Verifying Refutations with Extended Resolution

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

The University of Texas at Austin

Abstract. Modern SAT solvers use preprocessing and inprocessing techniques that are not solely based on resolution; existing unsatisfiability proof formats do not support SAT solvers using such techniques. We present a new proof format for checking unsatisfiability proofs produced by SAT solvers that use techniques such as extended resolution and blocked clause addition. Our new format was designed with three goals: proofs should be easy to generate, proofs should be compact, and validating proofs must be simple. We show how existing preprocessors and solvers can be modified to generate proofs in our new format. Additionally, we implemented a mechanically-verified proof checker in ACL2 and a proof checker in C for the proposed format.

1 Introduction

Satisfiability (SAT) solvers have become the core search engine in many tools used for combinational [1,2] and sequential equivalence checking [3,4], bounded [5] and unbounded model checking [6], and debugging [7]; thus, it is crucial that SAT solvers produce correct results. Presently, some of the best solvers use implementation techniques for which no tools exist to validate the correctness of their results because existing proof formats can only express a subset of the implemented techniques. We introduce a new proof format to express refutation proofs produced by SAT solvers that covers all existing techniques. Additionally, we implemented new tools to verify these refutation proofs.

State-of-the-art SAT solvers are used for a variety of applications. These applications rely on SAT solvers to be efficient enough to solve large problems and provide the correct results. Solvers are often used not only to find a solution for a Boolean formula, but also to make the claim that no solution exists. If a solver claims that a formula is satisfiable, we can check a reported solution linearly in the size of the formula. Yet if a solver claims that a formula has no solutions, we have to trust that the solver fully exhausted the search space for the problem. This is complicated by the fact that state-of-the-art SAT solvers employ a large array of complex techniques to maximize efficiency. Errors can be introduced at a conceptual level and an implementation level [8].

One approach to gain assurance that a SAT solver is correct is to validate the output of the SAT solver. A proof trace is a sequence of clauses that are claimed to be redundant with respect to a given formula. If a SAT solver reports

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

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 345–359, 2013.

that a given formula is unsatisfiable, it can provide a proof trace that can be checked by a smaller, easier-to-trust program called a proof checker. We only need to trust the proof checker; such a checker can validate the results of multiple solvers. Ideally, a proof trace should be compact, easy to obtain, efficient to verify, expressive enough to capture all solving techniques, and it should facilitate a simple checker implementation. We can then either trust that the proof checker is correct or go a step further and formally verify the implementation of the proof checker. By focusing our efforts on a proof checker, we gain assurance while avoiding the need to trust or formally verify a variety of solvers with differing implementations.

SAT solvers have traditionally been checked by emitting a resolution-based proof that is validated by an external checker [9,10,11]. Validating such resolution-based proofs is fast and simple; however, emitting proofs in resolution-based formats is hard and these proofs can be very large. Clausal proofs [10,12] are an alternative approach to resolution-based proofs, and are primarily checked using unit propagation. Clausal proofs are compact and easy to emit, yet verification tools based on clausal approaches are slower and more complex.

Extended Resolution (ER) [13] is the basis for some techniques used during learning [14] and preprocessing [15] in state-of-the-art SAT solvers. Refutations using ER can be exponentially smaller than refutations based solely on resolution. Examples include the pigeon-hole problems where Haken [16] showed that resolution proofs are exponential in size, while Cook [17] demonstrated how to construct polynomial-sized refutations based on ER.

The only (resolution-based) proof-checking tool that can deal with ER is tracecheck, which checks proofs emitted by the EBDDRES [18,19] solver. It simply treats ER clauses as input clauses, and thus does not verify them. Moreover, it is hard to express some techniques, such as bounded variable addition [15], using (multiple applications of) the extension rule. Other techniques, such as blocked clause addition [20], are based on a generalization of ER that cannot be expressed by conventional ER.

To overcome these problems, we propose a new clausal-proof format to compactly express techniques that go beyond resolution. Our proof format is based on a recently-introduced redundancy property of clauses called *Resolution Asymmetric Tautology* (RAT) [21]. All preprocessing and inprocessing techniques used in contemporary state-of-the-art SAT solvers can be simulated by adding and removing RAT clauses [21]. It is easy to emit a refutation in our RAT format for most techniques used in today's solvers and the proofs are compact. We present two tools to check proofs in the proposed format: a mechanically verified checker in the ACL2 theorem prover [22] and a small, fast implementation in C.

Our paper proceeds by presenting some preliminary information in Section 2. Section 3 deals with redundancy properties of clauses. We provide in Section 4 some motivating examples of why a proof format should exist that supports techniques based on ER. In Section 5, we detail resolution proofs and clausal proofs as methods to add clauses that are logically implied by a formula. Our new proof format is presented in Section 6 and two implementations of checkers

for this format are discussed in Section 7. We give an evaluation in Section 8, and we conclude in Section 9.

2 Preliminaries

We briefly review necessary background concepts: conjunctive normal form (CNF), extended resolution, and Boolean constraint propagation.

