Satisfiability Solvers

Marijn J.H. Heule[1] and Armin Biere[2]

[1] The University of Texas at Austin, United States
[2] Johannes Kepler University, Linz, Austria

1 Introduction

Satisfiability (SAT) solvers have become powerful tools to solve a wide range of applications. In case SAT problems are satisfiable, it is easy to validate a witness. However, if SAT problems have no solutions, a proof of unsatisfiability is required to validate that result. Apart from validation, proofs of unsatisfiability are useful in several applications, such as interpolation [50] and extracting a minimal unsatisfiable set (MUS) [38] and in tools that use SAT solvers such as theorem provers [3, 51–53].

Since the beginning, proof logging of unsatisfiability results was based on two approaches: *resolution proofs* and *clausal proofs*. Resolution proofs, introduced in `zChaff` in 2003 [55], require for learned clauses (lemmas) a list of antecedents. On the other hand, for clausal proofs, introduced in `Berkmin` in 2003 [22], the proof checker needs to find the antecedents for lemmas. Consequently, resolution proofs are much larger than clausal proofs, while resolution proofs are easier and faster to validate than clausal proofs.

Both proof approaches are used in different settings. Resolution proofs are often required in applications like interpolation [36] or in advanced techniques for MUS extraction [39]. Clausal proofs are more popular in the context of validating results of SAT solvers, for example during the SAT Competitions or recently for the proof of Erdős Discrepancy Theorem [30].

Proof logging support became widespread in state-or-the-art solvers, such as `Lingeling` [10], `Glucose` [6], and `CryptoMiniSAT` [43], since SAT Competition 2013 made unsatisfiability proofs mandatory for solvers participating in the unsatisfiability tracks. About half the solvers that participated in recent SAT Competitions can emit clausal proofs, including the strongest solvers around. However, very few solvers support emitting resolution proofs.

The lack of support for resolution proofs is due to the difficulty to represent some techniques used in contemporary SAT solvers in terms of resolution. One such technique is conflict clause minimization [44], which requires several modifications of the solver in order to express it using resolution steps [48]. In contrast, emitting a clausal proof from SAT solvers such as `MiniSAT` [20] and `Glucose` requires only small changes to the code[3].

[3] A patch that adds clausal proof logging support to `MiniSAT` and `Glucose` is available on `http://www.cs.utexas.edu/~marijn/drup/`.

2 Proof Systems

2.1 Preliminaries and Notation

We briefly review necessary background concepts regarding the Boolean satisfiability (SAT) problem. 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 variable and literals occurring in a CNF formula F is denoted by $\mathsf{vars}(F)$ and $\mathsf{lits}(F)$, respectively. A truth assignment for a CNF formula F is a partial function τ that maps literals $l \in \mathsf{lits}(F)$ to $\{\mathbf{t}, \mathbf{f}\}$. If $\tau(l) = v$, then $\tau(\bar{l}) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. An assignment can also be thought of as a conjunction of literals. Furthermore, given an assignment τ:

- A clause C is *satisfied* by τ if $\tau(l) = \mathbf{t}$ for some $l \in C$.
- A clause C is *falsified* by τ if $\tau(l) = \mathbf{f}$ for all $l \in C$.
- A formula F is *satisfied* by τ if $\tau(C) = \mathbf{t}$ for all $C \in F$.
- A formula F is *falsified* by 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. The symbol ϵ refers to the unsatisfiable empty clause. Any formula that contains ϵ is unsatisfiable. A proof of unsatisfiability shows why ϵ is redundant with respect to a given formula.

2.2 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 $C = C_1 \diamond C_2$. C_1 and C_2 are called the *antecedents* of C. C is logically implied by any formula containing C_1 and C_2. A *resolution chain* is a sequence of resolution operations such that result of the last operation is an antecedent of the next operation. Resolution chains are computed from left to right. Notice that the resolution operation is not associative. For example, $\big((a \vee c) \diamond (\bar{a} \vee b)\big) \diamond (\bar{a} \vee \bar{b}) = (\bar{a} \vee c)$, while $(a \vee c) \diamond \big((\bar{a} \vee b) \diamond (\bar{a} \vee \bar{b})\big) = (c)$.

Throughout this chapter we will use the example formula E to the explain the different concepts:

$$E := (\bar{b} \vee c) \wedge (a \vee c) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a \vee \bar{b}) \wedge (b \vee \bar{c})$$

A *unit clause* is a clause of length one. A unit clause forces its only literal to be true. Unit propagation is an important technique used in SAT solvers and works as follows: Given a formula F, repeat the following until fixpoint: If F contains a unit clause (l), remove all clauses with containing l and remove all literal occurrences of $\bar{l}$. If unit propagation on a formula F produces ϵ, denoted by $F \vdash_1 \epsilon$, F is unsatisfiable.

Let $C := (l_1 \lor l_2 \lor \cdots \lor l_k)$ be a clause. We denote with $\bar{C}$ the set of clauses $(\bar{l}_1) \land (\bar{l}_2) \land \cdots \land (\bar{l}_k)$. C is called a *reverse unit propagation* (RUP) clause with respect to F, if $F \land \bar{C} \vdash_1 \epsilon$ [47]. A RUP clause C with respect to F is logically implied by F and one can construct a resolution chain for C using the clauses in F of size at most $|\mathsf{vars}(F)|$. For example, $E \land (\bar{c}) \vdash_1 (\bar{b}) \vdash_1 (a) \vdash_1 \epsilon$ using the clauses $(\bar{b} \lor c)$, $(a \lor c)$, and $(\bar{a} \lor b)$. We can convert this in a resolution chain $(c) := (\bar{a} \lor b) \diamond (a \lor c) \diamond (\bar{b} \lor c)$.

