Chapter preprint of the second edition of the Handbook of Satisfiability (2021)
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
Published by IOS Press. The print edition can be ordered at
https://www.iospress.nl/book/handbook-of-satisfiability-2/
The published electronic chapters (Version of Record) are available at
https://ebooks.iospress.nl/ISBN/978-1-64368-161-0/

Chapter 15

Proofs of Unsatisfiability

Marijn J.H. Heule

15.1. Introduction

Satisfiability (SAT) solvers have become powerful tools to solve a broad spectrum of applications. Since the SAT problem is NP-complete, it is easy to validate a satisfiability claim using a certificate in the form of a variable assignment. However, if a solver reports that a formula has *no* solutions, the validation of that claim requires a proof of unsatisfiability. The correctness of SAT solving results is important for applications ranging from hardware verification to proving mathematical theorems. Proof checking is significantly less complex compared to the implementations of state-of-the-art SAT solvers, allowing efficient validation even by formally verified checkers [CFHH$^+$17, Lam20]. Proofs of unsatisfiability are useful in several applications, including interpolation [VRN13], extraction of minimal unsatisfiable sets (MUSes) [Nad10], and several areas that rely on SAT solvers, such as theorem proving [AFG$^+$11, Web06, WA09, WHH13]. This chapter covers the practical aspects of proofs of unsatisfiability, while Chapter 7 on proof complexity discusses the theoretical aspects in more detail.

Many proof formats have been proposed [ZM03, ES03, Bie08, Van08, WHH14, CFHH$^+$17, Lam20, HKB19], and even though they might seem very different, they actually have a lot in common. Essentially all well-known proofs of unsatisfiability are so-called *clausal proofs*: sequences of clauses that are claimed to be redundant with respect to a given formula, where *redundant* means that the addition of a clause preserves the satisfiability status of the formula. The valid addition of the (trivially unsatisfiable) empty clause, usually in the final proof step, shows that the formula is unsatisfiable. In addition, the validity of each step should be efficiently checkable. The main difference between existing proof formats is whether so-called *hints* are included. A hint is information that a checker can use to validate the correctness of a clause-addition step more efficiently. Another difference is whether or not the formula is included in a proof.

The generation of unsatisfiability proofs in practical SAT solving dates back to 2003, when Zhang and Malik integrated logging of resolution steps with hints in the SAT solver `zChaff` [ZM03], while Goldberg and Novikov enhanced the solver `Berkmin` [GN03] with logging of proofs without hints. For clausal proofs based on the resolution proof system, the hints refer to the antecedents of the resolution rule. Without the antecedents, a checker needs to search for clauses that can be resolved together to justify a clause addition step. Although the antecedents can be determined in polynomial time, this can be expensive in practice. On the other hand, including hints in proofs can make them much larger and render the proof production more complicated and expensive.

In recent years, clausal proofs have also been used to validate results of strong proof systems that go beyond resolution. An example of such a proof system is extended resolution [Tse83], which allows the addition of definitions to a formula. Such definitions consist of multiple clauses. In a clausal proof, these clauses are listed in an arbitrary order and validity is checked per clause. Some strong proofs systems allow the addition of clauses for which checking the validity is NP-complete [HKB19] if no hint—usually called *witness* in that context—is provided. As a consequence, witnesses are mandatory whereas hints are not.

Whether or not to include hints in proofs typically depends on the usage of the proofs. For various applications that require resolution proofs, such as interpolation [McM03] or MUS extraction [NRS13], it is common to produce proofs with hints. Proofs without hints are more popular in the context of validating SAT solving results. For example, to check the results of the SAT Competitions or the proofs of mathematical theorems, such as the Erdős Discrepancy Theorem [KL14], the Pythagorean Triples Problem [HKM16], and Keller's Conjecture [BHMN20]. There are exceptions: proofs without hints have also been studied for interpolation [GV14] and MUS extraction [BHMS14].

Support for proof logging started to become widespread in state-of-the-art solvers such as `Lingeling` [Bie13], `Glucose` [AS13], and `CryptoMiniSAT` [Soo13], in 2013, when the SAT Competition made unsatisfiability proofs mandatory for solvers participating in the unsatisfiability tracks. Practically all sequential solvers that participated in recent SAT Competitions, including the strongest solvers around (such as the three solvers mentioned above) emit proofs *without* hints, and only very few solvers support emitting proofs *with* hints.

The lack of support for proofs with hints is due to the overhead (both in CPU and memory) and the difficulty to represent some techniques used in contemporary SAT solvers in terms of resolution. One such technique is conflict clause minimization [SB09], which requires additional bookkeeping to produce a resolution proof [Van09]. Computing the resolution steps increases the runtime on both satisfiable and unsatisfiable formulas. In contrast, emitting a proof without hints incurs only a negligible overhead and requires just modest modifications for most solvers.[1] Proof validation is only required if the solver claims unsatisfiability.

Support for proof logging has become mainstream in recent years and it is now common practice to produce and validate a proof when using SAT technology to solve hard problems, such as the mathematical problems we mentioned earlier. The development of formally-verified proof checkers further increases the confidence in the correctness of results. The high level of trust in the results is important as SAT solvers are increasingly often used to check the correctness of hardware and software, and such claims should not be based on bugs.

[1]A patch to add logging of proofs without hints to `MiniSAT` and `Glucose` is available at `https://github.com/marijnheule/drup-patch`.

15.2. Proof Systems

A proof of unsatisfiability is a sequence of clauses that are *redundant* with respect to a given formula. The most general notion of redundancy is satisfiability preservation, which requires that the addition of a clause to a formula does not affect satisfiability. Under this notion of redundancy, every clause is trivially redundant with respect to every unsatisfiable formula. However, unless P = NP, there exists no polynomial-time algorithm that can check the validity of clause-addition steps under this general notion of redundancy. Because of this, several proof systems have been invented that rely on simpler syntactic criteria that guarantee redundancy while still being efficiently checkable. In this section, we discuss several such proof systems that allow the addition of clauses that are learned by practical SAT solvers or generated by typical preprocessing and inprocessing techniques.

15.2.1. Preliminaries and Notation

We consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is either a variable x (a *positive literal*) or the negation $\overline{x}$ of a variable x (a *negative literal*). The *complement* $\overline{l}$ of a literal l is defined as $\overline{l} = \overline{x}$ if $l = x$ and $\overline{l} = x$ if $l = \overline{x}$. For a literal l, we denote the variable of l by $var(l)$. A *clause* is a finite disjunction of the form $(l_1 \vee \cdots \vee l_k)$, where $l_1, \ldots, l_k$ are literals. If not stated otherwise, we assume that clauses do not contain *complementary literals*, i.e., a literal and its complement. A *formula* is a finite conjunction $C_1 \wedge \cdots \wedge C_n$, where $C_1, \ldots, C_n$ are clauses. Clauses can be viewed as sets of literals and formulas as sets of clauses.

An *assignment* is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). An assignment is *total* with respect to a formula if it assigns a truth value to all variables occurring in the formula, otherwise it is *partial*. We often denote assignments by the sequences of literals they satisfy. For instance, $x\,\overline{y}$ denotes the assignment that makes x true and y false. We denote the domain of an assignment α by $var(\alpha)$. A literal l is *satisfied* by an assignment α if l is positive and $\alpha(var(l)) = 1$ or if it is negative and $\alpha(var(l)) = 0$. A literal is *falsified* by an assignment if its complement is satisfied by the assignment. A clause is satisfied by an assignment α if it contains a literal that is satisfied by α. Finally, a formula is satisfied by an assignment α if all its clauses are satisfied by α. A formula is *satisfiable* if there exists an assignment that satisfies it, and *unsatisfiable* otherwise.

We denote the empty clause by $\bot$ and the satisfied clause by $\top$. Given an assignment α and a clause C, we define $C\,|\,\alpha = \top$ if α satisfies C, otherwise $C\,|\,\alpha$ denotes the result of removing from C all the literals falsified by α. Moreover, for a formula F, we define $F\,|\,\alpha = \{C\,|\,\alpha \mid C \in F \text{ and } C\,|\,\alpha \neq \top\}$. We say that a clause C *blocks* an assignment α if $C = \{x \mid \alpha(x) = 0\} \cup \{\overline{x} \mid \alpha(x) = 1\}$.

A *unit clause* is a clause that contains only one literal. The result of applying the *unit-clause rule* to a formula F is the formula $F\,|\,\alpha$ with α being an assignment that satisfies a unit clause in F. The iterated application of the unit-clause rule to a formula, until no unit clauses are left, is called *unit propagation*. If unit propagation on a formula F yields the empty clause $\bot$, we say that it derived a *conflict* on F. For example, unit propagation derives a conflict on $F = (\overline{x} \vee y) \wedge (\overline{y}) \wedge (x)$ since $F|x = (y) \wedge (\overline{y})$ and $F|xy = \bot$.

Two formulas are *logically equivalent* if they are satisfied by the same total assignments. Two formulas F and F' are *equisatisfiable* if both F and F' are satisfiable or if both F and F' are unsatisfiable. By $F \vDash F'$, we denote that F implies F', i.e., every assignment that satisfies F and assigns all variables in $var(F')$ also satisfies F'. Furthermore, by $F \vdash_1 F'$ we denote that for every clause $(l_1 \vee \cdots \vee l_k) \in F'$, unit propagation derives a conflict on $F \wedge (\bar{l}_1) \wedge \cdots \wedge (\bar{l}_k)$. If $F \vdash_1 F'$, we say that F *implies* F' *via unit propagation.* As an example, $(x) \wedge (y) \vdash_1 (x \vee z) \wedge (y)$, since unit propagation derives a conflict on both $(x) \wedge (y) \wedge (\overline{x}) \wedge (\overline{z})$ and $(x) \wedge (y) \wedge (\overline{y})$.