2.1 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. A clause is a *tautology* if it contains both x and $\bar{x}$ for some variable x. The set of literals occurring in a CNF formula F is denoted by $\mathrm{LIT}(F)$. A truth assignment for a CNF formula F is a partial function τ that maps literals $l \in \mathrm{LIT}(F)$ 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 CNF formula F is *satisfied* by assignment τ if $\tau(C) = \mathbf{t}$ for all $C \in F$.
- A CNF formula F is *falsified* by assignment τ if $\tau(C) = \mathbf{f}$ for some $C \in F$.

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. Two formulas are *logically equivalent* if they have the same set of solutions over the common variables. Two formulas are *satisfiability equivalent* if both have a solution or neither has a solution.

2.2 Resolution and Extended Resolution

The resolution rule states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We say C is the *resolvent* of C_1 and C_2 and write $C = C_1 \bowtie C_2$. C is logically implied by any formula containing C_1 and C_2. Resolution can also be applied to sets of clauses. Let S_x be a set of clauses containing literal x and $S_{\bar{x}}$ a set of clauses containing literal $\bar{x}$. $S_x \bowtie S_{\bar{x}}$ is the set of non-tautological resolvents $R := C_1 \bowtie C_2$ with $C_1 \in S_x$ and $C_2 \in S_{\bar{x}}$.

For a given CNF formula F, the *extension rule* [13] allows one to iteratively add definitions of the form $x := a \wedge b$ by adding clauses $(x \vee \bar{a} \vee \bar{b}) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee b)$ to F, where x is a new variable and a and b are literals in the current formula. *Extended Resolution* [13] is a proof system, whereby the *extension rule* is repeatedly applied to a CNF formula F, followed by applications of the resolution rule. This proof system surpasses what can be done using only resolution; it can even polynomially simulate extended Frege systems [23].

2.3 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 $\mathrm{BCP}(F)$. If $(l) \in \mathrm{BCP}(F)$ for some unit clause $(l) \notin F$, we say that BCP *assigns* the literal l to $\mathbf{t}$ (and the literal $\bar{l}$ to $\mathbf{f}$). If $(l), (\bar{l}) \in \mathrm{BCP}(F)$ for some literal $l \in \mathrm{LIT}(F)$ (or, equivalently, $\emptyset \in \mathrm{BCP}(F)$), we say that BCP *derives a conflict*.

Example 1. Consider the formula $F = (a) \wedge (\bar{a} \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$. We have $(a) \in F$, so $\mathrm{BCP}(F)$ removes literal $\bar{a}$, resulting in the new unit clause (b). After removal of the literals $\bar{b}$, two complementary unit clauses (c) and $(\bar{c})$ are created.

3 Verification Using the RAT Redundancy Property

Clausal proof checking relies on the addition of redundant clauses to a CNF formula. Refutations are a sequence of clauses, terminating with the empty clause, that are redundant w.r.t. a given formula. The most basic redundancy property is T (tautology). RAT is a redundancy property of clauses, computable in polynomial time, that preserves satisfiability; all preprocessing, inprocessing, and solving techniques in state-of-the-art SAT solvers can be expressed in terms of addition and removal of RAT clauses [21]. In this section, we provide an overview of redundancy properties that are covered by RAT.

For a clause C, (*asymmetric literal addition*) $\mathrm{ALA}(F, C)$ computes the *unique* clause resulting from repeating the following until fixpoint: if $l_1, \ldots, l_k \in C$ and there is a clause $(l_1 \vee \cdots \vee l_k \vee l) \in F \setminus \{C\}$ for some literal l, let $C := C \cup \{\bar{l}\}$. A clause C has property AT (asymmetric tautology) with respect to a CNF formula F if and only if $\mathrm{ALA}(F, C)$ has property T. Clauses with the property AT are also known as *reverse unit propagation* (RUP) clauses [10,12]. A clause C is RUP (or has AT) if unit propagation on an assignment that falsifies C will result in a conflict. More formally, let $\overline{C}$ denote the set of unit clauses that falsify all literals in C. Clause C is RUP if and only if $\emptyset \in \mathrm{BCP}(F \cup \overline{C})$.

Given a CNF formula F and a clause $C \in F$, C *has property* R$\mathcal{P}$ (with $\mathcal{P} \in \{\mathrm{T}, \mathrm{AT}\}$) if and only if either (i) C has the property $\mathcal{P}$, or (ii) there is a literal $l \in C$ such that for each clause $C' \in F$ with $\bar{l} \in C'$, each resolvent in $C \bowtie C'$ has $\mathcal{P}$. In the latter case, we say C has R$\mathcal{P}$ on l. Clauses with property RT (resolution tautology) with respect to a CNF formula F are also known as *blocked clauses* [20].

If a clause C has one of the redundancy properties w.r.t. a CNF formula F, one can add C to F and preserve satisfiability, or remove C from F and preserve unsatisfiability. We will focus on adding redundant clauses to a given formula either with the redundancy property AT, which is the strongest redundancy property that preserves logical equivalence, or RAT, which is the strongest redundancy property that preserves satisfiability equivalence. Fig. 1 shows the relationships between clause redundancy properties.