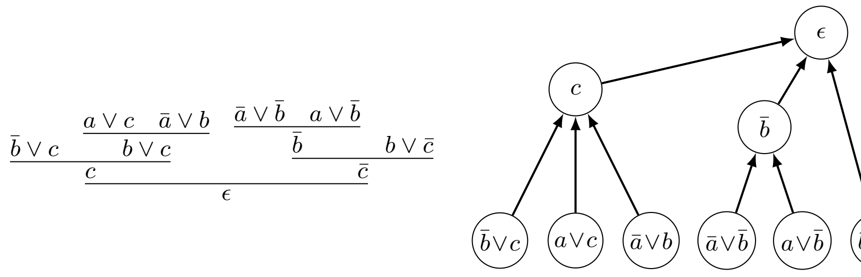

Fig. 1. A resolution derivation (left) and a resolution graph (right) for an example formula.

2.3 Extended Resolution and Its Generalizations

For a given CNF formula F, the *extension rule* [45] allows one to iteratively add definitions of the form $x := a \land b$ by adding clauses $(x \lor \bar{a} \lor \bar{b}) \land (\bar{x} \lor a) \land (\bar{x} \lor b)$ to F, where x is a new variable and a and b are literals in the current formula. *Extended Resolution* [45] 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 [15]. Several generalizations of extended resolution have been proposed. Two important generalizations regarding proof systems are blocked clause addition [31] and resolution asymmetric clause addition [29].

Blocked Clauses Given a CNF formula F, a clause C, and a literal $l \in C$, the literal l *blocks* C with respect to F if (i) for each clause $C' \in F$ with $\bar{l} \in C'$, $C \diamond_l C'$ is a tautology, or (ii) $\bar{l} \in C$, i.e., C is itself a tautology. Given a CNF formula F, a clause C is *blocked* with respect to F if there is a literal that blocks C with respect to F. Addition and removal of blocked clauses results in a satisfiability-equivalent formula [31].

Example 1. Recall example formula E. Clause $(\bar{b} \lor c)$ is blocked on c with respect to E, because resolution on the only clause containing $\bar{c}$, results in a tautology, i.e., $(\bar{b} \lor c) \diamond (b \lor \bar{c}) = (\bar{b} \lor b)$. Since we know that E is unsatisfiable, $E \setminus \{(\bar{b} \lor c)\}$ must be unsatisfiable.

To see that blocked clause addition is a generation of extended resolution, consider a formula containing variables a and b, but without variable x. The three clauses from the extension rule, i.e, $(x \vee \bar{a} \vee \bar{b})$, $(\bar{x} \vee a)$, and $(\bar{x} \vee b)$, are all blocked on x / $\bar{x}$ regardless of the order in which they are added. Hence blocked clause addition can add these three clauses, while preserving satisfiability.

In contrast to extended resolution, blocked clause addition can extend the formula with clauses that are not logically implied by the formula and do not contain a fresh variable. For example, consider the formula $F := (a \vee b)$. The clause $(\bar{a} \vee \bar{b})$ is blocked on $\bar{a}$ (and $\bar{b}$) with respect to F and can thus be added using blocked clause addition.

Resolution Asymmetric Tautologies Resolution asymmetric tautologies (or RAT clauses) [29] are a generation of both RUP clauses and blocked clauses (and hence extended resolution). A clause C has RAT on l (referred to as the pivot literal) with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$ holds that

$$F \wedge \bar{C} \wedge (\bar{D} \setminus \{(l)\}) \vdash_1 \epsilon.$$

Adding and removing RAT clauses results in a satisfiability-equivalent formula [29]. Given a formula F and a clause C that has RAT on $l \in C$ with respect to F. Let τ be an assignment that satisfies F and falsifies C. The assignment τ', which is a copy of τ with the exception that $\tau'(l) = \mathbf{t}$, satisfies $F \wedge C$. This observation can be used to reconstruct a satisfying assignment for the original formula in case it is satisfiable.

To see that RAT clauses are a generalization of blocked clauses and RUP clauses, observe the following. If a clause has RAT on some $l \in C$ with respect to a formula F, it also has RUP with respect to F because

$$F \wedge \bar{C} \vdash_1 \epsilon \implies F \wedge \bar{C} \wedge (\bar{D} \setminus \{(l)\}) \vdash_1 \epsilon.$$

Furthermore, if a clause C is blocked on l with respect to F, then for all $D \in F$ with $\bar{l} \in D$ holds that C contains a literal $k \neq l$ such that $\bar{k} \in D$. Now we have

$$F \wedge (k) \wedge (\bar{k}) \vdash_1 \epsilon \implies F \wedge \bar{C} \wedge (\bar{D} \setminus \{(l)\}) \vdash_1 \epsilon.$$

3 Proof Search

The leading paradigm to solve satisfiability problems is the conflict-driven clause learning (CDCL) approach [35]. In short, CDCL adds lemmas, typically referred to as conflict clauses, to a given input formula until either it finds a satisfying assignment or is able to learn the empty clause (prove unsatisfiability). We refer to a survey on the CDCL paradigm for details [35].

An alternative approach to solve satisfiability problems is the lookahead approach [27]. Lookahead solvers solve a problem via a binary search-tree. In each node of the search-tree, the best splitting variable is selected using so-called lookahead techniques. Although it is possible to extract unsatisfiability proofs from lookahead solvers, it hardly happens in practice and hence we ignore lookahead solvers in the remainder of this chapter.