Throughout this chapter we will use the following formula E as an example to explain various concepts:

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

15.2.2. Resolution

The resolution rule [Rob65] states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\overline{x} \vee b_1 \vee \ldots \vee b_m)$ with a complementary pair of literals (in this case x and $\overline{x}$), 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. The resolution proof system (RES) infers new clauses using the resolution rule. Chapter 7 covers the resolution proof system in more detail.

A *resolution chain* is a sequence of resolution operations such that the result of each 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, we have $\big((a \vee c) \diamond (\overline{a} \vee b)\big) \diamond (\overline{a} \vee \overline{b}) = (\overline{a} \vee c)$, while $(a \vee c) \diamond \big((\overline{a} \vee b) \diamond (\overline{a} \vee \overline{b})\big) = (c)$.

Let $C := (l_1 \vee l_2 \vee \cdots \vee l_k)$ be a clause. We denote by $\overline{C}$ the conjunction $(\bar{l}_1) \wedge (\bar{l}_2) \wedge \cdots \wedge (\bar{l}_k)$ of unit clauses. C is called a *reverse unit propagation* (RUP) clause with respect to F, if F implies C via unit propagation [Van08]. Remember that F implies C via unit propagation, denoted by $F \vdash_1 C$, if unit propagation derives a conflict on $F \wedge \overline{C}$. The prototypical RUP clauses are the learned clauses in conflict-driven-clause-learning (CDCL) solvers, which are the most common solvers (see Chapter 4 and Sect. 15.3). The conventional procedure for showing that these learned clauses are implied by the formula applies unit propagations in reverse of the order in which they were derived by the CDCL solver, which explains the name *reverse* unit propagation.

Clearly, if C is a RUP clause with respect to a formula F, then F implies C. This is an easy consequence of the fact that F implies C via unit propagation. In fact, it can be shown that one can construct a resolution chain for C using at most $|var(F)|$ resolutions. For example, $E \wedge (\overline{c}) \vdash_1 (\overline{b}) \vdash_1 (a) \vdash_1 \bot$ uses the clauses $(\overline{b} \vee c)$, $(a \vee c)$, and $(\overline{a} \vee b)$. We can convert this into a resolution chain $(c) := (\overline{a} \vee b) \diamond (a \vee c) \diamond (\overline{b} \vee c)$.

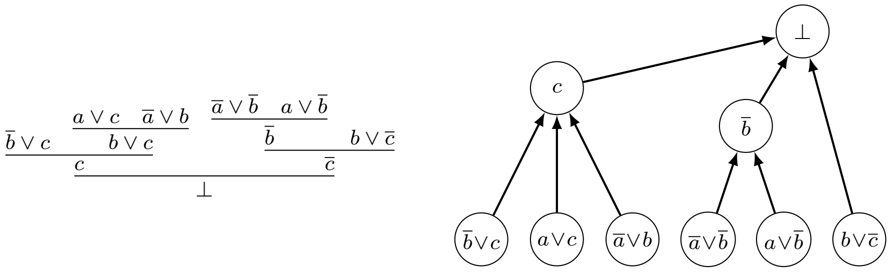


Figure 15.1. A resolution derivation (left) and a resolution graph (right) for the example CNF formula E.

The notion of RUP clauses gives rise to the RUP *proof system*, which consists of a single rule that allows the addition of RUP clauses. More specifically, a RUP proof of a clause C_m from a formula F is a sequence $C_1, \dots, C_m$ of clauses such that $F \wedge C_1 \wedge \cdots \wedge C_{i-1} \vdash_1 C_i$ for each $i \in 1, \dots, m$.

15.2.3. Extended resolution and its Generalizations

Extended resolution is a simple but powerful generalization of the ordinary resolution proof system, obtained by adding the so-called *extension rule* [Tse83]. Given a CNF formula F, the extension rule allows one to iteratively add definitions of the form $x \leftrightarrow a \wedge b$ by adding the *definition clauses* $(\overline{x} \vee a) \wedge (\overline{x} \vee b) \wedge (x \vee \overline{a} \vee \overline{b})$ to F, where x is a fresh variable (i.e., a variable that does not occur in the formula or in the previous part of the proof), and a and b are existing literals in the current formula or proof. The proof system of extended resolution (ER) [Tse83] consists of two rules: the resolution rule and the extension rule.

It can be easily seen that the extension rule is sound in the sense that it preserves satisfiability: Every satisfying assignment of a formula *before* addition of the definition clauses can be extended to a satisfying assignment of the formula *after* the addition of the definition clauses by setting the truth value of the fresh variable x to the truth value of $a \wedge b$.

Generalizations of extended resolution, which we discuss later, have a similar property: A satisfying assignment of the formula before addition of a redundant clause can be transformed into a satisfying assignment of the formula after addition of the clause. However, in these more general proof systems it is often not enough to simply extend the existing assignment—sometimes, the truth values of some literals need to be made true. This set of literals is often required as a *witness* (as mentioned earlier) to allow the validation of proofs in polynomial time.

The extended resolution proof system can polynomially simulate extended Frege systems [CR79], which is considered to be one of the most powerful proof systems. Extended resolution proofs can be exponentially smaller compared to resolution proofs. For example, Cook constructed short (polynomial-length) ER proofs for the pigeon hole formulas [Coo76], while Haken proved that resolution proofs of the pigeonhole principle must at least be exponential in size of the formula [Hak85]. However, searching for ER proofs is challenging as it is difficult to come up with effective applications of the extension rule.

This difficulty of finding useful applications of the extension rule is also a reason why several generalizations of extended resolution have been proposed, in part to make it easier to find short proofs without the need to introduce new variables. The first of these generalizations is the *blocked clause* (BC) *proof system* [Kul99]. The concept of blocked clauses can be generalized to conditional autarky clauses [HKSB17], which give rise to two other proof systems.

15.2.3.1. 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 $D \in F$ with $\overline{l} \in D$, $C \diamond_l D$ is a tautology, or (ii) $\overline{l} \in C$, 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 preserves satisfiability of formulas [Kul99].

Example 15.1. Recall the example formula E. Clause $(\overline{b} \vee c)$ is blocked on c with respect to E, because applying the resolution rule on the only clause containing $\overline{c}$ results in a tautology, i.e., $(\overline{b} \vee c) \diamond (b \vee \overline{c}) = (\overline{b} \vee b)$. Since we know that E is unsatisfiable, $E \setminus \{(\overline{b} \vee c)\}$ must be unsatisfiable.

To see that blocked clause addition is a generalization of extended resolution, consider a formula without variable x, but that contains variables a and b. The three definition clauses from the extension rule, i.e, $(\overline{x} \vee a)$, $(\overline{x} \vee b)$, and $(x \vee \overline{a} \vee \overline{b})$ are all blocked on x or $\overline{x}$, regardless of the order in which they are added. This is an easy consequence of the fact that x is fresh and that the only resolvents upon x or $\overline{x}$ are obtained by resolving the definition clauses with each other. 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* that do not contain a fresh variable. For example, consider the formula $F := (a \vee b)$. The clause $(\overline{a} \vee \overline{b})$ is blocked on $\overline{a}$ (and $\overline{b}$) with respect to F and can thus be added using blocked clause addition.

15.2.3.2. Conditional Autarkies

An *autarky* is a partial assignment that satisfies all clauses it touches, i.e., all clauses for which the assignment assigns a truth value to at least one literal. Simple examples of autarkies are pure literals and satisfying assignments of a whole formula. Chapter 14 discusses autarkies in detail. A *conditional autarky* [HKSB17] is a partial assignment that consists of a so-called *conditional part* and an *autarky part*, where the autarky part is an autarky for the formula restricted under the conditional part.

Example 15.2. Consider the formula $F = (x \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z})$. The assignment $\omega = \overline{y}\, z$ is a conditional autarky with conditional part $\omega_{\text{con}} = \overline{y}$: By applying ω_{con} to F, we obtain the formula $F\,|\,\overline{y} = (x) \wedge (\overline{x} \vee z)$. The only clause of $F\,|\,\overline{y}$ that is touched by ω is the clause $(\overline{x} \vee z)$, which is satisfied by ω. The literal z is a conditional pure literal with respect to ω_{con}.

Conditional autarkies give rise to redundant clauses [KHB19]: If the assignment $c_1 \ldots c_m \, a_1 \ldots a_n$ is a conditional autarky (with conditional part $c_1 \ldots c_m$) for a formula F, then F and $F \cup \{(c_1 \vee \cdots \vee c_m \vee a_i) \mid 1 \leq i \leq n\}$ are equisatisfiable. We refer to the redundant clauses that can be added using this reasoning as conditional autarky clauses. It turns out that blocked clauses are conditional autarky clauses [HKSB17] with the blocking literal being the autarky part, while the other literals form the conditional part of the conditional autarky.

By allowing the autarky part to consist of multiple literals, the notion of blocked clauses can be generalized to *set-blocked clauses* [KSTB16] and *globally-blocked clauses* [KHB19]. Set-blocked clauses are conditional autarky clauses for which all literals in the autarky part are included in the clause. Globally-blocked clauses are conditional autarky clauses for which at least one literal in the autarky part is present in the clause. An example of these clauses is described below.