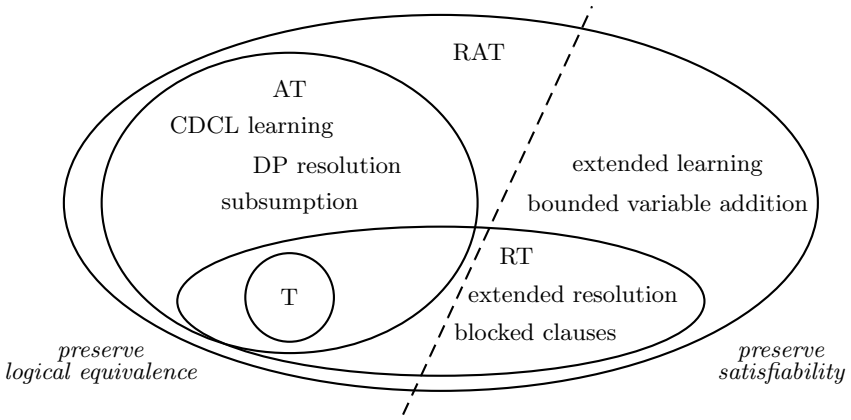

Fig. 1. Relationships between clause redundancy properties that can be computed in polynomial time. Techniques shown in an area denote the cheapest check one can apply to verify a proof trace from a SAT solver that uses that technique. All techniques used in state-of-the-art SAT solvers can expressed as a sequence of RAT clauses [21]. The dashed line separates techniques that preserve logical equivalence and those that preserve satisfiability.

Example 2. Let formula $F = (a \vee b) \wedge (b \vee c) \wedge (\bar{b} \vee \bar{c})$.

- The clause $(a \vee \bar{a})$ is a tautology because it contains a and $\bar{a}$ and therefore has T and thus AT, RT and RAT.
- The clause $(a \vee \bar{c})$ does not have T. However, it has RT (and RAT) with respect to F on literal a, because F contains no clauses with literal $\bar{a}$. Furthermore, it also has AT because unit propagation under the assignment $(\bar{a} \wedge c)$ results in a conflict.
- The clause $(\bar{a} \vee c)$ has RAT, but not T, AT, or RT. It is clear that $(\bar{a} \vee c)$ does not have T. Unit propagation under the assignment $(a \wedge \bar{c})$ does not result in a conflict, so $(\bar{a} \vee c)$ does not have AT. Also, $(\bar{a} \vee c)$ does not have RT, because there is a non-tautological resolvent on $\bar{a}$ with $(a \vee b)$ and on c with $(\bar{b} \vee \bar{c})$. Finally, there is only one resolvent on literal $\bar{a}$. The resolvent of $(\bar{a} \vee c)$ and $(a \vee b)$ is $(b \vee c)$, which is already in F. Therefore, unit propagation on the assignment $(\bar{b} \wedge \bar{c})$ will result in a conflict. Hence, $(\bar{a} \vee c)$ has RAT.

4 Extended Resolution in Practice

This section provides an overview of several techniques that use Extended Resolution or a generalization. We will present these techniques as motivating examples for our proof format (discussed in Section 6) based on RAT clauses.

4.1 Manually-Constructed Proofs

A classic problem known to be hard for resolution provers is the *pigeon-hole* problem. A pigeon-hole problem of size n (denoted by PH_n) describes whether

n pigeons can be placed in $n-1$ holes such that each hole contains at most one pigeon. Although the problem is easy for any n from an abstract level, it is hard to refute the straight-forward translation of pigeon-hole problems into a CNF formula. The number of resolutions to derive the empty clause is exponential in n, and SAT solvers also require exponential runtime on these problems.

Let Boolean variable $x_{i,j}$ denote whether pigeon i is in hole j with $i \in \{1..n\}$ and $j \in \{1..n{-}1\}$. The straight-forward SAT translation of PH_n consists of a set of clauses describing that pigeon i is in at least one hole, and a set of clauses enforcing that if pigeon i is in hole j that pigeon $k > i$ cannot be in hole j.

$$\bigwedge_{i\in\{1..n\}} (x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n-1}) \wedge \bigwedge_{i,j\in\{1..n-1\}} \bigwedge_{k\in\{i+1..n\}} (\bar{x}_{i,j} \vee \bar{x}_{k,j}) \quad (1)$$

Although the problem is hard for resolution [16], polynomial-size refutations do exist for the Extended Resolution [13] technique. ER can reduce PH_n into PH_{n-1}. Applying the reduction $n{-}1$ times, results in the trivial PH_1. The first step of the reduction is introducing auxiliary Boolean variables $y_{i,j}$ with $i \in \{1..n{-}1\}$ and $j \in \{1..n{-}2\}$. When all reduction steps are applied, these variables $y_{i,j}$ encode that pigeon i is in hole j in the PH_{n-1} problem. Let

$$y_{i,j} := x_{i,j} \vee (x_{n,j} \wedge x_{i,n-1}) \quad (2)$$

The definition can be translated into clauses that have all RT on $y_{i,j}$.

