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 9

Preprocessing in SAT Solving

Armin Biere, Matti Järvisalo, and Benjamin Kiesl

Preprocessing has become a key component of the Boolean satisfiability (SAT) solving workflow. In practice, preprocessing is situated between the encoding phase and the solving phase, with the aim of decreasing the total solving time by applying efficient simplification techniques on SAT instances to speed up the search subsequently performed by a SAT solver. In this chapter, we overview key preprocessing techniques proposed in the literature. While the main focus is on techniques applicable to formulas in conjunctive normal form (CNF), we also selectively cover main ideas for preprocessing structural and higher-level SAT instance representations.

9.1. Introduction

One of the main reasons for the success of SAT solving is the ability of modern conflict-driven clause-learning solvers (described in Chapter 4 on CDCL) to search over large CNF formulas with millions of variables and clauses. Nevertheless, real-world applications such as model checking (Chapter 18) and software verification (Chapter 20) often yield formulas that are still too large for state-of-the-art solvers, which in turn can limit the applicability of SAT solving as the solution method of choice. The immense size of formulas is often caused by a combination of the complex nature of the problems themselves, the propositional encodings available, and the automated encoding techniques that generate the formulas in practice.

Automated encoding techniques in particular can deteriorate solver performance by introducing redundancies in the form of unnecessary clauses and variables. Moreover, one and the same problem can often be encoded in various different ways, meaning that SAT solvers are at the mercy of the user in terms of how a problem instance is actually encoded. This calls for *preprocessing*, that is, the application of automated techniques that simplify formulas before passing them to solvers in order to improve the efficiency of the whole solving pipeline. Preprocessing not only improves efficiency in general, it also takes the burden of crafting complex problem encodings from the user, thus allowing non-expert users to leverage the power of modern SAT solvers.

Preprocessing has within recent years become a key component of the SAT solving workflow. A landmark in the history of preprocessing is the technique of *bounded variable elimination* (BVE) [EB05], as implemented in the SatELite preprocessor [EB05] and later integrated in version 2.0 [ES06] of MiniSat [ES03].[1] In 2005 and 2006 it contributed to the largest improvement of solver performance witnessed in the history of the SAT competitions. Even today, bounded variable elimination is still arguably the most important practical preprocessing technique.

In the SAT solving workflow, preprocessing is situated between the encoding phase, during which a user or a dedicated tool encodes a problem as a formula of propositional logic, and the solving phase, during which a SAT solver tries to determine the satisfiability of a formula. The distinction between these phases, however, is not straightforward: on the one hand, preprocessing can be viewed as an automated re-encoding of a formula, which would make it part of the encoding phase; on the other hand, it can also be viewed as a form of reasoning itself, which would make it part of the solving phase. The latter view is emphasized by the recent rise of so-called *inprocessing* SAT solvers [JHB12, FBS19], which interleave search with phases during which typical preprocessing techniques are applied. In fact, since 2013 the SAT competitions have been dominated by CDCL solvers that utilize inprocessing. However, combining incremental SAT solving [Hoo93, KWS00, WKS01, ES03, ALS13, BBIS16] with preprocessing and inprocessing still poses a challenge in practice [KNPH05, KN11, NRS12, NR12, NRS14, FBS19] and will not be considered much further in this chapter.

From an abstract point of view, preprocessing techniques can be seen as implementations of inference rules that, when applied to formulas arising from real-world problem domains, yield equisatisfiable formulas that are easier to solve. Most preprocessing techniques are designed to simplify CNF formulas on the syntactic level by reducing the number of variables and the number of clauses. A popular example of this is again the method of bounded variable elimination, which uses Davis-Putnam resolution [DP60] and other techniques to eliminate variables without increasing the number of clauses.