Example 15.3. Consider again the formula $F = (x \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z})$. The clause $(x \vee \overline{y} \vee z)$ is a set-blocked clause with respect to F with witness $\omega = x\,\overline{y}\,z$. The clause (x) is a globally-blocked clause with respect to F with witness $\omega = x\,\overline{y}\,z$. Notice that ω is even an autarky for F.

Deciding if a clause is set-blocked or globally-blocked with respect to a formula is an NP-complete problem [KSTB16]. To efficiently validate the set-blockedness or globally-blockedness property, a witnessing assignment (or witness in short) is required. The witness is the autarky part of corresponding conditional autarky.

The set-blocked clause (SBC) proof system combines the resolution rule with the addition of set-blocked clauses, while the globally-blocked clause (GBC) proof system combines the resolution rule with the addition of globally-blocked clauses.

15.2.4. Strong Proof Systems

In the last decade, several strong proof systems have been proposed that combine the strengths of RUP (easy to emit and compact) with the generalizations of ER (expressive). These proof systems allow for short proofs for hard problems and can compactly express all techniques used in top-tier SAT solvers. The first proof system in this direction is RAT [JHB12]. Two recent generalizations are PR [HKB19] and SR [BT19].

15.2.4.1. Resolution Asymmetric Tautologies

resolution asymmetric tautologies (or RAT clauses) [JHB12] are a generalization of both RUP clauses and blocked clauses. Remember that a blocked clause is a clause that contains a literal such that all resolvents upon this literal are tautologies. If we do not require the resolvents to be tautologies, but instead allow them to be RUP clauses (remember that every tautology is a RUP clause), we obtain the notion of a RAT clause: A clause C is a RAT on a literal l (called the witness or pivot literal) with respect to a formula F if for all $D \in F$ with $\overline{l} \in D$, it holds that $F \vdash_1 C \diamond D$.

It can be shown that the addition and removal of RAT clauses does not affect the satisfiability (or unsatisfiability) of a formula [JHB12]. More specifically, given a formula F and a clause C that is a RAT on $l \in C$ with respect to F, let α be an assignment that satisfies F and falsifies C. The assignment ω, which is a copy of α with the exception that $\omega(l) = 1$, satisfies $F \wedge C$. While this is not supposed to be obvious, the proof of this observation is quite simple. The observation can be used to reconstruct a satisfying assignment for the original formula in case it is satisfiable. Details about solution reconstruction are described in Section 9.5 in Chapter 9.

We have already discussed why RAT clauses are a generalization of blocked clauses in the sense that every blocked clause is also a RAT clause. It can also be shown that RAT clauses generalize RUP clauses, based on the observation that whenever a clause C is a RUP clause with respect to a formula F, then C is a RAT on each of its literals. Note that this assumes that the considered clause is non-empty. As the empty clause does not contain any literals that could ensure the RAT property, it is often still considered a RAT clause "by definition".

The RAT proof system consists of a single clause addition rule: the addition of RAT clauses. This proof system extends the RUP proof system by allowing to compactly express techniques that go beyond resolution, including bounded variable addition [MHB12] and symmetry breaking [HHJW15].

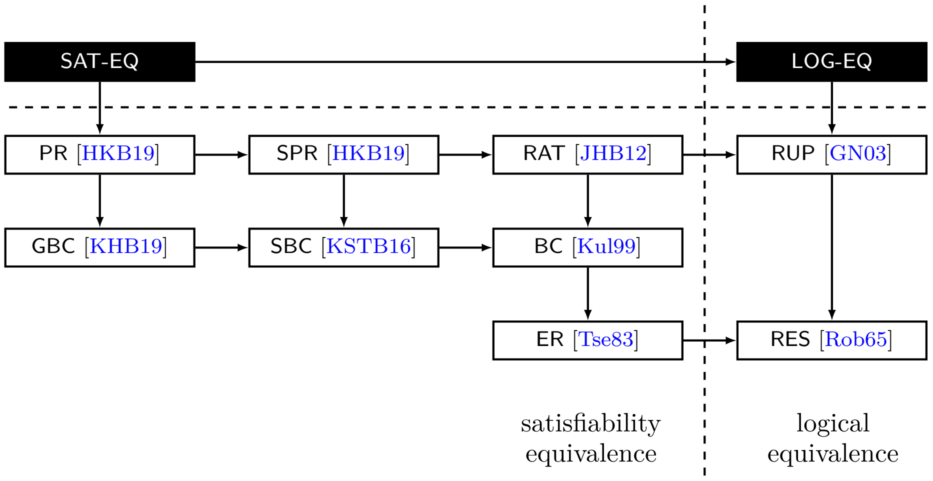


Figure 15.2. Relations between proofs systems for propositional logic. An arrow pointing from proof system A to proof system B means that A is a generalization of B. The black boxes on the top are not actual proof systems, but refer to abstract systems that would allow any clause-addition step that preserves satisfiability (SAT-EQ) or logical equivalence (LOG-EQ).

15.2.4.2. Propagation Redundancy

Another way to express the RAT property is as follows: Given a clause C and a formula F, let α denote the smallest assignment that falsifies C. Now, C is a RAT clause on literal l with respect to F if and only if $F|\alpha \vdash_1 F|l$ [HKB19]. This notion can be generalized by using a set of literals instead of the single literal l.

A clause C has the *set-propagation redundancy* (SPR) property with respect to a formula F if and only if $F|\alpha \vdash_1 F|L$, with α denoting the smallest assignment that falsifies C, and L denoting a set of literals that all occur in C [HKB19]. The set L is the witness of the SPR property of C. SPR clauses generalize both set-blocked clauses and RUP clauses in a similar way that RAT clauses generalize blocked clauses and RUP clauses. The SPR proof system uses a single rule: the addition of SPR clauses. The SPR proof system is surprisingly powerful even when it is restricted such that no new variables are allowed to be introduced, known as SPR$^-$. For example, there exist short SPR$^-$ proofs for many formulas that are hard for resolution, including the pigeonhole principle, parity principle, Tseitin tautologies, and clique-coloring tautologies [HKB19, BT19].

Finally, the *propagation redundancy* (PR) property generalizes the redundant clauses discussed in this section. A clause C has the PR property with respect to a formula F if and only if $F|_\alpha \vDash_1 F|_\omega$, with α denoting the smallest assignment that falsifies C, and ω being an arbitrary witness assignment that satisfies C [HKB19]. The PR proof system allows the addition of PR clauses. Figure 15.2 shows a hierarchy of proof systems with PR being the most general one. There exists a polynomial simulation between all proof systems that go beyond logical equivalence, from ER to PR [KRPH18, HB18]. That does not mean that they are similar in practice. A polynomial blowup from PR to ER could mean an enormous increase in validation costs. Also, it is easier to search for proofs in the more general systems as they admit short proofs *without new variables* for some hard problems [HKB19].

15.2.5. Clause Deletion

All the proof systems considered so far in this section add clauses to a given formula. SAT solvers, however, combine both the addition and the deletion of clauses. Most of the discussed proof systems thus have corresponding systems that extend them by allowing the deletion of clauses. The most popular one of these proof systems is probably DRAT, which allows the addition of RAT clauses as well as the deletion of arbitrary clauses. It is important to support the deletion of clauses in proofs without hints for two reasons.

First, it is practically impossible to efficiently validate a proof without hints if no clauses are removed: Due to the lack of hints, the checker needs to find the antecedents, which becomes increasingly expensive as clauses are added to the formula. Even with clause deletion it is generally more expensive to validate the proof compared to its construction. This topic is further discussed in Section 15.6.2.

Second, clause deletion allows for a compact simulation of advanced techniques in solvers. For example, the reasonably compact transformation of a PR proof into a DRAT proof [HB18] is facilitated by the clause deletion step in DRAT. Several clause addition steps would be invalid if some clauses were not removed first. It is possible to simulate the deletion of clauses with extended resolution and thus its generalizations by making a copy of all the clauses apart from the deleted ones using copies (renaming) of the variables [KRPH18]. However, the number of copied clauses is linear in the size of the formula and thus typically much more expensive compared to removal.

In the context of proofs of unsatisfiability, one only needs to check whether clause addition steps are valid. Clause deletion steps can be performed without a check, because the validity of the unsatisfiability result only depends on clause addition. However, when checking also the validity of clause deletion steps, i.e., whether clause deletion preserves unsatisfiability, one can construct proofs of satisfiability by showing the equivalence of the input formula with the empty formula. This is not really useful in propositional logic as a satisfying assignment can be used as a certificate for satisfiability. Yet, this approach can be useful for richer logics, such as Quantified Boolean Formulas [HSB14], see Chapter 31.

15.3. Proof Search

The leading paradigm to solve satisfiability problems is the conflict-driven clause learning (CDCL) approach, described in Chapter 4. In short, CDCL adds lemmas, typically referred to as conflict clauses, to a given input formula until either it finds a satisfying assignment or it is able to learn (i.e., deduce) the empty clause (prove unsatisfiability).

CDCL solvers typically use a range of preprocessing techniques, see Chapter 9 for details. Examples of such techniques are bounded variable elimination (also known as Davis-Putnam resolution) [DP60, EB05], blocked clause elimination [JBH12], subsumption, and hyper binary resolution [BW03]. Preprocessing techniques are frequently crucial to solve large formulas efficiently. These techniques can also be used during the solving phase, which is known as *inprocessing* [JHB12]. 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.