CDCL solvers typically use a range of preprocessing techniques, such as bounded variable elimination (also known as Davis-Putnam resolution) [17, 18], blocked clause elimination [28], subsumption, and hyper binary resolution [7]. Preprocessing techniques are frequently crucial to solve large formulas efficiently. These preprocessing can also be used during the solving phase, which is known as inprocessing [29]. Most preprocessing techniques can be expressed using a few resolutions, such as bounded variable elimination and hyper binary resolution. Other techniques can be ignored in the context of unsatisfiability proofs, because they weaken the formula, such as blocked clause elimination and subsumption. A few techniques can only be expressed in extended resolution or its generalizations, such as bounded variable addition and blocked clause addition.

Some CDCL solvers use preprocessing techniques which are hard to represent using existing proof formats. Examples of such techniques are Gaussian Elimination (GE), Cardinality Resolution (CR) [16] and Symmetry Breaking (SB) [1]. These techniques cannot be polynomially simulated using resolution: Certain formulas based on expander graphs are hard for resolution [46], i.e., resolution proofs are exponentially large, while GE can solve them efficiently. Similarly, formulas arising from the pigeon hole principle are hard for resolution [23], but they can be solved efficiently using either CR or SB. Consequently, resolution proofs of solvers that use these techniques may be exponentially large in the size of the solving time. At the moment, there is no solver that produces resolution proofs for these techniques.

Techniques such as GE, CR, and SB, can be simulated polynomially using extended resolution and its generalizations. However, it is not know how to simulate these techniques efficiently / elegantly using extended resolution. One method to translate GE into extended resolution proofs is to convert the GE steps into BDDs and afterwards translate the BDDs to extended resolution [41].

4 Proof Formats

Unsatisfiability proofs come in two flavors: resolution proofs and clausal proofs. A handful of formats have been designed for resolution proofs [55, 20, 9]. These formats differ in several details, such as whether the input formula is stored in the proof, whether resolution chains are allowed, and whether resolutions in the proofs must be ordered. This section focusses on the TraceCheck format which is the most widely used format for resolution proofs.

For clausal proofs, there is essentially only one format, called RUP (reverse unit propagation) [47]. RUP can be extended with clause deletion information [25], and with a generalization of extended resolution [26]. The format with both extensions is known as DRAT [54] which is backward compatible with RUP.

4.1 Resolution Proofs

The proof checker TraceCheck can be used to check whether a trace represents a piecewise regular input resolution proof. A trace is just a compact representation of general resolution proofs. In [8] a regular input resolution proof is called a trivial proof.

The parts of the proof which are regular input resolution proofs are called chains in the following discussion. The whole trace consists of original clauses and chains.

Note that input clauses in chains can still be arbitrary derived clauses with respect to the overall proof and do not have to be original clauses. We distinguish between original clauses of the CNF, which are usually just called input clauses, and input clauses to the chains. Since a chain can be seen as new proof rule, we call its input clauses 'antecedents' and the final resolvent just 'resolvent'.

The motivation for using this format is as in [8] that learned clauses in a DP solver can be derived by regular input resolution. A unique feature of TraceCheck is that the chains do not have to be sorted, neither between chains (globally) nor their input clauses (locally). If possible the checker will sort them automatically. This allows a simplified implementation of the trace generation.

Chains are simply represented by the list of their antecedents and the resolvent. Intermediate resolvents can be omitted which saves quite some space if the proof generator can easily extract chains.

Chains can be used in the context of searched based CDCL to represent the derivation of learned clauses. It is even more difficult to extract a resolution proof directly, if more advanced learned clause optimizations are used. Examples are shrinking or minimization of learned clauses. The difficult part is to order the antecedents correctly. The solver can leave this task to the trace checker.

Furthermore, this format allows a simple encoding of hyper resolution proofs. A hyper resolution step can be simulated by a chain. General resolution steps can also be encoded in this format easily by a trivial chain consisting of the two antecedents of the general resolution step. Finally, extended resolution proofs can directly be encoded, since variables introduced in extended resolution can be treated in the same way as the original variables.

The syntax of a trace is as follows:

$$\begin{aligned}
\langle \text{trace} \rangle &= \{\langle \text{clause} \rangle\} \\
\langle \text{clause} \rangle &= \langle \text{pos} \rangle \langle \text{literals} \rangle \langle \text{antecedents} \rangle \\
\langle \text{literals} \rangle &= \text{“} * \text{”} \mid \{\langle \text{lit} \rangle\} \text{“0”} \\
\langle \text{antecedents} \rangle &= \{\langle \text{pos} \rangle\} \text{“0”} \\
\langle \text{lit} \rangle &= \langle \text{pos} \rangle \mid \langle \text{neg} \rangle \\
\langle \text{pos} \rangle &= \text{“1”} \mid \text{“2”} \mid \cdots \mid \langle \text{max}-\text{idx} \rangle \\
\langle \text{neg} \rangle &= \text{“} - \text{”} \langle \text{pos} \rangle
\end{aligned}$$

where | means choice, $\{\dots\}$ is equivalent to the Kleene star operation (that is a finite number of repetitions including 0) and $\langle \text{max}-\text{idx} \rangle$ is $2^{28} - 1$.

