofs, which are then converted to the RUP format by removing the d lines and to the IORUP format using the converter presented in Section 7.2. Proofs in the RES format were obtained by applying the RUPtoRES tool on RUP proofs.

benchmark	RES	RUP	DRUP	IORUP
aes-bottom12	9,831.71	70.99	141.04	78.46
aes-bottom13	65,703.17	521.54	1,044.81	573.04
AProVE07-08	51,386.24	238.61	479.19	259.47
eq.atree.09	4,531.55	31.27	61.94	36.67
eq.atree.10	16,549.82	113.76	227.87	132.47
maxor064	62,010.42	246.32	478.61	260.53
maxxor032	46,831.80	289.96	574.58	302.83
q_query_3_l44	3,949.44	121.79	243.14	137.55
q_query_3_l45	3,807.54	118.51	237.99	134.20
q_query_3_l46	3,554.44	117.81	236.91	133.35
rbcl_xits_07	3,747.44	79.17	158.56	95.46

8.1 Preprocessing results

As discussed in Section 6, clausal proof-checking techniques can be used for preprocessing as well. Presently, `SatELite` [10] is the most widely used preprocessor. Modifying `SatELite` to emit a proof in the DRUP format was only slightly harder than modifying the `Glucose` 2.1 solver: seven lines of code (see Appendix A) are required. `SatELite` implements removal of literals in a way that fits within the DRUP format as a shortened clause is added and the original clause is deleted.

Preprocessing techniques can typically consume a large amount of computation. `SatElite` is optimized to keep the cost of preprocessing low by avoiding expensive techniques. Regarding the SAT 2012 challenge suite, `SatELite` finishes in less than a second on half the instances, and terminates on almost all instances in less than ten seconds.

Table 3. Comparison of the preprocessing time (in seconds) by `SatELite`, the proof-checking time of RUP proofs, and the proof-checking time of the IORUP proofs. $|V|$, $|C|$, and $|L|$ denote the number of variables, clauses and lemmas, respectively.

benchmark	$\|V\|$	$\|C\|$	$\|L\|$	*preproc.*	RUP	IORUP
`aigs-lfsr_008_079_112-t`	448,370	1,130,525	5,205,849	227.29	136.13	50.65
`clauses-6`	683,996	2,623,082	1,412,364	39.49	4.79	4.89
`safe-50-h50-sat`	633,392	2,141,556	6,925,757	147.61	295.46	81.59

In order to show the effectiveness of clausal proof formats in checking preprocessing techniques, the three most costly (in terms of `SatELite` runtime) benchmarks were selected from the SAT 2012 challenge. The results are shown in Table 3. For these and other benchmarks, checking IORUP proofs requires less time than the preprocessing time. Checking RUP proofs may be more costly than preprocessing — as shown for `safe-50-h50-sat` in the table. In general, checking preprocessing tends to take less time than checking solving procedures.

9 Related Work

9.1 Related Tools

At the time of this article, only one proof checker based on the RUP format is publicly available. This checker, `RUPtoRES` by Van Gelder [3], converts RUP proofs into resolution (RES) proofs, which in turn are validated by a RES checker. In Section 8, this approach was compared to the RUP and IORUP checker implementations.

Four tools have been developed to produce resolution proofs: `zChaff` [1], `Minisat` 1.14 [13], `Picosat` [14] and `Booleforce` [15]. Unfortunately, these tools use different resolution proof formats, although all except `Minisat` have the option to output to the official RES format. The solvers that support the RES

format also participated in the certified UNSAT track of the SAT 2007 competition [24]. The first two tools, `zChaff` and `Minisat` 1.14, have not been updated recently[3]; therefore, these tools lack the recent improvements of modern SAT solvers. Consequently, the performance of these tools is not competitive when compared to most SAT 2012 challenge participants.