Some CDCL solvers, however, use preprocessing techniques that are hard to represent using resolution proofs. Examples of such techniques are bounded variable addition [MHB12], blocked clause addition [JHB12], Gaussian elimination, cardinality resolution [CCT87], and symmetry breaking [ASM06]. These techniques cannot be polynomially simulated using the resolution rule: For example, certain formulas based on expander graphs are hard for resolution [Urq87], meaning that they admit only resolution proofs of exponential size, while Gaussian elimination can solve them efficiently. Similarly, formulas arising from the pigeon hole principle are hard for resolution [Hak85], but they can be solved efficiently using either cardinality resolution or symmetry breaking. Consequently, solvers that can efficiently solve such problems cannot produce resolution proofs that are linear in size of the solving time as all resolution proofs are at least exponential in size.

Techniques such as Gaussian elimination, cardinality resolution, and symmetry breaking, can be simulated polynomially using extended resolution and its generalizations. However, it is not known how to simulate these techniques efficiently or elegantly using extended resolution. One method to translate Gaussian elimination into ER proofs is to convert the Gaussian elimination steps into BDDs and afterwards translate the BDDs to a extended resolution proof [SB06].

An alternative approach to solve satisfiability problems is the look-ahead approach, described in Chapter 5. Look-ahead 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 look-ahead techniques. Although it is possible to extract unsatisfiability proofs from look-ahead solvers, this hardly happens in practice. However, look-ahead SAT solvers can also be used to partition a problem into many subproblems, which is known as the *cube-and-conquer* approach [HKWB11]. The subproblems are solved using CDCL. The proofs produced by the CDCL solvers can be merged into a proof of the original problem. This method was used to produce proofs of the Erdős Discrepancy Theorem [KL14], the Pythagorean Triples Problem [HKM16], and Keller's Conjecture [BHMN20].

Searching for proofs in the stronger proof systems, such as PR, is more complicated and costly as determining whether a clause is a PR clause with respect to a formula is an NP-complete problem. One approach to computing PR proofs is *satisfaction-driven clause learning* (SDCL) [HKSB17], which generalizes CDCL by performing a more aggressive pruning of the search tree via the addition of PR clauses.

15.4. Proof Formats

Proof formats define how the proofs of a given proof system should be represented syntactically so that they can be validated by automated tools. Formats for unsatisfiability proofs come in two flavors: with or without hints. A handful of formats have been designed for resolution proofs [ZM03, ES03, Bie08]. All these proof formats include hints in the form of details that describe how to validate each proof step.

The hints in resolution proofs refer to the antecedents of an application of the resolution rule. These formats differ in several details, such as whether the input formula is stored in the proof, whether resolution chains are allowed, or whether hints in the proofs must be ordered. We will first discuss the two most widely used formats with hints: `TraceCheck` and `LRAT`. The `stc` [Bie08] tool can be used to validate `TraceCheck` files. Various checkers exist for `LRAT` files, including `LRAT-check` and the formally verified tool `ACL2-check` [HHKW17]. The `LRAT` format also supports stronger proof systems, but we will first focus on how to use it for resolution proofs.

For proofs without hints, there is essentially only one format, which lists the redundant clauses using the same syntax as `DIMACS`. Such proofs can be extended with clause deletion information [HHW13a], and with a generalization of extended resolution [HHW13b]. The format that supports both extensions is `DRAT` [WHH14], which corresponds to the `DRAT` proof system; it is backward compatible with the RUP proof format, which corresponds to the RUP proof system. The `DRAT-trim` [WHH14] tool can efficiently validate proofs without hints in the various proof systems.

15.4.1. Formats with Hints

There exist two popular proof formats that include hints: `TraceCheck` and `LRAT`. For resolution proofs, the syntax of both formats is the same. The `TraceCheck` format is restricted to resolution, while the `LRAT` format also allows proofs in stronger proof systems. An important difference between `TraceCheck` and `LRAT` is that only the former includes the input clauses as part of the proof. In contrast, checkers for `LRAT` proofs also require the formula as input. The other difference between these formats is that `TraceCheck` allows an arbitrary order of the lines and the hints, while for `LRAT` the order of lines and hints is strict. These difference can be explained by historical considerations: `TraceCheck` was proposed in 2006, when it was not possible to efficiently validate proofs without hints and when it was also difficult to compute the right order of hints for some techniques. The `LRAT` format was proposed more than a decade later and was designed to make the format strict to make it easy to develop formally-verified checkers.

The proof checker `tracecheck` can be used to check whether a trace represents a *piecewise regular input resolution proof*, which is also known as a *trivial proof* [BKS04]. A proof is *regular* if variables are resolved at most once along any path in the directed acyclic graph (DAG) formed by the proof. It is an *input resolution* proof 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 node has at least one leaf as child. A *trace* is just a compact representation of general resolution proofs. The `TraceCheck` format is more compact than other resolution formats, because it uses resolution chains, and because the resulting resolvent does not need to be stated explicitly. The parts of the proof which are regular input resolution proofs are called chains in the following discussion. The full trace consists of original clauses and the 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 that learned clauses in a CDCL solver can be derived by regular input resolution [BKS04]. A unique feature of `TraceCheck` is that the chains do not have to be sorted, neither between chains (globally) nor between their input clauses (locally). If possible the checker will sort them automatically. This allows a simplified implementation of the trace generation.

A chains is simply represented by the list of its 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 derivations of learned clauses. Computing the chains can become challenging when more advanced learned clause optimizations are used. Shrinking or minimization of learned clauses [SB09] are examples of these optimizations. The difficult part is to order the antecedents correctly. The solver can leave this task to the trace checker instead of changing the minimization algorithm [Van09].

Moreover, both TraceCheck and LRAT facilitate 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. Checkers for TraceCheck don't validate the clauses introduced by the extension rule, while LRAT checkers validate the definition clauses introduced via the extension rule by treating them as blocked-clause-addition steps.

The syntax of a trace is as follows:

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

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

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, which are represented by integers. 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. In TraceCheck, 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. The LRAT format enforces a strict order which will be described below.

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
```

proof without hints (DRUP)

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

proof with hints (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  7 6 8  0
```

Figure 15.3. An input formula (left) in the classical DIMACS format which is supported by most SAT solvers. A proof without hints for the input formula in DRUP format (middle). In both the DIMACS and DRUP formats, each line ending with a zero represents a clause, and each non-zero element represents a literal. Positive numbers represent positive literals, while negative numbers represent negative literals. For example, `-2 3 0` represents the clause $(\bar{b} \vee c)$. A TraceCheck file (right) is a resolution graph that includes the formula and proof with hints. 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 hints.

Consider, for example, the trace shown in Fig. 15.3 (right), which consists of the six clauses from our example CNF formula E, which we introduced on page 4. The corresponding `DIMACS` file is shown in Fig. 15.3 (left). This proof in `TraceCheck` can easily be converted into `LRAT` by simply removing the first few lines, which represent the input formula. This is possible as the order of the clauses in the `DIMACS` file is the same as the order of the original clauses in the trace (which is not required).

The first derived clause in the trace starts with index **7**. This is the unary clause which consists of the literal `-2` ($\overline{b}$). It is obtained by resolving the original clause **4** against the original clause **5** on variable `1` (a).

A chain for the last derived clause, which is the empty clause $\bot$, can be obtained by resolving the antecedents **7**, **6**, and **8**: first **7** is resolved with **6** to obtain the intermediate resolvent `-3` ($\overline{c}$), which in turn can be resolved with clause **8** to obtain the empty clause $\bot$.

The `LRAT` format enforces a strict ordering of the hints. Checking a proof step starts with the assignment that falsifies all literals in the clause. For example, for line **8** it starts with the assignment that makes variable `3` (c) false. The hints are read from left to right and each hint is required to be a unit clause under the current assignment. That assignment is extended by making the unit literal true. The last hint is an exception: the current assignment needs to falsify it. So when checking line **8**, the first hint points to clause ($\overline{b} \vee c$), which is unit ($\overline{b}$) and $\overline{b}$ is added to the assignment. The second hint points to ($a \vee c$), which again is unit (a) and a is added to the assignment. The final hint points to ($\overline{a} \vee b$), which is indeed falsified by the current assignment.

As discussed above, the order of the lines (clauses) in `TraceCheck` is irrelevant, in contrast to `LRAT`. The checker will sort them automatically using the clause identifiers. The same holds for the hints. So, the last two lines of the example trace can be replaced by:

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

Note that the clauses **7** ($\overline{b}$) and **8** (c) cannot be resolved with each other, because they do not contain complementary literals. In this case, the checker has to reorder the antecedents as in the original example.

The main motivation for having hints in the proof for each learned clause is to speed up proof validation. While checking a learned clause, unit propagation can focus on the list of specified antecedents. It can further ignore all other clauses, particularly those that were already discarded at the point where the solver learned the clause.