The interpretation is as follows. Original clauses have an empty list of antecedents and derived clauses have at least one antecedent. A clause definition starts with its index and a zero terminated list of its literals. This part is similar to the DIMACS format except that each clause is preceded by a unique positive number, the index of the clause. Another zero terminated list of positive indices of its antecedents is added, denoting the chain that is used to derive this clause as resolvent from the antecedents. The order of the clauses and the order of the literals and antecedents of a chain is arbitrary.

The list of antecedents of a clause should permit a regular input resolution proof of the clause with exactly the antecedents as input clauses. A proof is regular if variables are resolved at most once. It is an input resolution if each resolution step resolves at most one non input clause. Therefore it is also linear and has a degenerated graph structure of a binary tree, where each internal has at least one leaf as child.

input formula (DIMACS)

```
p cnf 3 6
-2  3 0
 1  3 0
-1  2 0
-1 -2 0
 1 -2 0
 2 -3 0
```

clausal proof (RUP)

```
-2  0
 3  0
 0
```

resolution proof (TraceCheck)

```
1 -2  3 0 0
2  1  3 0 0
3 -1  2 0 0
4 -1 -2 0 0
5  1 -2 0 0
6  2 -3 0 0
7 -2  0 4 5 0
8  3  0 1 2 3 0
9  0  6 7 8 0
```

Fig. 2. An input formula (left) in the classical DIMACS format which is supported by most SAT solvers. A clausal proof for the input formula in RUP format (middle). A TraceCheck file (right) is a resolution graph that includes the formula and proof. Each line begins with a clause identifier (bold), then contains the literals of the original clause or lemma, and ends with a list of clause identifiers (bold).

As example consider the following trace

```
1 -2  3 0 0
2  1  3 0 0
3 -1  2 0 0
4 -1 -2 0 0
5  1 -2 0 0
6  2 -3 0 0

7 -2  0 4 5 0
8  3  0 1 2 3 0
9  0  6 7 8 0
```

which consists of the six clauses from example formula E. The corresponding DIMACS formula is shown in Fig. 2 (left).

The first derived clause with index 7 is the unary clause which consists of the literal -2 alone. It is obtained by resolving the original clause 4 against the original clause 5 on variable 1.

A chain for the last derived clause, which is the empty clause ϵ, can be obtained by resolving the antecedents 6, 7 and 8, first 6 with 7 to obtain the intermediate resolvent consisting of the literal -3 alone, which in turn can be resolved with clause 8 to obtain ϵ.

As discussed above, the order of the clauses, that is the order of the lines and the order of the antecedents indices is irrelevant. The checker will sort them automatically. The last two lines of the example can for instance be replaced by:

```
9 0 6 7 8 0
8 3 0 1 2 3 0
```

Note that that the clauses 7 and 8 cannot be resolved together because they do not have a clashing literal. In this case the checker has to reorder the antecedents as in the original example.

It is also possible to skip the literal part for derived clauses by specifying a $\star$ instead of the literal list. The literals are then collected by the checker from the antecedents. Since resolution is not associative, the checker assumes that the antecedents are correctly sorted when $\star$ is used.

```
8 ⋆ 1 2 3 0
9 ⋆ 6 7 8 0
```

Furthermore, trivial clauses and clauses with multiple occurrences of the same literal can not be resolved. The list of antecedents is not allowed to contain the same idx twice. All antecedents have to be used in the proof for the resolvent.

Beside these local restrictions the proof checker generates a global linear order on the derived clauses making sure that there are no cyclic resolution steps. The roots of the resulting DAG are the target resolvents.

4.2 Clausal Proofs

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.

$$\begin{aligned}
\langle\text{proof}\rangle &= \{\langle\text{lemma}\rangle\}\\
\langle\text{lemma}\rangle &= \langle\text{delete}\rangle\{\langle\text{lit}\rangle\}\text{“0”}\\
\langle\text{delete}\rangle &= \text{“”} \mid \text{“}\mathtt{d}\text{”}\\
\langle\text{lit}\rangle &= \langle\text{pos}\rangle \mid \langle\text{neg}\rangle\\
\langle\text{pos}\rangle &= \text{“1”} \mid \text{“2”} \mid \cdots \mid \langle\text{max} - \text{idx}\rangle\\
\langle\text{neg}\rangle &= \text{“} - \text{”}\langle\text{pos}\rangle
\end{aligned}$$

There exists four proof formats for clausal proofs which have mostly the same syntax and all of them can be expressed using the grammar above.

The most basic format is RUP (reverse unit propagation) [47]. A RUP proof is a sequence of lemmas, with each lemma being a list of positive and negative integers to express positive and negative literals, respectively, which are terminated with a zero.

Given a formula F, and a clausal proof $P := \{L_1, \dots, L_m\}$. P is a valid RUP proof for F if $L_m = \epsilon$ and for all L_i holds that

$$F \wedge L_1 \wedge \cdots \wedge L_{i-1} \wedge \bar{L}_i \vdash_1 \epsilon$$

Recall the example formula E. The proof $P_E := \{(\bar{b}), (c), \epsilon\}$ is a valid proof for E, because P_E terminates with ϵ and

$$\begin{aligned}
E \wedge (b) &\vdash_1 \epsilon\\
E \wedge (\bar{b}) \wedge (\bar{c}) &\vdash_1 \epsilon\\
E \wedge (\bar{b}) \wedge (c) &\vdash_1 \epsilon
\end{aligned}$$

The DRUP (delete reverse unit propagation) format [25] extends RUP by integrating clause deletion information into proofs. The main reason to add clause deletion information to a proof is to reduce the cost to validate a proof which will be discussed in Section 6.1. Clause deletion information is expressed using the prefix d.