$$(y_{i,j} \vee \bar{x}_{i,j}) \wedge (y_{i,j} \vee \bar{x}_{n,j} \vee \bar{x}_{i,n-1}) \wedge (\bar{y}_{i,j} \vee x_{i,j} \vee x_{n,j}) \wedge (\bar{y}_{i,j} \vee x_{i,j} \vee x_{i,n-1}) \quad (3)$$

By adding the clauses (3) with $i \in \{1..n-1\}, j \in \{1..n-2\}$ to the formula, the clauses encoding the left set of clauses of (1) with $y_{i,j}$ variables are:

$$(y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,n-2}) \quad (4)$$

Notice that the clauses (4) have AT after the presence of (3): First assign $y_{i,1}, y_{i,2}, \ldots, y_{i,n-2}$ to false, then BCP assigns $x_{i,1}, x_{i,2}, \ldots, x_{i,n-2}$ to false using $(y_{i,j} \vee \bar{x}_{i,j})$. This in turn makes $x_{i,n-1}$ true by $(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n-1})$. After these assignments, all the clauses $(y_{i,j} \vee \bar{x}_{n,j} \vee \bar{x}_{i,n-1})$ become unit assigning all $x_{n,1}, x_{n,2}, \ldots, x_{n,n-2}$ to false. Now, $(x_{n,1} \vee x_{n,2} \vee \cdots \vee x_{n,n-1})$ assigns $x_{n,n-1}$ to true, and a conflict arises because the clause $(\bar{x}_{i,n-1} \vee \bar{x}_{n,n-1})$ is falsified.

The right set of clauses of (1) using $y_{i,j}$ variables is required to finish the reduction. These $(\bar{y}_{i,j} \vee \bar{y}_{k,j})$ don't have RAT. Yet the clauses $(\bar{y}_{i,j} \vee \bar{y}_{k,j} \vee x_{i,n-1})$ have AT and in the presence of $(\bar{y}_{i,j} \vee \bar{y}_{k,j} \vee x_{i,n-1})$, $(\bar{y}_{i,j} \vee \bar{y}_{k,j})$ has AT as well.

$$(\bar{y}_{i,j} \vee \bar{y}_{k,j} \vee x_{i,n-1}); \quad (\bar{y}_{i,j} \vee \bar{y}_{k,j}) \quad (5)$$

So by adding the clauses (3), (4), and (5) —all having RT or AT, and thus RAT— we can reduce the PH_n problem into a PH_{n-1} problem. Repeating this $n-1$ times results in a refutation. A similar, but much larger proof of unsatisfiability can be obtained by combining resolution and ER, as in [17].

4.2 Extended Learning

Our manually-constructed extended resolution proof above illustrates the potential of introducing new variables; however, it is hard to capitalize on this potential in practice. Most applications of the extension rule will result in useless variables which can slow down the search. The most serious study of practical ER during search looks for a pattern between consecutive conflict clauses [14]. When such a pattern is found, a new variable is introduced.

The pattern consists of conflict clauses that differ in exactly one literal. Given two successive conflict clauses $C = (l_1 \vee \alpha)$ and $D = (l_2 \vee \alpha)$ with α being a disjunction of literals, the extension rule is applied using $z := (\bar{l}_1 \vee \bar{l}_2)$. Newly introduced variables are used to shorten future conflict clauses. Let γ be a disjunction of literals. If a conflict clause $(l_1 \vee l_2 \vee \gamma)$ is found and the variable $z := (l_1 \vee l_2)$ was created in the past, $(z \vee \gamma)$ is added to the learned clause database.

4.3 Bounded Variable Addition

One of the most effective preprocessing/inprocessing techniques is *Bounded Variable Elimination* (BVE) [24]. This technique tries to reduce the sum of the number of variables and the number of clauses by eliminating variables. Given a CNF formula F, let F_l denote the subset of clauses of F that contains literal l. BVE searches for a variable x, such that it can replace F_x and $F_{\bar{x}}$ by the set of non-tautological resolvents of $F_x \bowtie F_{\bar{x}}$ if and only if $|F_x \bowtie F_{\bar{x}}| \leq |F_x| + |F_{\bar{x}}|$.

Recently a complementary technique was proposed, called *Bounded Variable Addition* (BVA) [15], that introduces new variables. BVA uses the same metric for substitution: new variables are added while the sum of the number of variables and clauses strictly decreases. BVE is based on resolution and therefore one can use existing resolution and clausal proof formats to verify an implementation. However, BVA cannot be simulated by resolution and simulation by ER is non-trivial. Yet a proof trace of BVA can elegantly be expressed using RAT clauses.

BVA works as follows. Given a CNF formula F and a new Boolean variable x, BVA searches for sets of clauses G_x (clauses containing literal x) and $G_{\bar{x}}$ (clauses containing literal $\bar{x}$), such that all non-tautological resolvents of $G_x \bowtie G_{\bar{x}}$ are in F and $|G_x \bowtie G_{\bar{x}}| > |G_x| + |G_{\bar{x}}|$. Whenever BVA finds such a G_x and $G_{\bar{x}}$, it replaces the resolvents by: $F := (F \cup G_x \cup G_{\bar{x}}) \setminus G_x \bowtie G_{\bar{x}}$.