In the `TraceCheck` format, it might be convenient 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 ⋆ 7 6 8 0
```

Furthermore, trivial clauses and clauses with multiple occurrences of the same literal cannot be resolved. The list of antecedents is not allowed to contain the same index 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.

One feature in the `LRAT` format that is currently not supported in `TraceCheck` is the deletion of clauses. Clause deletion lines in `LRAT` start with an identifier (typically the same identifier as the previous line), followed by "`d`" and a list of clause indices to be deleted, and they end with a "`0`".

15.4.2. Formats Without Hints

Proof formats without hints are designed to make proof logging easy. Apart from the lack of hints, they also don't use clause indices. The absence of hints and clause indices results in much smaller proofs, but these proofs are more expensive to validate. To validate proofs without hints in reasonable time, it is common to include deletion information. Adding this information is also easy.

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 original clauses, while lemmas will refer to added clauses.

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

There exist a few proof formats for proofs without hints; they have a very similar syntax as `DIMACS` and can all be expressed using the above grammar. The most basic format is `DRUP`, short for delete reverse unit propagation, which combines RUP (reverse unit propagation) additions [Van08] and clause deletions. A `DRUP` proof is a sequence of lemmas that are either added or deleted. Each lemma is a list of positive and negative integers (to express positive and negative literals, respectively) that terminates with a zero as in `DIMACS`. Clause deletion steps are expressed using the prefix `d`. An example is shown in Figure 15.3.

Recall that a clausal proof $P := \{C_1, \ldots, C_m\}$ is a valid RUP proof for a formula F if $C_m = \bot$ and for each C_i, it holds that

$$F \wedge C_1 \wedge \cdots \wedge C_{i-1} \vdash_1 C_i$$

This means that for each $i \in 1, \ldots, m$, unit propagation must derive a conflict on the formula $F \wedge C_1 \wedge \cdots \wedge C_{i-1} \wedge \overline{C}_i$. Consider again the example CNF formula E

from page 4. A RUP proof is shown in Figure 15.3. This proof, $P_E := \{(\bar{b}), (c), \bot\}$, is a valid proof for E, because P_E terminates with $\bot$ and

$$\begin{array}{lll} E & \vdash_1 & (\bar{b}) \\ E \wedge (\bar{b}) & \vdash_1 & (c) \\ E \wedge (\bar{b}) \wedge (c) & \vdash_1 & \bot \end{array}$$

15.4.3. Formats with Witnesses

So far we only considered proof formats that validate techniques which can be simulated using resolution. As mentioned earlier, however, some SAT solvers use techniques—such as blocked clause addition or symmetry breaking—that cannot be simulated using resolution. To validate these techniques, proof formats need to support stronger proof systems such as extended resolution or one of its generalizations.

In these formats, a witness is provided for each proof step in addition to the clause and the optional hints. For most proof systems, the witness consists of a single literal that needs to be part of the clause. To minimize overhead, the convention is to place the witness as the first literal in the clause. For the clauses added by the extension rule this means to put the literals referring to the new variable first.

Resolution proofs, as the name suggests, can only be used to check techniques that can be expressed using resolution steps. 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 `DRAT` proof format [WHH14], which is syntactically the same as the `DRUP` format, supports expressing techniques based on extended resolution and its generalizations. The difference between the `DRUP` and `DRAT` format is in the redundancy check that is computed in the checker for proofs in that format. A checker for `DRUP` proofs validates whether each lemma is a RUP clause, while a checker of `DRAT` proofs checks whether each lemma is a RAT clause [HHW13b].

Example 15.4. Consider the following CNF formula

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

On the left of Fig. 15.4, G is shown in the `DIMACS` format, using the conventional mapping from the alphabet to numbers, where a is mapped to `1`, $\bar{a}$ is mapped to `-1`, b to `2`, $\bar{b}$ to $-$`2`, and so on. In the middle of Fig. 15.4, a `DRAT` proof for G is shown. The proof uses the earlier-mentioned technique of bounded variable addition [MHB12], which cannot be expressed using resolution steps. Bounded variable addition 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. 15.4 shows how to express bounded variable addition in the `DRAT` format: First add the new binary clauses using the pivot (witness) literal on the first position, followed by deleting the old ones. After the replacement, the proof is short $\{(f), \bot\}$.

Validating proofs based on extended resolution or one of its generalizations requires reasoning about absence. For example, the addition of definition clauses via the extension rule is only valid when the new variable is indeed new and does not occur in the formula. We need to check all clauses that are reduced but not satisfied by the witness. In most cases, the witness is a single literal, so we need to check only the clauses in which the complement of the literal occurs.

Consider the following example of a proof that again uses bounded variable addition. Fig. 15.4 shows the example formula G in `DIMACS` as well as a `DRAT` proof and an `LRAT` proof. The clauses $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$ are trivially redundant with respect to G using witness f, because G does not contain any clause with literal $\overline{f}$. Making f true would satisfy these three clauses. However, $(\overline{f} \vee d)$ and $(\overline{f} \vee e)$ are not trivially redundant with respect to G using witness $\overline{f}$ after the addition of $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$. For example, the redundancy of $(\overline{f} \vee d)$ depends on the presence of $(a \vee d)$, $(b \vee d)$, and $(c \vee d)$ as the RAT check would fail without these clauses.

Let's recall the definition of RAT clauses from Section 15.2.4.1: A clause C is a RAT on l with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$, it holds that $F \vdash_1 C \diamond D$. In the proof, $(\overline{f} \vee d)$ is claimed to be RAT on $\overline{f}$. Let G' denote the formula when checking $(\overline{f} \vee d)$: $G' := G \wedge (f \vee a) \wedge (f \vee b) \wedge (f \vee c)$. We can confirm the RAT claim by applying the following three checks:

$$\begin{aligned} G' &\vdash_1 (a \vee d) \qquad \text{using } D = (f \vee a) \\ G' &\vdash_1 (b \vee d) \qquad \text{using } D = (f \vee b) \\ G' &\vdash_1 (c \vee d) \qquad \text{using } D = (f \vee c) \end{aligned}$$

The first check succeeds because $(a \vee d) \in G'$ is reduced to the empty clause, whereas the second and third checks succeed, because $(b \vee d) \in G'$ and $(c \vee d) \in G'$ are reduced to the empty clause. In the `LRAT` proof format this is expressed as follows: For each clause that is reduced but not satisfied by the witness (the D clauses), in this case $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$, the negated clause index is listed: `-9`, `-10`, and `-11`, respectively. After the negated clause index, the clause indices of the unit clauses are listed and finally the clause that is reduced to the empty clause. For $(f \vee a)$, clause $(a \vee d)$ is reduced to the empty clause, which has clause index `2`.

15.4.4. Binary Formats and Proof Compression

It is common practice to store proofs on disk, and we discussed various formats for this purpose. 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. In this subsection, we discuss light-weight options to compress proofs [HB16]. The variable-byte encoding discussed below is supported by most SAT solvers.

15.4.4.1. Byte Encoding

The ASCII encoding of a proof line without hints in Figure 15.3 is easy to read, but rather verbose. For example, consider the literal `-123456789`, which requires

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 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
```

LRAT proof

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

Figure 15.4. Example formula G in the classical `DIMACS` format (left). A proof without hints for the input formula in `DRAT` format (middle). The corresponding proof with hints as a `LRAT` file is shown on the right. Notice that the formula is not included. Each line begins with a clause identifier (bold), then contains the literals of the lemma, and ends with a list of hints (bold).

11 bytes to express (one for each ASCII character and one for the separating space). This literal can also be represented by a signed integer (4 bytes). If all literals in a proof can be expressed using signed integers, only 4 bytes are required to encode each literal. Such an encoding also facilitates omitting a byte to express the separation of literals. Consequently, one can easily compress an ASCII proof with a factor of roughly 2.5 by using a binary encoding of literals.

In case the length of literals in the ASCII representation differs a lot, it may not be efficient to allocate a fixed number of bytes to express each literal. Alternatively, the *variable-byte encoding* [WMB99] can be applied, which uses the most significant bit of each byte to denote whether a byte is the last byte required to express a given literal. The variable-byte encoding can express the literal `1234` (`10011010010` in binary notation) using only two bytes: `11010010 00001001`. (in little-endian ordering, e.g., least-significant byte first).

15.4.4.2. Sorting Literals

The order of literals in a clausal proof has no influence on the validity or the size of a proof. However, the order of literals can influence the cost of validating a proof, as it influences unit propagation and in turn determines which clauses will be marked in backwards checking (the default validation algorithm used in checkers of proofs without hints), see Section 15.6.2 for details. The order of literals in the proof produced by the SAT solver is typically not better or worse than any permutation for validation purposes. However, experience shows that this is often not the case for SAT solving: the given order of literals in an encoding results in stronger solver performance compared to any permutation.

Sorting literals before compression has advantages in both light-weight and heavy compression. In light-weight compression, one can use delta encoding: store the difference between two successive literals. Clauses in a proof are typically long (dozens of literals) [HHW13a], resulting in a small difference between two successive sorted literals. Delta encoding is particularly useful in combination with variable-byte encoding.

In heavy compression, off-the-shelf zip tools could exploit the sorted order of literals. Many clauses in proofs have multiple literals in common. SAT solvers tend to emit literals in a random order. This makes it hard for compression tools to detect overlapping literals between clauses. Sorting literals potentially increases the observability of overlap which in turn could increase the quality of the compression algorithm.