4.3 Proofs with Extended Resolution

So far we only considered proof formats that validate techniques that can be simulated using resolution. Some SAT solver use techniques that cannot be simulated using resolution, such as blocked clause addition [31]. To validate these techniques, proof formats need to support a richer representation that includes extended resolution or one of its generalizations.

Resolution proofs, as the name suggests, can only be used to check techniques based on resolution. The TraceCheck format partially supports extended resolution in the sense that one can add the clauses from the extension rule using an empty list of antecedents. Hence these clauses are considered to be input clauses without actually validating them.

The RAT clausal proof format [26], which is syntactically the same as the RUP format, supports expressing techniques based on extended resolution and its generalizations. The difference between the RUP and RAT format is in the redundancy check that is computed in the checker for proofs in that format. A checker for RUP proofs validates whether a lemma is a RUP clause, while a checker of RAT proofs check whether each lemma is a RAT clause. The DRAT format [54] extends RAT by supporting clause deletion information.

Example 2. Consider the following CNF formula

$$G := (\bar{a} \vee \bar{b} \vee \bar{c}) \wedge (a \vee d) \wedge (a \vee e) \wedge (b \vee d) \wedge (b \vee e) \wedge (c \vee d) \wedge (c \vee e) \wedge (\bar{d} \vee \bar{e})$$

Fig. 3 shows G in the DIMACS format (left) using the conventional mapping from the alphabet to numbers, i.e., $(a\ 1)(\bar{a}\ \text{-}1)\ldots(e\ 5)(\bar{e}\ \text{-}5)$ and a DRAT clausal proof for G (middle). The proof for G uses a technique, called bounded variable addition (BVA) [34], that cannot be expressed using resolution steps. BVA can replace the first six binary clauses by five new binary clauses using a fresh variable f: $(f \vee a)$, $(f \vee b)$, $(f \vee c)$, $(\bar{f} \vee d)$, and $(\bar{f} \vee e)$. These new binary clauses are RAT clauses. Fig. 3 shows how easy it is to express BVA in the DRAT format: First add the new binary clauses, followed by deleting the old ones. After the replacement, the proof is short $\{(f), \epsilon\}$.

It is not clear how bounded variable addition can be expressed in a resolution-style format. Fig. 4 shows the main issue for the example formula G and the BVA based proof. The clauses $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$ are trivially redundant with respect to G, because G does not contain any clause with variable f. Assigning f to $\mathbf{t}$ would satisfy these three clauses. However, $(\bar{f} \vee d)$ and $(\bar{f} \vee e)$ are not trivially redundant with respect to G after the addition of $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$. There redundancy of $(\bar{f} \vee d)$ depends on the presence of $(a \vee d)$, $(b \vee d)$, and $(c \vee d)$. One option to express BVA is adding the dependency relationship to the proof, as suggested in [54]. This results in a TraceCheck$^+$ proof for which each lemma has either a list of antecedents or a list of dependencies. However, there exists no procedure yet to validate such a TraceCheck$^+$ proof.

4.4 Open Issues and Challenges in Proof Formats

It is common practice to store proofs on disk and we discussed various formats for this purposes. However, in many applications where proofs have to be further processed and are used subsequently or even iteratively, disk I/O is considered a substantial overhead. There are only few publically available SAT solvers, which keep proofs in memory. Beside the question, whether these proofs are stored as resolution or clausal proofs, and the technical challenge to reduce memory usage, partially addressed in [9, 5], there is also no common understanding on what kind of API should be used to manipulate proofs.

DIMACS formula

```
p cnf 5 8
-1 -2 -3 0
    1  4 0
    1  5 0
    2  4 0
    2  5 0
    3  4 0
    3  5 0
   -4 -5 0
```

DRAT clausal proof

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

TraceCheck$^+$ resolution proof

```
 1  1  4  0  0
 2  1  5  0  0
 3  2  4  0  0
 4  2  5  0  0
 5  3  4  0  0
 6  3  5  0  0
 7 -1 -2 -3  0  0
 8 -4 -5  0  0
 9  6  1  0  0
10  6  2  0  0
11  6  3  0  0
12 -6  4  0  1  3  5 0
13 -6  5  0  2  4  6 0
14  6  0  1  9 10 11 0
15  0  8 12 13 14  0
```

Fig. 3. Example formula G in the classical DIMACS format (left). A clausal proof for the input formula in DRAT format (middle). A TraceCheck$^+$ file (right) is a dependency graph that includes the formula and proof. Each line begins with a clause identifier (bold), then contains the literals of the original clause or lemma, and ends with a list of clause identifiers (bold).

Beside checking the proof online for testing and debugging, common operations might be, extracting a resolution proof from a clausal proof, generating interpolants, minimizing proofs, or to determine a clausal or a variable core. A generic API for traversing proof objects might also be useful. Last but not least it should be possible to dump these proofs to disk.

Related to reducing the memory usage of storing proofs is the question of a binary disk format for proofs, or specific compression techniques, as used in the version of `MiniSAT` with proof trace support or `PicoSAT`.

Finally, as SAT solving is at the core of state-of-the-art SMT solving and also used in theorem provers, producing and manipulating proofs for these more expressive logics will need to incorporate techniques discussed in this chapter. Interoperability, mixed formats, and APIs etc. are open issues.