Example 3. Consider the formula $F = (a \vee c) \wedge (a \vee d) \wedge (a \vee e) \wedge (b \vee c) \wedge (b \vee d) \wedge (b \vee e)$. For F there exists $G_x = (a \vee x) \wedge (b \vee x)$ and $G_{\bar{x}} = (\bar{x} \vee c) \wedge (\bar{x} \vee d) \wedge (\bar{x} \vee e)$ such that all non-tautological resolvents of $G_x \bowtie G_{\bar{x}}$ are in F and $|G_x \bowtie G_{\bar{x}}| > |G_x| + |G_{\bar{x}}|$.

All clauses added by the extension rule are blocked (have RT) on the new variable. However, this is not the case with BVA. In Example 3, none of the clauses in G_x and $G_{\bar{x}}$ are blocked; all these clauses have RAT on the new variable. Without loss of generality consider a clause $C \in G_x$. All resolvents $R := C \bowtie C'$ with $C' \in G_{\bar{x}}$ are either tautologies or R is subsumed by F (namely, $R \in F$). Both tautologies and subsumptions are asymmetric tautologies, and therefore, C has RT.

5 Existing Proof Formats

Conflict-driven clause learning (CDCL) [25] is the leading paradigm of modern SAT solvers. A core aspect of CDCL solvers is the addition and removal of clauses. The main form of reasoning is known as conflict analysis, which adds clauses. Additionally, state-of-the-art CDCL solvers use preprocessing and inprocessing techniques that both add and remove clauses. Proof formats for CDCL solvers express *how to check* that a clause addition step preserves satisfiability. This section provides an overview of existing formats. In next section, we present our new format.

We appeal to the notion that *lemmas* are used to construct a proof of a theorem. Here, lemmas represent the learned clauses and the theorem is the statement that the formula is unsatisfiable. From now on, we will use the term clauses to refer to input clauses, while lemmas will refer to added clauses.

5.1 Resolution Proofs

The early approaches to prove refutations produced by SAT solvers were based on resolution [9]. The lemmas computed by CDCL solvers can be simulated by a sequence of resolutions [26]. Let L be a lemma and $\{C_1, \ldots, C_n\}$ be the input clauses. For each L, one must specify a sequence such that $L = (((C_i \bowtie C_j) \ldots) \bowtie C_k)$. This sequence may use added lemmas to construct new lemmas.

As resolution is an elementary operation, simple and fast checking algorithms exist [9,10,11]. However, resolution proofs can be huge (dozens of gigabytes). It may also be hard to modify a SAT solver to emit a resolution refutation; for instance, one must determine the clauses on which to apply resolution, and specifying the order may be difficult. Since only a handful of (mostly outdated) solvers support resolution-based proofs, a user would need to modify a solver to emit resolution proofs. Even for the author(s) of a SAT solver this is not an easy task. In the case one wishes to integrate a portfolio of SAT solvers into a SMT solver or theorem prover, it would be a daunting task to enhance them all to emit resolution proofs.

5.2 Clausal Proofs

An alternative approach using *clausal proofs* was proposed by Goldberg and Novikov [12]. They observed that each lemma L learned by CDCL conflict analysis can be checked using BCP. Lemmas, like clauses, are disjunctions of literals. If $\text{BCP}(F \cup \overline{L})$ results in a conflict, i.e., produces the empty clause $\emptyset$, then L is implied by F. Notice that this corresponds to the redundancy property AT. Lemmas with AT are also known as *reverse unit propagation* (RUP) lemmas [10].

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 proof of F consists of lemmas L_i that are redundant w.r.t. F. Let $F_0 = F$ and $F_i := F_{i-1} \cup \{L_i\}$. Existing clausal proof formats expect that lemma L_i has AT w.r.t. F_{i-1}. In our proposed format (see Section 6) lemmas should have the more general RAT redundancy property.

The elegance of clausal proofs is that they can be expressed in conjunctive normal form; however, the order of the lemmas is important. Clausal proofs are significantly smaller when compared to resolution proofs, and only minor modifications of a SAT solver are required to output such proof records. However, checking of clausal proofs can be quite expensive. Checking algorithms for clausal proofs are also typically more complex than those for resolution proofs, making it harder to trust or prove correctness of the algorithm.

```
RUPchecker (CNF formula F, queue Q of lemmas)
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 ∃ (x) ∈ F do
12    for C ∈ F with x̄ ∈ C do
13      C := C \ {x̄}
14    for C ∈ F with x ∈ C do
15      F := F \ {C}
16  return F
```

Fig. 2. Pseudo-code to check clausal proofs for lemmas with AT (or RUP lemmas)

Fig. 2 shows the pseudo-code of a clausal, proof-checking algorithm for lemmas with AT. The input is a CNF formula F and a queue Q of lemmas representing a refutation of F. Lemmas are sorted in chronological order as learned by the 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 we fail to check that L is logically implied by F and terminate (line 3 and 4). Otherwise, L is added to F. In case L was unit, the new F is simplified using BCP (line 5). If unit propagation 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 were validated but no top-level conflict was encountered.