Table 15.1. Eight encodings of an example `DRAT` proof line. The first two encodings are shown as ASCII text using decimal numbers, while the last six are shown as hexadecimals using the `MiniSAT` encoding of literals. The prefix s denotes sorted, while the prefix ds denotes delta encoding after sorted. 4byte denotes that 4 bytes are used to represent each literal, while vbyte denotes that variable-byte encoding is used.

encoding	**example** (prefix pivot $\text{lit}_1 \ldots \text{lit}_{k-1}$ end)	**#bytes**
ascii	`d 6278 -3425 -42311 9173 22754 0\n`	33
sascii	`d 6278 -3425 9173 22754 -42311 0\n`	33
4byte	`64 0c 31 00 00 c3 1a 00 00 8f 4a 01 00 aa 47 00 00 c4 b1 00 00 00 00 00 00`	25
s4byte	`64 0c 31 00 00 c3 1a 00 00 aa 47 00 00 c4 b1 00 00 8f 4a 01 00 00 00 00 00`	25
ds4byte	`64 0c 31 00 00 c3 1a 00 00 e8 2c 00 00 1a 6a 00 00 cb 98 00 00 00 00 00 00`	25
vbyte	`64 8c 62 c3 35 8f 95 05 aa 8f 01 c4 e3 02 00`	15
svbyte	`64 8c 62 c3 35 aa 8f 01 c4 e3 02 8f 95 05 00`	15
dsvbyte	`64 8c 62 c3 35 e8 59 9a d4 01 cb b1 02 00`	14

15.4.4.3. Literal Encoding

In most SAT solvers, literals are mapped to natural numbers. The default mapping function $map(l)$, introduced in MiniSAT [ES03] and also used in the AIGER format [Bie07], converts signed `DIMACS` literals into unsigned integer numbers as follows:

$$map(l) = \begin{cases} 2l + 1 \text{ if } l > 0 \\ -2l \quad\;\; \text{otherwise} \end{cases}$$

Table 15.1 shows a proof line in the `DRAT` format and in several binary encodings. For all non-ASCII encodings, we will use $map(l)$ to represent literals. Notice that the first literal in the example is not sorted, because the proof checker needs to know the witness / pivot literal (which is the first literal in each clause). The remaining literals are sorted based on their $map(l)$ value.

15.5. Proof Production in Practical SAT Solving

Proof logging of unsatisfiability results from practical SAT solvers started in 2003, with both resolution proofs *with* hints [ZM03] and resolution proofs *without* hints [GN03]. Proofs with hints are typically hard to produce and tend to require lots of overhead in memory, which in turn slows down a SAT solver. Emitting proofs without hints is easy and requires hardly any overhead in memory. We will first describe how to produce proofs without hints, followed by how to produce proofs without hints, and finally describe parallel proof production.

15.5.1. Proofs Without Hints

For all proof formats without hints, SAT solvers simply emit proofs directly to disk. Consequently, there is no memory overhead. In contrast to proofs with hints, it is easy to emit proofs. For the most simple format of proofs without hints, RUP, one only needs to extend the proof with all clauses learned by the CDCL solver. This can be implemented by a handful lines of code. Below we discuss how to produce proofs without hints that support deletion information and techniques based on generalizations of extended resolution.

Proofs without hints need deletion information for efficient validation, see Section 15.6.2 for details. Adding deletion information to a proof without hints is straightforward. 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. Recall that clauses in proofs without hints have no clause index. Hence all literals in the deleted clause are listed. The checker needs to find the deleted clause in the clause database. This can be done efficiently using a hash function. 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 easily be expressed in proofs without hints. Similar to techniques based on resolution, one simply adds the lemmas to the proof for most techniques. The RAT and `DRAT` formats only require that the witness / pivot literal is the first literal of the lemma in the proof. However, as discussed in Section 15.3, there exist some techniques for which it is not known whether they can be expressed elegantly in the `DRAT` format. As mentioned already, especially Gaussian elimination, cardinality resolution, and symmetry breaking are hard to express in the current formats. Finally, for proof systems that require witnesses, the witness is printed directly after the clause and before the zero that marks the end of the line.

15.5.2. Proofs With Hints

Early work on proofs with hints [ZM03] mostly showed that proofs can be generated in principle, but ignored dealing with practical aspects such as keeping proofs compact and supporting all techniques. The first proof format that took these aspects into account was TraceCheck, which we discussed already and which was first supported by `PicoSAT` [Bie08]. The addition of proof logging was motivated by making testing and debugging more efficient. In combination with file-based delta-debugging [BLB10], proof trace generation allows to reduce discrepancies much more than without proof tracing.

The original idea was to use resolution traces. However, for some time it was unclear how to extract resolution proofs during a-posteriori clause minimization [SB09]. This was the reason for using a trace format instead: clause minimization is obviously compatible with reverse unit propatation, since required clause antecedents can easily be obtained. However, determining the right order of the hints for a resolution proof is hard to generate directly and probably needs a RUP style algorithm anyhow. It was shown how clause minimization can be integrated in `MiniSAT` using a depth-first search for the first-unique-implication-point clause [Van09], which at the same time 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 [HS09] as well. The solution is also not as easy to add to existing solvers as tracing added and deleted clauses.

As already discussed above, memory consumption of proofs stored in memory (or disk) can become a bottleneck. One way to reduce space requirements is to delete garbage clauses that are not used anymore. This garbage collection was implemented with saturating reference counters in `PicoSAT` [Bie08]. It has been shown that full reference counting can result in substantial reductions [ANORC08]. 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.

15.5.3. Parallel Proof Production

A framework based on the cube-and-conquer paradigm has been used to deal with hard combinatorial problems that require thousands of hours of computation to solve [HKM16]. In short, the framework consists of five phases: *encode*, *re-encode*, *split*, *solve*, and *validate*. The focus of the encode phase is to make sure that encoding the problem into SAT is valid. The re-encode phase reformulates the problem to reduce the computation costs of the later phases. The split phase partitions the transformed formula into many (typically millions of) subproblems. The subproblems are solved in the solve phase. The validation phase checks whether the proofs emitted in the prior phases are a valid refutation for the original formula. Figure 15.5 shows an illustration of the framework.

In recent years, three long-standing open problem in mathematics have been solved using this framework: the Pythagorean Triples Problem (PTP) [HKM16], Schur number five (S5) [Heu18], and Keller's Conjecture [BHMN20]. The encoding of PTP and S5 is natural and can be achieved with about a dozen lines of code. The encoding of Keller is more involved and also included several symmetry-breaking clauses that arise from symmetries of the problem, but these are not symmetries in the encoding. The re-encoding phase of all three problems consists of adding symmetry-breaking predicates. Additionally, about 30% of the clauses of PTP are redundant and they have been removed using blocked clause elimination.

All three problems used a two-layered splitting approach. For PTN and S5, the re-encoded problem was split into a few million subproblems using the look-ahead solver march_cu. This can be done in less than an hour of a single CPU. For Keller, the problem was manually split into many thousands of subproblems. All these subproblems were split into even smaller problems using march_cu. This second layer of splitting was performed in parallel. It resulted in billions of subproblem for PTN and S5 and millions of subproblems for Keller. The subproblems were solved using a CDCL solver. Most subproblems are very easy and almost any CDCL solver could be used for this purpose.

The proofs of these problems consists of three parts: The *re-encoding proof*, the *implication proof*, and the *tautology proof*. These proof part shows the satisfiability equivalence of three formulas: 1) the encoded formula F, 2) the re-encoded formula R, and 3) the negation of all of cubes that are part of the first layer

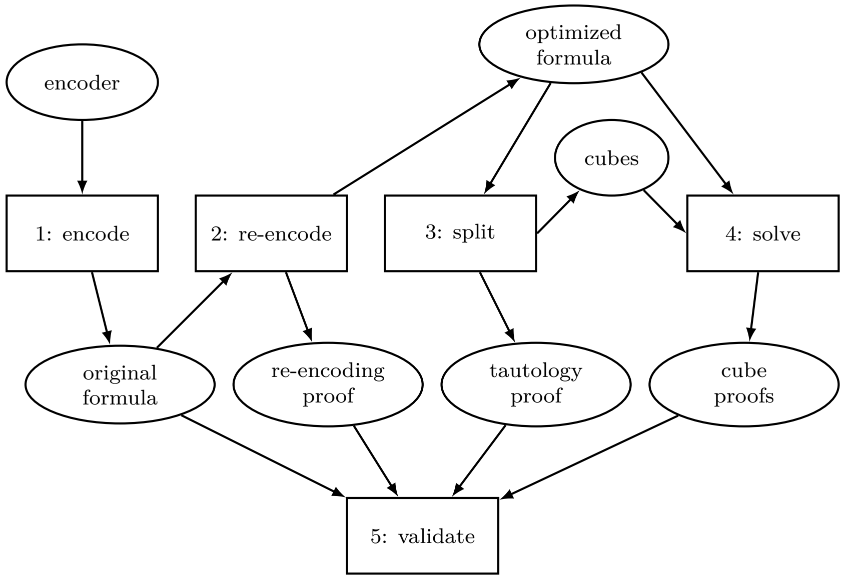


Figure 15.5. Illustration of the framework to solve hard combinatorial problems. The phases are shown in the rectangle boxes, while the input and output files for these phases are shown in oval boxes.

split T. The latter is called T as the cubes need to cover the entire search space, which means that it needs to be a tautology. Together the proof parts show the following relationship:

$$\overbrace{F \models R}^{\text{re-encoding proof}} \models \overbrace{T \models \bot}^{\text{tautology proof}} \; .$$
$$\underbrace{R \models T}_{\text{implication proof}}$$

The re-encoding proof expresses the correctness of the re-encoding phase, which consisted of symmetry breaking and the removal of redundant clauses. The re-encoding cannot be compactly expressed using resolution, so a proof of extended resolution or its generalizations is required. For PTP, S5, and Keller, the re-encoding proof used `DRAT`.

By far the largest part of the proof (typically over 99%) is the implication proof. It shows that R is unsatisfiable by proving that every clause $C \in T$ is logically implied by R. For each clause $C \in T$ a separate proof piece is produced and these pieces are glued together at the end, which can be done in any arbitrary order. The lack of hints in `DRAT` proofs make the glueing easy: one can simply concatenate the pieces.