`Picosat` is the only state-of-the-art tool that can output resolution proofs. Proofs are emitted in the tracecheck format [15] which is an alternative resolution proof format for which one can optionally add clausal proof information. Validating proofs in the tracecheck format is very fast. Unfortunately, there is no tool that can convert RUP proofs into tracecheck proofs. However, one can easily convert tracecheck proofs with the additional clause proof information into IORUP proofs. In the tracecheck format, each lemma stores all of the clauses that are required to construct it using resolution. The clauses can be used to compute the (perfect) 'out' time-stamp for each lemma by determining when each clause will be last used. Using this method, tracecheck proofs can be converted into IORUP proofs.

In general, validating tracecheck proofs is about a factor of four faster than checking the corresponding IORUP proof. Checking IORUP proofs is slower because it performs many unit propagations that are not required to derive the conflict. In essence, the tracecheck format exchanges time for space by explicitly describing which unit clauses the checker should examine and in what order. However, extracting this information from a SAT solver is essentially the same as building a resolution proof.

It will be hard to beat the runtime of checking algorithms for resolution proofs, but runtime is not the bottleneck when it comes to validating refutations produced by SAT solvers. Currently the biggest obstacle is the availability of solvers producing these proofs.

9.2 Verified Resolution Proof Checking

Weber [19,20] demonstrated the first mechanically-verified resolution-based proof checker using `Isabelle/HOL`, evaluated the verified proof checker with resolution proofs produced by `zChaff`, and compared the results to the built-in `zChaff` proof checker using CPU times (not wall-clock times). The verified checker was one to two orders of magnitude slower than the `zChaff` built-in checker and took about 50% longer than the `zChaff` solver. Memory problems were described during evaluation as `MiniSAT` failed to produce resolution proofs for all but one of the selected benchmarks. It should be noted that the author's main focus was developing a verified proof checker in order to integrate SAT solving technology into the `Isabelle/HOL` theorem prover, which was successful.

Darbari et al. [12] verified a resolution-based proof checker in `Coq` which is able to execute outside of the theorem-prover environment. The performance was better than that of Weber's, but memory issues were still a concern. The authors found that up to 60% of the total time was spent in garbage collection.

[3] `Minisat` has been improved, but not the proof-logging support.

Armand et al. [21] extended a SAT resolution-based proof checker to include SMT proofs using Coq.

9.3 Verified Solving

One method of assurance for SAT solvers is to develop a verified SAT solver. This approach avoids the need for any post-processing, but this does not provide assurance for state-of-the-art, non-verified solvers. Furthermore, there is a delicate balance between efficiency and verification.

The Davis-Putnam-Logemann-Loveland (DPLL) [25,26] algorithm is one of the most basic SAT solving algorithms where unit propagation is combined with backtracking. Lescuyer and Conchon [27] formalized and verified a DPLL algorithm with Coq [28] using a process called reflection. A certified Ocaml implementation was extracted but the efficiency was not evaluated on larger, industrial-scale problems, and the perofrmance was poor for small examples. Shankar and Vaucher [29] also verified a DPLL solver using PVS; however, the performance was never evaluated.

Modern SAT solvers are built around the CDCL [30] paradigm. In CDCL-based solvers, redundant clauses are added to a formula during solving to reduce the search space. Marić verified pseudocode fragments of a CDCL solver in 2009 [31] and verified a CDCL solver using Isabelle/HOL [32] in 2010 [33]. While the author spent years on the verification process (including verification of a two-watched literal data structure), the performance was never clearly evaluated. Oe et al. [34] developed a verified CDCL solver in Guru called VerSAT. The performance was compared to PicoSAT (which is no longer a state-of-the-art solver), with RUP, and with tracecheck proof generation: VerSAT solved significantly fewer benchmarks within the timeout of 3600 seconds (6 of 16 chosen benchmarks). On the benchmarks VerSAT completed, it was often an order or two of magnitude slower than PicoSAT with tracecheck.

10 Conclusions

A new refutation proof format was presented as well as a new checker that makes it practical to verify unsatisfiability results. Using this approach, proof discovery time is essentially unchanged and the output can be checked in a time similar to the proof discovery time. The new DRUP and IORUP proof formats were introduced. To facilitate the checking of unsatisfiability results, existing SAT solvers and their preprocessors should be modified to emit DRUP proofs. Then, these DRUP proofs can be converted into the IORUP format which can then be efficiently checked. The required modifications are simple, and the conversion and validation are efficient. It appears that all CDCL-based solvers could include a similar capability.