6 The RAT Proof Format

In this section, we propose the new RAT proof format. This is an alternative clausal-proof format that supports both AT (or RUP) and RAT lemmas.

The main decision regarding a proof format for ER and its generalizations was whether to use a resolution-style or clausal-style proof format. Apart from the known disadvantages of resolution-style proofs (recall Section 5.1), there is another drawback of ER proofs: techniques like blocked clause addition [20], cannot be expressed using the extension rule. Consequently, if one wants to verify all known techniques, a clausal-style proof format seems the most viable option.

We considered whether to specify the simplest redundancy property for each lemma. All redundancy properties are covered by RAT, so it is not necessary to distinguish between them; however, efficiency may be gained by distinguishing

them. In practice, the majority of lemmas has the AT property; therefore, by first checking for AT, which is part of the RAT check, we reduce overhead.

In case a lemma does not have AT, our proof format expects the lemma to have RAT on its first literal. Fig. 3 shows three refutations in the RUP (mid left) and RAT formats (both on the right). The last RAT refutation shows that one can introduce new variables in a RAT proof — which is not allowed in RUP proofs.

CNF formula

```
p cnf 4 16
 1  2  3  4 0
 1  2  3 -4 0
 1  2 -3  4 0
 1  2 -3 -4 0
 1 -2  3  4 0
 1 -2  3 -4 0
 1 -2 -3  4 0
 1 -2 -3 -4 0
-1  2  3  4 0
-1  2  3 -4 0
-1  2 -3  4 0
-1  2 -3 -4 0
-1 -2  3  4 0
-1 -2  3 -4 0
-1 -2 -3  4 0
-1 -2 -3 -4 0
```

smallest RUP proof

```
1 2 3 0
1 2 0
1 3 0
1 0
2 3 0
2 0
3 0
0
```

smallest RAT proof

```
1 0
2 0
3 0
0
```

RAT proof with ER

```
 5  1  2  0
 5  1 -2  0
 5 -1  2  0
 5 -1 -2  0
-5  3  4  0
-5  3 -4  0
-5 -3  4  0
-5 -3 -4  0
 5  1  0
 5  0
 3  0
 0
```

Fig. 3. An example of a CNF problem in the typical DIMACS format (left) as well as three refutations; one in the RUP format (mid left) and two in RAT format (right). The proofs in the middle show a smallest proof (in the number of lemmas) for RUP and RAT. Whitespaces can be of any length; the spacing is to improve readability. A `0` marks the end of clauses and lemmas. The RUP and RAT formats have the same syntax. Only the RAT format allows lemmas to have the RAT redundancy property.

7 Implementation

We have implemented[1] a RAT checker in ACL2 that is concise in its expression, and, more importantly, mechanically verified. Our proof of correctness for our RAT checker hinges on the mechanical proof of the redundancy of the RAT property presented in Section 3. We did this by modeling the RAT proof-checking algorithm as an ACL2 function, and then we used the ACL2 mechanical proof-checking system to assure that our RAT proof-checking algorithm is valid.

[1] The material presented in the paper, such as the formal proof, our tools, and the used benchmarks are available on `www.cs.utexas.edu/~marijn/rat/`. Our proof contains roughly 150 ACL2 (definitions and proof request) events.

Apart from our mechanically verified checker, we implemented a concise RAT checker in C (about 200 lines of code). Fig. 4 shows its pseudo-code. Our RAT checker extends the RUP checker pseudo-code (recall Fig. 2) and uses the same input parameters, initialization (lines 1 and 2), and termination (lines 8 to 10). Before validating a lemma L, it first checks whether L has AT (line 3). Otherwise, the expensive RAT check is applied. According to the RAT proof format, lemma L should have RAT on its first literal l (line 4). We compute for all clauses $C' \in F_{\bar{l}}$ (line 5), the resolvent $R := C' \bowtie L$ (line 6), and check whether R has AT (line 7). If all these R have AT, L is added to F (line 8); otherwise, we return "checking failed".

```
   RATchecker (CNF formula F, queue Q of lemmas)
1    while Q is not empty
2      L := Q.pop()
3      if ∅ ∉ BCP(F ∪ L̄) then          // check if L has AT, otherwise
4        let l be the first literal in L.      // assume L has RAT on l
5        forall C' ∈ F_l̄ do
6          R := C' ⋈ L
7          if ∅ ∉ BCP(F ∪ R̄) then return "checking failed"
8      F := BCP(F ∪ L)
9      if ∅ ∈ F then return "unsatisfiable"
10   return "all lemmas validated"
```

Fig. 4. Pseudo-code to check Resolution Asymmetric Tautology (RAT) proofs

In general, the RAT check (lines 4 to 7) is more expensive than the AT check (line 3), because one has to do the AT check for each R. However, in practice, for half the RAT lemmas, $F_{\bar{l}}$ is empty and we skip lines 4 to 7.