The generation of the final part, the tautology proof, is easy as well as the cubes from the first layer split form a binary tree of assignments by construction. Let m be the number of cubes in our partition, then the tautology proof is a resolution proof consisting of $m-1$ resolution steps.

15.6. Proof Validation and Processing

Although proofs with hints are harder to produce than proofs without hints, they are in principle easy to check and actually needed for some applications, like generating interpolants [McM03]. However, the large size of resolution proofs provides challenges, particularly with respect to memory usage. See [Van12] for a discussion on checking large resolution proofs. The presence of hints makes it easy to implement a checker, especially if the proof format does not allow any ambiguity, such as the `LRAT` format. This allowed for the implementation of certified (formally verified) checkers, such as `ACL2-check` [HHKW17].

Proofs without hints are smaller, but validating proofs without hints is more complicated and more costly. Practically all top-tier solvers support the production of proofs without hints, as this is easy to implement, while hardly any of them supports the production of proofs with hints. To obtain the best of both worlds—easy proof production and certified results—there exist tools that turn a proof without hints into a proof with hints. One such tool is `DRAT-trim` [WHH14], which can turn a `DRAT` proof into an `LRAT` proof. `DRAT-trim` also allows the optimization of proofs, for instance, by removing redundant lemmas. The corresponding tool chain for certifying proofs is shown in Figure 15.6: A SAT solver produces a proof *without* hints, which is turned into an optimized proof *with* hints that is finally validated by a certified checker.

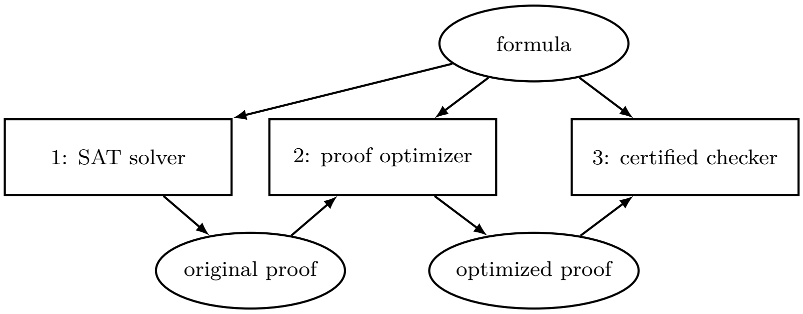


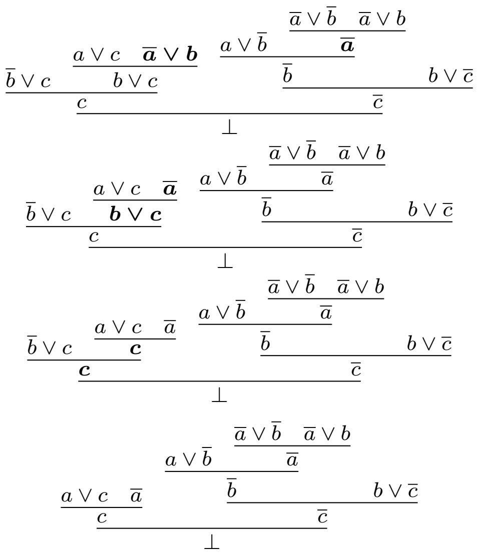
Figure 15.6. Tool chain of certifying proofs.

15.6.1. Proofs With Hints

Resolution proofs can be checked in deterministic logspace [Van08], a very low complexity class. The tool `stc` can efficiently check proofs in the `TraceCheck` format. More details about the format and its use are discussed in Section 15.4.1.

Apart from validation, there exists a vast body of work on compression techniques for resolution proofs [Amj07, BIFH$^+$08, Cot10, FMP11, RBST14, Sin07] that go beyond the compression techniques we discussed earlier. One such techniques is *RecycleUnits* [BIFH$^+$08]: unit clauses in the proof are used to replace some clauses in the proof that are subsumed by the units. The replacement typically makes the proof invalid (i.e., some resolvents are no longer the result of resolving the antecedents). However, the proof can be fixed with a few simple steps. Figure 15.7 illustrates the RecycleUnits procedure on a derivation of the example CNF formula E from page 4. In the example, notice that replacing a clause by a unit may strengthen the resolvent.

Two other proof compression techniques are *LowerUnits* and *RecyclePivots*. LowerUnits uses the observation that one needs to resolve over a unit clause only a single time. In case a unit clause occurs multiple times in a proof, it is lowered to ensure that it occurs only once. RecyclePivots reduces the *irregularity* in resolution proofs. A proof is irregular if it contains a path on which resolution is performed on the same pivot. Removing all irregularity in a proof may result in an exponential blow-up of the proof [Goe90]. Hence, techniques such as RecyclePivots need to be restricted in the context of proof compression. The tool `Skeptic` [BFP14], which includes most of the compression techniques, can be used to remove redundancy from resolution proofs.



Unit clause $\overline{a}$ can be recycled by replacing the non-antecedent $\overline{a} \vee b$

The resolvent changes after replacing $\overline{a} \vee b$ by $\overline{a}$. This is fixed by replacing $b \vee c$ with the actual resolvent c.

After replacing $b \vee c$ by c, a similar problem occurs. Now the antecedent c and the "resolvent" are the same.

Removal of the redundant resolution step results in a correct and more compact derivation.

Figure 15.7. An example of the proof compression technique RecycleUnits.

15.6.2. Proofs Without Hints

Proofs without hints are checked using unit propagation. The actual check for each step depends on the proof system. Recall that a RUP proof $\{C_1, \dots, C_m\}$ is valid for formula F, if $C_m = \bot$ and for $i \in \{1, \dots, m\}$, it holds that

$$F \wedge C_1 \wedge \cdots \wedge C_{i-1} \vdash_1 C_i.$$

The check for stronger proof systems is more elaborate and thus more expensive. See Section 15.2 for details. The most simple (but very costly) method to validate proofs without hints checks for every $i \in \{1, \dots, m\}$ whether or not the redundancy property of the corresponding proof system holds.

One can reduce the cost of validating proofs without hints by checking them *backwards* [GN03], thereby *marking* clauses that are used for the derivation of other clauses that occur later in the proof: Initially, only the final clause $C_m = \bot$ is marked. Now we loop over the lemmas in backwards order, i.e., $C_m, \ldots, C_1$. Before validating a lemma, we first check whether it is marked. If a lemma is not marked, it can be skipped, which reduces the computational cost. If a lemma is marked, we check whether the clause satisfies the above redundancy criterion. 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, this technique allows to skip over half of the lemmas during validation.

The main challenge regarding the validation of proofs without hints is efficiency: Validating a proof without hints 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. For most other logics, including first order logic, checking a proof is typically cheaper than finding the proof in the first place. There are two main reasons why checking unsatisfiability proofs 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 than in the solver. This was the main motivation why the proof formats `DRUP` and `DRAT` have been developed. These formats support the inclusion of clause-deletion information, thereby making the unit propagation costs between the solver and checker similar.

Second, SAT solvers *reuse* propagations between conflicts whereas proof checkers do not reuse propagations. Consider two consecutive lemmas C_i and C_{i+1}, produced by a SAT solver. In the most extreme (but not unusual) case, the branch that resulted in C_i and C_{i+1} differs only in a single decision (out of many decisions). Hence most propagations are reused by the solver. At the same time, the lemmas might have no overlapping literals, meaning that $C_i \cap C_{i+1} = \emptyset$. Consequently, the checker would not be able to reuse any propagations. In case $C_i \cap C_{i+1}$ is non-empty, the checker could potentially reuse propagations, although no checker implementation for proofs without hints exploits this.

While checking a proof without hints, one can easily produce an unsatisfiable core and a proof with hints. 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 unmarked and thus not part of the unsatisfiable core. The proof with hints contains for each marked lemma all the clauses that were required during its validation as hints.

The hints that are generated while checking a proof without hints might differ significantly from the hints that would have been produced by the SAT solver. For example, the resolution proof (with hints) for the example formula E might be equal to the resolution graph shown in Fig. 15.1. On the other hand, the resolution proof produced by checking the proofs without hints might be equal to Fig. 15.8. Notice that the resolution graph of Fig. 15.8 (right) does not use all original clauses. Clause $(\bar{b} \vee c)$ is redundant and not part of the core of E.

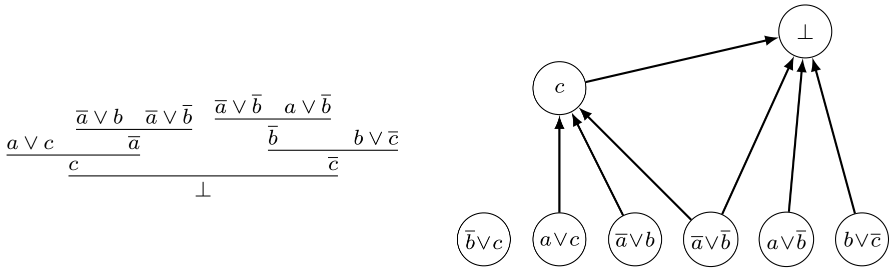


Figure 15.8. A resolution derivation (left) and a resolution graph (right) for the example formula E produced by checking a proof without hints.

15.7. Proof Applications

Proofs of unsatisfiability have been used to validate the results of SAT competitions.[2] Initially, during the SAT competitions 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 the SAT competition 2013 were `TraceCheck` and `DRUP`, but all solvers participating in these tracks opted for the `DRUP` format. For the SAT competition 2014, the only supported format was `DRAT`, which is backwards compatible with `DRUP`. Since 2016, the SAT competition requires proof logging to participate in the main track.