5 Proof Production

Proof logging of unsatisfiability results from SAT solvers started in 2003 of both resolution proofs [55] and clausal proofs [22]. Resolution proofs are typically hard to produce and tend to require lots of overhead in memory, which in turn slows down a SAT solver. Emitting clausal proofs is easy and requires hardly any overhead on memory. We will first describe how to produce resolution proofs and afterwards how to produce clausal proofs.

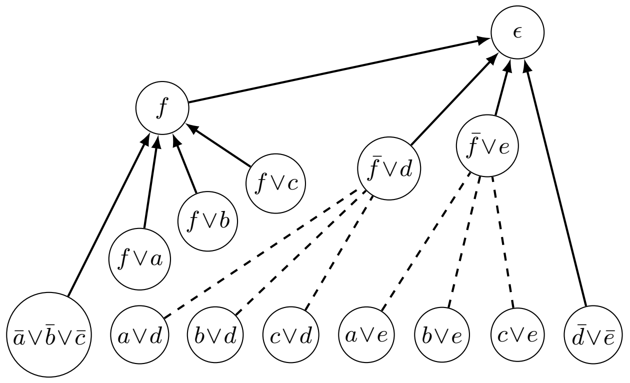

Fig. 4. A resolution-dependency graph illustrating a proof of example formula G. The clauses on bottom of the figure are the input clauses. The height of lemmas indicates the time that there were added to the proof: lower means earlier. Solid arrows represent resolution steps, while dashed lines represent a dependency relationship.

5.1 Resolution Proofs

The main motivation for adding proof support to `PicoSAT` was to make testing and debugging more efficient. In particular in combination with file based delta-debugging, proof trace generation allows to reduce discrepancies much more than without proof tracing.

The original idea was to use resolution traces. Originally it was however unclear how to extract resolution proofs during a-posteriori clause minimization [44]. This was the reason to use a trace format instead: clause minimization is obviously compatible with RUP, since required clause antecedents can easily be obtained. However, determining the right resolution order for a resolution proof is hard to generate directly and probably needs a RUP style algorithm anyhow. In [48] it was shown how clause minimization can be integrated into as DFS search for the first unique implication point clause, which at the same can produce a resolution proof for the minimized learned clause. Currently it is unsolved how to further extend this approach to work with on-the-fly subsumption [24] as well.

5.2 Reducing Memory Consumption

As already discussed above, memory consumption of proofs stored in memory (or disk) can become a bottle neck. As pioneered by the (unpublished) disk format for proof traces for `MiniSAT`, and extended in the in-memory format for proof traces of `PicoSAT`, the antecedent clauses of a learned clause can be sorted, as well as literals of learned clauses. After sorting, the antecedent lists or literals can be stored as differences between literals or antecedent ids instead of absolute values. Then these differences can be encoded efficiently in byte sequences of variadic length. In practice this technique needs slightly more then one byte on average per antecedent (or literal) [9]. Experience shows that other classical compression techniques are also more effective after this byte-encoding, such that in combination a large reduction can be achieved.

Another option to reduce space requirements for storing proofs in memory, is to remove garbage clauses in the proof, which are not used anymore. This garbage collection was implemented with saturating reference counters in `PicoSAT` [9]. In [5] it has been shown that full reference counting can result in substantial reductions. In principle it might also be possible to use a simple mark and sweep garbage collector, which should be faster, since it does not need to maintain and update reference counters.

5.3 Clausal Proofs

For all clausal proof formats, SAT solvers emit proofs directly to disk. Consequently, there is no memory overhead. In contrast to resolution proofs, it is easy to emit clausal proofs. For the most simple clausal proof format, RUP, one only needs to extend the proof with all added lemmas. This can be implemented by a handful lines of code. Below we discuss how to produce clausal proof formats that support deletion information and techniques based on generalized extended resolution.

Clausal proofs need deletion information for efficient validation, see Section 6.1 for details. Adding deletion information to a clausal proof is simple. As soon as a clause is removed from the solver, the proof is extended by the removed clause using a prefix expressing that it is a deletion step. If a solver removes a literal l from a clause C, then $C \setminus \{l\}$ is added as a lemma followed by deleting C.

Most techniques based on extended resolution or its generalizations can be easily be expressed in clausal proofs. Similar to techniques based on resolution, for most techniques, one simply adds the lemmas to the proof. The RAT and DRAT formats only require that the pivot literal is the first literal of the lemma in the proof. However, as discussed in Section 3, there exists some techniques for which it is not known whether they can be elegantly be expressed in the DRAT format. Especially Gaussian elimination, cardinality resolution and symmetry breaking are hard to express in the current formats.

6 Proof Consumption

Although resolution proofs are harder to produce than clausal proofs, they are in principle easy to check and actually needed for some applications, like generating interpolants. However, the large size of resolution proofs provides challenges, particular with respect to memory usage. See [49] for a discussion on checking large resolution proofs.

Clausal proofs are smaller, but validating clausal proofs is more complicated and more costly. Proofs are checked with dedicated tools, such as `stc` (short for simple tracecheck) for resolution proofs in TraceCheck format and `DRAT-trim` [54] for clausal proofs in DRAT format (and consequently in RUP, DRUP, and RAT formats due to backward compatibility).

6.1 Clausal Proofs

Clausal proofs are checked using unit propagation. Recall that a clausal proof $\{L_1, \ldots, L_m\}$ is valid for formula F, if $L_m = \epsilon$ and for $i \in \{1, \ldots, m\}$ holds that

$$F \wedge L_1 \wedge \cdots \wedge L_{i-1} \wedge \bar{L}_i \vdash_1 \epsilon$$

The most simple, but very costly method to validate clausal proofs is to check for every $i \in \{1, \ldots, m\}$ the above equation holds.