Using the new formats, the computational costs to verify clausal proofs are reduced significantly on medium-to-hard benchmarks. These formats eliminate the most important disadvantage of clausal proofs: their inefficiency. A new

proof checker was implemented to check proofs in these new formats, and the algorithm of the proof checker was analyzed with the aid of a theorem prover. This approach shows that clausal proofs can be compact, easy to obtain, efficient to check, and simple enough to be mechanically verified.

Given these results, one could argue that it is possible to check all refutations produced by SAT solvers and associated preprocessors. This increase in confidence will no doubt further increase their usage of SAT solving in all manner of tools.

References

1. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: Proceedings of the conference on Design, Automation and Test in Europe - Volume 1. DATE '03, IEEE Computer Society (2003) 10880–10885
2. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proceedings of the conference on Design, Automation and Test in Europe - Volume 1. DATE '03, IEEE Computer Society (2003) 10886–10891
3. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: ISAIM. (2008)
4. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Efficient generation of unsatisfiability proofs and cores in SAT. In Cervesato, I., Veith, H., Voronkov, A., eds.: LPAR. Volume 5330 of LNCS., Springer (2008) 16–30
5. de Moura, L.M., Bjørner, N.: Proofs and refutations, and z3. In Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S., eds.: LPAR Workshops. Volume 418 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
6. Van Gelder, A.: Generalized conflict-clause strengthening for satisfiability solvers. In Sakallah, K.A., Simon, L., eds.: SAT. Volume 6695 of LNCS., Springer (2011) 329–342
7. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (February 2009) 131–153
8. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston, MA (June 2000)
9. Järvisalo, M., Biere, A., Heule, M.J.H.: Blocked clause elimination. In Esparza, J., Majumdar, R., eds.: TACAS. Volume 6015 of LNCS., Springer (2010) 129–144
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: SAT. Volume 3569 of LNCS., Springer (2005) 61–75
11. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research **22** (2004) 319–351
12. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Theoretical Aspects of Computing (ICTAC). Volume 6255 of LNCS., Springer (2010) 260–274
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518

14. Biere, A.: PicoSAT essentials. JSAT **4**(2-4) (2008) 75–97
15. Biere, A.: Booleforce and Tracecheck. `http://fmv.jku.at/booleforce/`
16. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design (FMCAD). (2013)
17. Sörensson, N., Biere, A.: Minimizing learned clauses. [35] 237–243
18. Van Gelder, A.: Improved conflict-clause minimization leads to improved propositional proof traces. [35] 141–146
19. Weber, T.: Efficiently checking propositional resolution proofs in Isabelle/HOL. In: International Workshop on the Implementation of Logics (IWIL). Volume 212. (2006) 44–62
20. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic **7**(1) (2009) 26–40
21. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Wener, B.: Verifying SAT and SMT in Coq for a fully automated decision procedure. In: International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT). (2011)
22. Wetzler, N., Heule, M.J.H., Hunt, Jr., W.A.: Mechanical verification of SAT refutations with extended resolution. In: Conference on Interactive Theorem Proving (ITP). Volume 7998 of LNCS., Springer (2013) 229–244
23. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC, ACM (2001) 530–535
24. Van Gelder, A.: Producing and verifying extremely large propositional refutations: Have your cake and eat it too. AMAI **65**(4) (2012) 329–372
25. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7**(3) (1960) 201–215
26. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM **5**(7) (1962) 394–397
27. Lescuyer, S., Conchon, S.: A reflexive formalization of a SAT solver in Coq. In: International Conference on Theorem Proving in Higher Order Logics (TPHOLs). (2008)
28. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
29. Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electronic Notes in Theoretical Computer Science **269** (2011) 3–17
30. Marques-Silva, J.P., Lynce, I., Malik, S.: 4. In: Conflict-Driven Clause Learning SAT Solvers. IOS Press (2009) 131–153
31. Marić, F.: Formalization and implementation of modern SAT solvers. Journal of Automated Reasoning **43**(1) (2009) 81–119
32. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
33. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theoretical Computer Science **411**(50) (2010) 4333–4356
34. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: a verified modern SAT solver. In: Verification, Model Checking, and Abstract Interpretation (VMCAI), Springer (2012) 363–378
35. Kullmann, O., ed.: SAT 2009. In Kullmann, O., ed.: SAT. Volume 5584 of LNCS., Springer (2009)