As already mentioned, 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 that are not implied by reverse unit propagation.

Testing with forward proof checking is particularly effective in combination with fuzzing (generating easy formulas) and delta-debugging [BB09b] (shrinking a formula that triggers a bug). Otherwise failures produced by unsound 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 [ABS13] of the incremental API of a SAT solver is in our experience at least as effective as file-based fuzzing and delta-debugging. The online proof checking capabilities to `Lingeling` [Bie14] allow the combination of these two methods (model-based testing and proof checking).

Another important aspect of proof tracing is that it allows to generate a clausal (or variable) core (i.e., an unsatisfiable subset). These cores can be used in many applications, including MUS extraction [NRS13], MaxSAT [MHL+13], diagnosis [SKK03, NBE12], and abstraction refinement in model checking [EMA10] or SMT [ALS06, BB09a]. 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.

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

Finally, extraction of interpolants is an important usage of resolution proofs, particularly in the context of interpolation-based model checking [McM03]. Since resolution proofs are large and not easy to obtain, there have been several recent attempts to avoid proofs and obtain interpolants directly, see for instance [VRN13]. Interpolation-based model checking became the state-of-the-art until the invention of IC3 [Bra11]. 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` [ES03] (see also [LB13]) instead of proof-based techniques.

15.8. 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.

Proofs of unsatisfiability come in two types: with and without hints. Proofs without hints are useful for various applications, but they are costly and complicated to produce. Therefore few SAT solvers support proof logging with hints. Proof logging without hints is easy and therefore most state-of-the-art solvers support it. However, validating proofs without hints is costly, although recent advances significantly improved performance of checkers.

There are several challenges regarding unsatisfiability proofs. How can one store proofs with hints using much less space on disk and produce them using much less memory overhead? Can the cost of validating proofs with hints be be reduced further? 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 in unsatisfiability proofs.

Acknowledgements

The author thanks Benjamin Kiesl, Emre Yolcu, and David Narv'aez for their useful comments on earlier drafts. This chapter reuses parts from “Proofs for Satisfiability Problem”, the first chapter of “All about Proofs, Proofs for all” [HB15].

References

[ABS13] 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.

[AFG$^+$11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 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.

[ALS06] 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.

[Amj07] Hasan Amjad. Compressing propositional refutations. *Electr. Notes Theor. Comput. Sci.*, 185:3–15, 2007.

[ANORC08] 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.

[AS13] Gilles Audemard and Laurent Simon. Glucose 2.3 in the SAT 2013 Competition. In Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 42–43, 2013.

[ASM06] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006.

[BB09a] Robert Brummayer and Armin Biere. Effective bit-width and under-approximation. In *Proc. EUROCAST'09*, volume 5717 of *LNCS*, pages 304–311, 2009.

[BB09b] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *International Workshop on Satisfiability Modulo Theories (SMT)*, pages 1–5. ACM, 2009.

[BFP14] Joseph Boudou, Andreas Fellner, and Bruno Woltzenlogel Paleo. Skeptik: A proof compression system. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 374–380. Springer, 2014.

[BHMN20] Joshua Brakensiek, Marijn J. H. Heule, John Mackey, and David Narváez. The resolution of keller's conjecture. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 48–65, Cham, 2020. Springer International Publishing.

[BHMS14] Anton Belov, Marijn J. H. Heule, and Joao P. Marques-Silva. Mus extraction using clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing — SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 48–57. Springer International Publishing, 2014.

[Bie07] Armin Biere. The AIGER and-inverter graph (AIG) format, version 20070427, 2007.

[Bie08] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[Bie13] Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT Competition 2013. In Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2013*, volume B-

2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 51–52, 2013.

[Bie14] Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors, *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.

[BIFH+08] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In Hana Chockler and Alan J. Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2008.

[BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *JAIR*, 22:319–351, 2004.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 6175 of *LNCS*, pages 44–57. Springer, 2010.

[Bra11] 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.

[BS10] 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.

[BT19] Sam Buss and Neil Thapen. DRAT proofs, propagation redundancy, and extended resolution. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2019.

[BW03] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In Giunchiglia and Tacchella [GT04], pages 341–355.

[CCT87] W. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25 – 38, 1987.

[CFHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.

[Coo76] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, October 1976.

[Cot10] Scott Cotton. Two techniques for minimizing resolution proofs. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing — SAT 2010*, volume 6175 of *Lecture Notes* *in Computer Science*, pages 306–312. Springer, 2010.

[CR79] 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.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[EB05] 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.

[EMA10] Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In Bloem and Sharygina [BS10], pages 181–188.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Giunchiglia and Tacchella [GT04], pages 502–518.

[FMP11] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2011.

[GN03] 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.

[Goe90] Andreas Goerdt. Comparing the complexity of regular and unrestricted resolution. In Heinz Marburger, editor, *GWAI-90 14th German Workshop on Artificial Intelligence*, volume 251 of *Informatik-Fachberichte*, pages 181–185. Springer Berlin Heidelberg, 1990.

[GT04] 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.

[GV14] Arie Gurfinkel and Yakir Vizel. Druping for interpolants. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 19:99–19:106, Austin, TX, 2014. FMCAD Inc.

[Hak85] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[HB15] Marijn J.H. Heule and Armin Biere. Proofs for satisfiability problems. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for all*, chapter 1. College Publications, 2015.

[HB16] Marijn J. H. Heule and Armin Biere. Clausal proof compression. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL-2015. 11th International Workshop on the Implementation of Logics*, volume 40 of *EPiC Series in Computing*, pages 21–26. EasyChair, 2016.

[HB18] Marijn J. H. Heule and Armin Biere. What a difference a variable makes. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 75–92, Cham, 2018. Springer International Publishing.

[Heu18] Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pages 6598–6606. AAAI Press, 2018.

[HHJW15] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 591–606, Cham, 2015. Springer International Publishing.

[HHKW17] Marijn J. H. Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 269–284, Cham, 2017. Springer International Publishing.

[HHW13a] 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.

[HHW13b] 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.

[HKB19] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *Journal of Automated Reasoning*, 64:533 – 554, 2019.

[HKM16] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.

[HKSB17] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 179–194, Cham, 2017. Springer International Publishing.

[HKWB11] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, volume 7261 of *LNCS*, pages 50–65. Springer, 2011.

[HS09] HyoJung Han and Fabio Somenzi. On-the-fly clause improvement. In Kullmann [Kul09], pages 209–222.

[HSB14] Marijn J. H. Heule, Martina Seidl, and Armin Biere. A unified proof system for qbf preprocessing. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 91–106, Cham, 2014. Springer International Publishing.

[JBH12] Matti Järvisalo, Armin Biere, and Marijn Heule. Simulating circuit-level simplifications on CNF. *J. Autom. Reasoning*, 49(4):583–619, 2012.

[JHB12] 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.

[KHB19] Benjamin Kiesl, Marijn J. H. Heule, and Armin Biere. Truth assignments as conditional autarkies. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*, pages 48–64, Cham, 2019. Springer International Publishing.

[KL14] Boris Konev and Alexei Lisitsa. A SAT attack on the erdős discrepancy conjecture. pages 219–226, 2014.

[KRPH18] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, pages 516–531, Cham, 2018. Springer International Publishing.

[KSTB16] Benjamin Kiesl, Martina Seidl, Hans Tompits, and Armin Biere. Super-blocked clauses. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning*, pages 45–61, Cham, 2016. Springer International Publishing.

[Kul99] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999.

[Kul09] 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.

[Lam20] Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64:513–532, 2020.

[LB13] 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.

[McM03] 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.

[MHB12] 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.

[MHL$^+$13] 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.

[Nad10] Alexander Nadel. Boosting minimal unsatisfiable core extraction.

In Bloem and Sharygina [BS10], pages 221–229.

[NBE12] Alexander Nöhrer, Armin Biere, and Alexander Egyed. Managing SAT inconsistencies with HUMUS. In *Proc. VaMoS'12*, pages 83–91. ACM, 2012.

[NRS13] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *FMCAD*, pages 197–200. IEEE, 2013.

[RBST14] Simone Fulvio Rollini, Roberto Bruttomesso, Natasha Sharygina, and Aliaksei Tsitovich. Resolution proof transformation for compression and interpolation. *Formal Methods in System Design*, 45(1):1–41, 2014.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[SB06] 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.

[SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Kullmann [Kul09], pages 237–243.

[Sin07] Carsten Sinz. Compressing propositional proofs by common subproof extraction. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *EUROCAST*, volume 4739 of *Lecture Notes in Computer Science*, pages 547–555. Springer, 2007.

[SKK03] 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.

[Soo13] Mate Soos. Strangenight. In A. Belov, M. Heule, and M. Järvisalo, editors, *Proceedings of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 89–90, 2013.

[Tse83] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2*, pages 466–483. Springer, 1983.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.

[Van08] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.

[Van09] Allen Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In Kullmann [Kul09], pages 141–146.

[Van12] 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.

[VRN13] 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 Com-*

puter Science, pages 330–346. Springer, 2013.

[WA09] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26–40, 2009.

[Web06] Tjark Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In *International Workshop on the Implementation of Logics (IWIL)*, volume 212, pages 44–62, 2006.

[WHH13] 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.

[WHH14] 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.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers Inc., San Francisco, CA, 1999.

[ZM03] 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.