The main reason why a RAT checker is more complex and less efficient, as compared to a RUP checker, is the requirement in line 5 to compute $F_{\bar{l}}$, the set of clauses containing literal $\bar{l}$. In order to do this computation efficiently, the checker needs to maintain a full occurrence list of all clauses. Alternatively, a RUP checker could use a watch-pointer data structure.

Notice that for all checks (lines 3 and 7), all literals $l' \in L \setminus l$ will be assigned to false. One can optimize a RAT checker by first assigning all the literals $l' \in L \setminus l$ to false followed by unit propagation and perform the checks on this assignment. This optimization is implemented in our C checker.

8 Evaluation

To demonstrate the usefulness of the RAT proof format, we experimented with our tools on the problems discussed in Section 4. We ran our tests on a 4-core Intel Xeon CPU E31280 3.50GHz, 32 Gb RAM machine running Ubuntu 10.04.

8.1 Manually-Constructed Proofs

The first experiment evaluates the performance of our RAT checking tools on the manually-contructed proofs of PH_n problems presented in Section 4.1. Although these problems are notoriously hard for SAT solvers, the manually-contructed proofs are small and can be checked in a fraction of a second using our C implementation, see Table 1. The ACL2 checker is significantly slower, but was not written with speed in mind.

Table 1. Evaluation of manually-constructed proofs of PH_n problems. The first column shows the benchmark name. The next two columns show the number of variables in the input formula and in the proof. The number of original, AT, and RAT clauses, as well as their sum (total) is shown in the next four columns. The last two columns show the time (in seconds) to check the proofs using our C and ACL2 implementations.

	#variables		*#clauses*				*time*	
benchmark	input	proof	input	AT	RT	total	C	ACL2
PH_6	30	70	81	160	145	386	0.003	0.26
PH_7	42	112	133	280	301	714	0.005	1.55
PH_8	56	168	204	448	560	1,212	0.007	7.91
PH_9	72	240	297	672	960	1,929	0.010	34.26
PH_{10}	90	330	415	960	1,545	2,920	0.014	129.78
PH_{11}	110	440	561	1,320	2,365	4,246	0.016	440.69
PH_{12}	132	572	738	1,760	3,476	5,974	0.020	1358.65

8.2 Extended Learning

We modified the solver `GlucosER 1.0` [14], which combines CDCL learning and ER, such that it emits a proof in the proposed RAT format[2]. We evaluated the *C* checking tool on benchmarks where `GlucosER` has an edge over SAT solvers without ER learning, such as the PH_n instances and the Urquhart benchmarks [27].

Table 2 shows the results of the second experiment with our C checking tool. Compared to the prior results, the C tool requires much more time to verify the output of `GlucosER`. Notice that although `GlucosER` uses ER, it cannot compete with the manually-constructed proofs on the same problems.

8.3 Bounded Variable Addition

The technique BVA, discussed in Section 4.3, is a helpful preprocessing technique for several families of benchmarks, including the PH_n problems and some hard bioinformatics [28] benchmarks. We modified a preprocessing tool which includes a BVA implementation, `coprocessor` [29], and the `Glucose 2.1` solver [30] (the winner of the SAT 2012 Challenge) to output lemmas in the RAT format. We verified the merged file consisting of the original problem and the lemmas produced by the preprocessor and solver.

[2] Additionally, we removed the code that allows reuse of variables that have been eliminated. The removal of this part of the code made it easier to verify and has no noticeable effect on the performance.

Table 2. Evaluation of Extended Learning on PH_n and Urquhart benchmarks. The first column shows the benchmark name. The next two columns show the number of variables in the input formula and in the proof. The number of original, AT, and RAT clauses, as well their sum (total) is shown in the next four columns. The last two columns show the time (in seconds) to solve the benchmarks and to check the emitted proofs using our C checker.

	#variables		*#clauses*				*time*	
benchmark	input	proof	input	AT	RT	total	solving	checking
PH_{10}	90	379	415	99,682	867	100,973	5.28	24.72
PH_{11}	110	814	561	260,677	2,112	263,350	13.51	72.08
PH_{12}	132	1,450	738	1,512,453	3,954	1,517,145	145.29	3,521.23
Urq_3_5	45	2,126	446	281,761	6,243	288,450	8.33	17.38
Urq_3_6	54	3,842	688	1,156,477	11,364	1,168,529	52.69	152.36
Urq_3_7	42	1,147	342	102,950	3,315	106,607	2.20	3.95
Urq_3_8	44	1,518	416	149,286	4,422	154,124	3.70	5.86

Table 3 shows the results regarding the performance improvements due to BVA and the RAT proof checking costs. The performance difference when using `Glucose 2.1` on the original and BVA-preprocessed instances is huge: the largest instance cannot be solved in 12 hours, while the preprocessed formula is solved in two minutes. It is important to check that these gains are not caused by a bug. Our proof checker confirms that the refutation is correct.