One can reduce the cost to validate clausal proofs by checking them *backwards* [22]: Initially, only $L_m = \epsilon$ is marked. Now we loop over the lemmas in backwards order, i.e., $L_m, \ldots, L_1$. Before validating a lemma, we first check whether it is marked. If a lemma is not marked, it can be skipped, thereby reducing the computational costs. If a lemma is marked, we check whether the clause satisfies the above equation. If the check fails, the proof is invalid. Otherwise, we mark all clauses that were required to make the check succeed (using conflict analysis). For most proofs, over half the lemmas can be skipped during validation.

The main challenge regarding validating clausal proofs is efficiency. Validating a clausal proof is typically much more expensive than obtaining the proof using a SAT solver, even if the implementation uses backwards checking and the same data-structures as state-of-the-art solvers. There are two main reasons why checking is more expensive than solving.

First, SAT solvers aggressively delete clauses during solving, which reduces the cost of unit propagation. If the proof checker has no access to the clause deletion information, then unit propagation is much more expensive in the checker as compared to the solver. This was the main motivation why the proof formats DRUP and DRAT have been developed. These formats support expressing clause deletion information, thereby making the unit propagation costs between the solver and checker similar.

Second, SAT solvers reuse propagations in between conflicts, while a proof checker does not reuse propagations. Consider two consecutive lemmas L_i and L_{i+1} produced by a SAT solver. In the most extreme, but not unusual case, the branch that resulted in L_i and L_{i+1} may differ only in a single decision (out of many decisions). Hence most propagations will be reused in the solver. At the same time, it may be that the lemmas have no overlapping literals, i.e., $L_i \cap L_{i+1} = \emptyset$. Consequently, the checker would not be able to reuse any propagations. In case $L_i \cap L_{i+1}$ is nonempty, the checker could potentially reuse propagations, although no clausal proof checker implementation exploits this.

While checking a clausal proof, one can easily produce an unsatisfiable core and a resolution proof. The unsatisfiable core consists of the original clauses that were marked during backwards checking. For most unsatisfiable formulas that arise from applications, many clauses are redundant, i.e, are not marked and thus not in the unsatisfiable core. The resolution proof has for each marked lemma all the clauses that were required during its validation as antecedents.

The resolution proof that is produced by clausal proof checking might differ significantly from the resolution proof that would correspond with the actions of the SAT solver that emitted the clausal proof. For example, the resolution proof for the example formula E might be equal to the resolution graph shown in Fig. 1. On the other hand, the resolution proof produced by clausal proof checking might be equal to Fig. 5. Notice that the resolution graph of Fig. 5 (right) does not use all original clauses. Clause $(\bar{b} \vee c)$ is redundant and not part of the core of E.

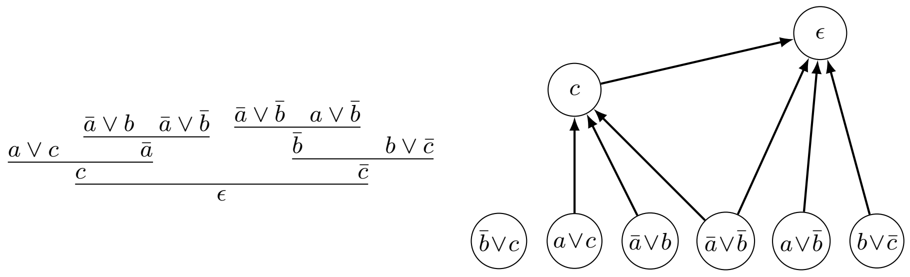

Fig. 5. A resolution derivation (left) and a resolution graph (right) for an example formula produce by checking a clausal proof.

7 Proof Applications

Proofs of unsatisfiability have been used to validate the results of SAT competitions[4]. Initially, during the SAT competition of 2007, 2009, and 2011, a special track was organized for which the unsatisfiability results were checked. For the SAT competitions of 2013 and 2014, proof logging became mandatory for tracks with only unsatisfiable benchmarks. The supported formats for SAT competition 2013 were TraceCheck and DRUP, but all solvers participating in these tracks opted for the DRUP format. For SAT competition 2014, the only supported format was DRAT, which is backwards compatible with DRUP.

As already mentioned above, one motivation for using proofs is to make testing and debugging of SAT solvers more effective. Checking learned clauses online with RUP allows to localize unsound implementation defects as soon they lead to clauses, which are not implied by reverse unit propagation.

Testing with forward proof checking is particularly effective in combination with fuzzing and delta-debugging [14]. Otherwise failures produced by unsound

[4] see `http://www.satcompetition.org` for details.

reasoning can only be observed if they turn a satisfiable instance into an unsatisfiable one. This situation is not only difficult to produce, but also tends to lead to much larger input files after delta-debugging.

However, model based testing [4] of the incremental API of a SAT solver is in our experience at least as effective as file based fuzzing and delta-debugging. More recently we added online proof checking capabilities to `Lingeling`, which allows to combine these two methods (model based testing and proof checking).

Probably the most important aspect of proof tracing is that it allows to generate clausal (or variable) cores. These cores in turn can be used in many applications, including MUS extraction [39], MaxSAT [37], diagnosis [42, 40], for abstraction refinement in model checking [19] or SMT [2, 13]. Note that this list of references is subjective and by far not complete. It should only be considered as a starting point for investigating related work on using cores.