Reducing the size of formulas, however, does not always improve the efficiency of the solving process. Sometimes SAT solvers actually benefit from redundant formula parts, which makes the search for effective preprocessing more complex than the search for methods that just yield equisatisfiable formulas of smaller size. The most evident example of this is the clause learning mechanism at the heart of CDCL solvers (Chapter 4 on CDCL): clause learning drastically improves the solving process by adding clauses that are logically entailed by the input formula and thus, by definition, redundant. Additionally, binary clauses (i.e., clauses with only two literals) are often considered special and thus not removed by SAT solvers due to their high potential for enabling more propagation.

A drastic example illustrating the worst-case impact of preprocessing techniques that *remove* clauses comes from the theoretical study of proof systems (Chapter 7): In 1985, Haken proved that a natural SAT encoding of the pigeonhole principle (PHP) admits only resolution proofs of exponential size [Hak85], meaning that CDCL solvers—which are based on resolution—require exponential

[1] While tight integration of preprocessing with search is now more common there also exist stand-alone preprocessing tools [Man12, WvdGS13] in the spirit of SatELite [EB05].

time to solve this encoding [BKS04, PD11]. But, as shown by Cook [Coo76], if we add specific clauses (so-called *definition clauses*) to the PHP encoding, the resulting CNF formulas have polynomial-size resolution proofs, meaning that CDCL solvers can, at least in theory, solve them in polynomial time.

Unfortunately, popular preprocessing techniques such as *bounded variable elimination*, *blocked clause elimination* [JBH10], and *cone-of-influence reduction* (a circuit-preprocessing technique described in Chapter 27) remove these definition clauses, turning an instance that is tractable for CDCL solvers into an intractable one. In practice, some of the most effective preprocessing techniques still aim at removing formula parts, but others also add strong redundant clauses such as binary or even unit clauses.

Another approach to analyze the effects and limits of preprocessing is offered by *kernelization* as studied in the area of fixed-parameter complexity, the topic of Chapter 17. However, while kernelization is an important theoretical concept in the context of SAT, it has—unlike the other techniques discussed in this chapter—so far not been used effectively in practical preprocessing.

Classifying practical preprocessing techniques in terms of their underlying inference rules is difficult, but we can identify two main categories: The first category consists of techniques that employ the resolution rule and its variations; examples are *unit propagation*, *bounded variable elimination* [SP04, Bie04, EB05], *hyper binary resolution* [Bac02, BW03, GS05, HJB13], and other advanced probing techniques such as *distillation* [HS07] and *vivification* [PHS08].

The second category consists of preprocessing techniques that identify literals or clauses that are redundant under different notions of redundancy; examples of such techniques are *clause subsumption*, *hidden literal elimination* [HJB11], *equivalent-literal substitution* (called *equivalence reduction* in [Bac02, BW03]), and *blocked clause elimination* [JBH10] together with its generalizations [HJL+15, KSTB16, KSTB18, KSS+17, HKB17, HKB20].

Preprocessing techniques also vary in the ways in which they analyze formulas. Most techniques are applied syntactically on CNF formulas, but some utilize graph representations of implication chains and other types of functional dependencies, or even higher-level constraints—implicitly represented by clauses—to preprocess a formula.

Although many individual preprocessing techniques are by themselves quite simple, the potential of preprocessing and inprocessing lies in combining different preprocessing techniques in such a way that they benefit from each other. For instance, at points where one technique is unable to make further progress, another technique might be applicable, and might even modify the formula in ways that trigger the first technique again.

In practice, however, running a specific preprocessing technique until completion is often only feasible for techniques that are computationally inexpensive. Because of this, even stand-alone preprocessors such as SatELite [EB05] rely on limits that bound the maximum number of occurrences of a candidate variable to be eliminated. Without such limits they would not be beneficial and run out of time on many instances. Moreover, scheduling the effort spent on individual preprocessing techniques is not trivial and has not seen much published research to date [Bie14]. Different preprocessing techniques can also have similar effects

in the sense that they perform the same modifications; for instance, both blocked clause elimination and bounded variable elimination have been shown to simulate cone-of-influence reduction on the CNF-level [JBH12].