A Code modifications

This section shows how to modify `Glucose` 2.1, the SAT 2012 challenge winner, and the widely used preprocessor `SatELite`, to emit a proof in the DRUP format. Add the following lines at the end of the procedure `Solver::analyze` in Solver.cc:

```
for (i = 0; i < out_learnt.size(); i++)
    printf("%i ", (var(out_learnt[i]) + 1) *
                  (-2 * sign(out_learnt[i]) + 1));
printf("0\n");
```

Add the following lines at the beginning of `Solver::removeClause` in Solver.cc:

```
printf("d ");
for (int i = 0; i < c.size(); i++)
    printf("%i ", (var(c[i]) + 1) * (-2 * sign(c[i]) + 1));
printf("0\n");
```

The same changes can be used for `Minisat` 2.2.0 on which `Glucose` 2.1 is based.

To emit a proof log by `SatELite`, almost the same lines can be used, but the placement location is less intuitive as compared to the solvers. For the addition of clauses, use the lines shown above, with `out_learn` replaced by `ps`, and add them to Solver_clause.iC in the procedure `Solver::addClause`. The code should be placed directly above the line "`if (ps.size() == 0){`". Placing the lines at this location causes `SatELite` to ignore literals in the input that are falsified by a unit clause in the input. Consequently, the parsed input in the proof is not identical to the raw input. The clause deletion lines are exactly the same and should be placed at the beginning of `Solver::deallocClause` (not in `Solver::removeClause`) in Solver_clause.iC.

B Verification of clausal proof-checking algorithms

The goal of emitting a proof of unsatisfiability is to gain confidence in both the result and the solver. However, it is possible to do more than to trust the proof checker; it is possible to mechanically verify proof-checker implementations. Mechanically-verified proof checkers for the RUP and IORUP formats are described below.

Two unsatisfiability proof checkers were modeled and analyzed with `ACL2` [8]: one checker uses the RUP format, and the other checker uses the IORUP format proposed in this paper. `ACL2` was used to mechanically prove the correctness of an `ACL2`-based implementation of an unsatisfiability proof checker. The soundness theorem states that if the proof checker can verify a proof, then there is no solution for the given formula. Alternatively, an unsatisfiability proof checker is correct if the following theorem is valid (where F is a formula, Q a proof, and τ a truth assignment).

$$\begin{aligned} &\textsc{Checker}\ (F,\ Q) = \texttt{UNSATISFIABLE} \\ \Rightarrow\ &\neg\exists\ \tau \text{ such that } \textsc{Evaluate}\ (F,\ \tau) = \texttt{true} \end{aligned} \tag{2}$$

The proofs of correctness for this RUP proof checker and the IORUP proof checker are quite similar. The IORUP proof checker uses essentially the same algorithm as the RUP proof checker except that it ignores certain clauses during unit propagation. If the checker fails to establish the logical inference of a lemma, then the proof-checking attempt fails and the theorem (2) is trivially valid. In other words, it is the responsibility of the tool that creates the IORUP proof to ensure that the proof is still valid.

The proof of the correctness of the RUP proof checker was done first. With only minor modifications, the IORUP proof checker and its proof of correctness were completed. The definition of the RUP proof checker and its proof of correctness is comprised of 42 `ACL2` events, which include both definitions and theorems. Using the RUP proof checker as a starting point, 10 `ACL2` events were added to create the IORUP proof checker and its proof of correctness. The IORUP proof of correctness takes only four additional seconds to mechanically verify.