Finally, an important usage of resolution proofs, is for extracting interpolants, particularly in the context of interpolation based model checking [36]. Since resolution proofs are large and not easy to obtain, there has been several recent attempts to avoid proofs and obtain interpolants directly, see for instance [50]. Interpolation based model checking became the state-of-the-art until the invention of IC3 [12]. The IC3 algorithm is also based on SAT technology, and also uses cores, but usually in a much more light weight way. Typical implementations use assumption based core techniques as introduced in `MiniSAT` [20] (see also [33]) instead of proof based techniques.

8 Conclusions

Unsatisfiability proofs are useful for several applications, such as computing interpolants and MUS extraction. These proofs can also be used to validate results of the SAT solvers that produced them and for tools that use SAT solvers, such as theorem provers.

There are two types of unsatisfiability proofs: resolution proofs and clausal proofs. Resolution proofs are used for most applications, but they are hard to produce. Therefore very few SAT solvers support resolution proof logging. Clausal proof logging is easy and therefore most state-of-the-art solvers support it. However, validating clausal proofs is costly, although recent advances significantly improved performance of checkers.

There are several challenges regarding unsatisfiability proofs. How can one store resolution proofs using much less space on disk and using much less memory overhead? Can the costs of validating clausal proofs be further be reduced? Last but not least, research is required to study how some techniques, such as Gaussian elimination, cardinality resolution, and symmetry breaking, can be expressed elegantly is unsatisfiability proofs.

References

1. Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006.

2. Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proc. ASP-DAC'06*, pages 19–24. IEEE, 2006.
3. M Armand, G Faure, B Grégoire, C Keller, L Théry, and B Wener. 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.
4. Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. In Margus Veanes and Luca Viganò, editors, *TAP*, volume 7942 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2013.
5. Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in sat. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.
6. Gilles Audemard and Laurent Simon. Glucose 2.3 in the SAT 2013 Competition. In A. Belov, M. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 42–43, 2013.
7. Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In Giunchiglia and Tacchella [21], pages 341–355.
8. Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *JAIR*, 22:319–351, 2004.
9. Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.
10. Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT Competition 2013. In A. Belov, M. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 51–52, 2013.
11. Roderick Bloem and Natasha Sharygina, editors. *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. IEEE, 2010.
12. Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
13. Robert Brummayer and Armin Biere. Effective bit-width and under-approximation. In *Proc. EUROCAST'09*, volume 5717 of *LNCS*, pages 304–311, 2009.
14. Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *International Workshop on Satisfiability Modulo Theories (SMT)*, pages 1–5. ACM, 2009.
15. Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):pp. 36–50, 1979.
16. W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25 – 38, 1987.
17. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
18. Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

19. Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental sat formulation of proof- and counterexample-based abstraction. In Bloem and Sharygina [11], pages 181–188.
20. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Giunchiglia and Tacchella [21], pages 502–518.
21. Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*. Springer, 2004.
22. Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 10886–10891. IEEE, 2003.
23. Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.
24. HyoJung Han and Fabio Somenzi. On-the-fly clause improvement. In Kullmann [32], pages 209–222.
25. Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188. IEEE, 2013.
26. Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *International Conference on Automated Deduction (CADE)*, volume 7898 of *LNAI*, pages 345–359. Springer, 2013.
27. Marijn J. H. Heule and Hans van Maaren. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter Chapter 5, Look-Ahead Based SAT Solvers, pages 155–184. IOS Press, February 2009.
28. Matti Järvisalo, Armin Biere, and Marijn Heule. Simulating circuit-level simplifications on cnf. *J. Autom. Reasoning*, 49(4):583–619, 2012.
29. Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.
30. Boris Konev and Alexei Lisitsa. A sat attack on the erdos discrepancy conjecture. 2014. Accepted for SAT 2014.
31. Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999.
32. Oliver Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer, 2009.
33. Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up mus extraction. In Matti Järvisalo and Allen Van Gelder, editors, *SAT*, volume 7962 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2013.
34. Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of boolean formulas. In *Proceedings of Haifa Verification Conference (HVC)*, volume 6397 of *LNCS*, pages 102–117. Springer, 2012.
35. Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter Chapter 4, Conflict-Driven Clause Learning SAT Solvers, pages 131–153. IOS Press, February 2009.
36. Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

37. António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and Joao Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.
38. Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Bloem and Sharygina [11], pages 221–229.
39. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient mus extraction with resolution. In *FMCAD*, pages 197–200. IEEE, 2013.
40. Alexander Nöhrer, Armin Biere, and Alexander Egyed. Managing SAT inconsistencies with HUMUS. In *Proc. VaMoS'12*, pages 83–91. ACM, 2012.
41. Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.
42. Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Artif. Intell. Eng. Des. Anal. Manuf.*, 17(1):75–97, January 2003.
43. Mate Soos. Strangenight. In A. Belov, M. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 89–90, 2013.
44. Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Kullmann [32], pages 237–243.
45. Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2*, pages 466–483. Springer, 1983.
46. Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.
47. Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.
48. Allen Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In Kullmann [32], pages 141–146.
49. Allen Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
50. Yakir Vizel, Vadim Ryvchin, and Alexander Nadel. Efficient generation of small interpolants in cnf. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 330–346. Springer, 2013.
51. T Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In *International Workshop on the Implementation of Logics (IWIL)*, volume 212, pages 44–62, 2006.
52. T Weber and H Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26–40, 2009.
53. Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. Mechanical verification of SAT refutations with extended resolution. In *Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 229–244. Springer, 2013.
54. Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, 2014.
55. Lintao Zhang and Sharad Malik. 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, pages 10880–10885. IEEE Computer Society, 2003.