Table 3. Evaluation of checking RAT produced by BVA preprocessing on PH_n and bioinformatics (rbclY) benchmarks. The timeout (denoted by —) is 12 hours. The first column shows the benchmark name. The next three columns show the number of variables, the number of clauses, and the solving time (in seconds) of the original formula. The next three columns show the same information for the BVA preprocessed formula. The last three columns show the number of AT and RAT clauses in the proofs, as well as the time (in seconds) to check the proofs using our C checker.

	original			BVA preprocessed			RAT proof checking		
benchmark	*#vars*	*#cls*	*time*	*#vars*	*#cls*	*time*	#AT	#RAT	*time*
PH_{10}	90	330	7.71	117	226	1.25	42,853	198	4.19
PH_{11}	110	440	84.42	151	281	12.34	225,959	295	152.82
PH_{12}	132	572	494.29	187	342	8.45	181,603	402	69.01
rbcl_07	1,128	57,446	52.92	1,784	7,598	2.88	72,073	19,681	6.76
rbcl_08	1,278	67,720	1,763.36	1,980	9,004	10.72	151,894	22,830	37.58
rbcl_09	1,430	79,118	—	2,190	10,492	129.20	882,213	26,639	2,631.28

We are working on techniques to decrease the time to check RAT proofs. Initial results indicate that RAT verification can be realized in a time similar to the solving time. Improvement to the speed will likely increase the complexity of the checker implementation.

9 Conclusions

We presented a new clausal proof format for SAT solvers. The crucial difference is that we allow lemmas to have the redundancy property RAT. Since all techniques used in state-of-the-art SAT solvers can be simulated by the addition and removal of RAT lemmas [21], our new format facilitates the verification of results produced by SAT solvers. For most techniques, it is easy to modify a solver to emit a proof in our format, which includes CDCL and ER learning, and bounded variable addition.

Two major challenges remain to conveniently verify the results of SAT solvers. Our C implementation may be slow when a solver emits a huge proof. It is still an open question whether only minor modifications to SAT solvers are needed for all techniques. For example, Gaussian elimination of XOR constraints can be simulated using ER [18,19] techniques, but these methods require several modifications for SAT solvers that use Gaussian elimination.

Our new format and our tools are the first, complete approach toward SAT solver verification. We expect them to be used to check implementations of the more complex techniques, in particular those based on ER. Our new format and tools support the development of new techniques that may further capitalize on the strength of ER.

References

1. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: DATE, pp. 114–121 (2001)
2. Mishchenko, A., Chatterjee, S., Brayton, R.K., Eén, N.: Improvements to combinational equivalence checking. In: Hassoun, S. (ed.) ICCAD, pp. 836–843. ACM (2006)
3. Baumgartner, J., Mony, H., Paruthi, V., Kanzelman, R., Janssen, G.: Scalable sequential equivalence checking across arbitrary design transformations. In: ICCD. IEEE (2006)
4. Kaiss, D., Skaba, M., Hanna, Z., Khasidashvili, Z.: Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In: FMCAD, pp. 20–26. IEEE Computer Society (2007)
5. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of bdds. In: DAC, pp. 317–320 (1999)
6. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
7. Chen, Y., Safarpour, S., Marques-Silva, J.P., Veneris, A.G.: Automated design debugging with maximum satisfiability. IEEE Trans. on CAD of Integrated Circuits and Systems 29(11), 1804–1817 (2010)
8. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 44–57. Springer, Heidelberg (2010)
9. Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE, pp. 10880–10885 (2003)

10. Van Gelder, A.: Verifying rup proofs of propositional unsatisfiability. In: ISAIM (2008)
11. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 260–274. Springer, Heidelberg (2010)
12. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for cnf formulas. In: DATE, pp. 10886–10891 (2003)
13. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2, pp. 466–483. Springer (1983)
14. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press (2010)
15. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Proceedings of Haifa Verification Conference 2012 (2012)
16. Haken, A.: The intractability of resolution. Theor. Comput. Sci. 39, 297–308 (1985)
17. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News 8(4), 28–32 (1976)
18. Sinz, C., Biere, A.: Extended resolution proofs for conjoining bdds. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 600–611. Springer, Heidelberg (2006)
19. Jussila, T., Sinz, C., Biere, A.: Extended resolution proofs for symbolic SAT solving with quantification. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 54–60. Springer, Heidelberg (2006)
20. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics 96-97, 149–176 (1999)
21. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
22. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston (2000)
23. Cook, S.A., Reckhow, R.A.: The relative efficiency of propositional proof systems. The Journal of Symbolic Logic 44(1), 36–50 (1979)
24. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
25. Marques-Silva, J.P., Lynce, I., Malik, S.: 4. In: Conflict-Driven Clause Learning SAT Solvers. Handbook of Satisfiability, pp. 131–153. IOS Press (February 2009)
26. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. JAIR 22, 319–351 (2004)
27. Urquhart, A.: Hard examples for resolution. J. ACM 34(1), 209–219 (1987)
28. Bonet, M.L., John, K.S.: Efficiently calculating evolutionary tree measures using SAT. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 4–17. Springer, Heidelberg (2009)
29. Manthey, N.: Coprocessor 2.0 – A flexible CNF simplifier. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 436–441. Springer, Heidelberg (2012)
30. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Boutilier, C. (ed.) IJCAI, pp. 399–404 (2009)