Finally, the naive incorporation of preprocessing and inprocessing into the solving pipeline can potentially lead to unexpected outcomes, including incorrect solving results. Especially in inprocessing SAT solvers, a lot of care [JHB12] is required to ensure that simplification techniques interact in a sound way with both clause learning and *clause forgetting* (learned clauses are *deleted* quite frequently during the *reduce* phase of a SAT solver to improve memory usage and speed up propagation). Additionally, since many preprocessing techniques only maintain equisatisfiability but not logical equivalence, solution-reconstruction methods are sometimes required to transform a satisfying assignment of a preprocessed formula into a satisfying assignment of the original formula.

Most techniques described in this chapter are also available in the SAT solver CaDiCaL [Bie17]. The goal of developing CaDiCaL was to produce a SAT solver that is thoroughly documented, easy to understand and modify, and in the end still comparable to the state of the art (in the SAT Race 2019 it actually solved the largest number of instances). Therefore, consulting the CaDiCaL source code and its (inlined) documentation at `https://github.com/arminbiere/cadical` in parallel to reading this chapter is highly recommended. As CaDiCaL has been participating in the SAT competition since 2017, its solver descriptions [Bie17, Bie18, Bie19, BFFH20] provide additional technical details.

In the rest of this chapter, we give an overview of key preprocessing techniques proposed in the literature. Although our main focus is on CNF-level preprocessing, we will also briefly discuss main ideas proposed for structure-based preprocessing on the level of Boolean circuit representations of propositional formulas. The chapter is organized into sections describing CNF-level techniques, from basic concepts such as unit propagation (Section 9.2), over more advanced resolution-based preprocessing such as bounded variable elimination (Section 9.3), to preprocessing beyond resolution-based techniques, such as clause elimination (Section 9.4), as well as how to reconstruct solutions to original input CNF formulas after preprocessing (Section 9.5). Beyond CNF-level techniques we provide an additional shorter overview of preprocessing techniques applicable on higher-level propositional representations (Section 9.6), and conclude the chapter with a short summary and future work (Section 9.7).

9.2. Classical Preprocessing Techniques

We begin by discussing various classical CNF-level preprocessing techniques: unit propagation (the main propagation mechanism in complete SAT solvers), pure literal elimination, basic clause elimination techniques, and the detection of connected components in a graph representation of formulas to identify formula parts that can be solved independently of each other.

9.2.1. Unit Propagation

Unit propagation is a straightforward technique that has been common in SAT solving for many decades [DP60]; it is based on the *unit-clause rule*: Given a

formula in conjunctive normal form, the unit-clause rule checks if the formula contains a *unit clause*, i.e., a clause that contains only a single literal. If so, it removes all clauses that contain the literal from the formula (this step can be seen as an instance of subsumed-clause removal) and it removes the negation of the literal from the other clauses. Unit propagation repeatedly applies the unit-clause rule until either it derives the empty clause or there are no more unit clauses left. In the former case, we say that unit propagation derives a *conflict*.

Example 1. Consider the formula $(x) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z \vee v)$. The unit-clause rule takes the unit clause (x) and first removes it from the formula. It then removes the literal $\bar{x}$ from the clause $(\bar{x} \vee y)$, resulting in the formula $(y) \wedge (\bar{y} \vee z \vee v)$. Another application of the unit-clause rule first removes the unit clause (y) and then removes the literal $\bar{y}$ from $(\bar{y} \vee z \vee v)$. We end up with the formula $(z \vee v)$.

Clearly, whenever a formula contains a unit clause, the formula can only be satisfied by assignments that make the literal of the unit clause true. The application of unit propagation during preprocessing is therefore safe insofar as the resulting formula and the original formula are equisatisfiable. If unit propagation derives a conflict, we can conclude that the formula must be unsatisfiable. The converse, however, is not true. For example, the formula $(x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y})$ is unsatisfiable, but since it does not contain any unit clauses, unit propagation cannot derive a conflict.

Unit propagation (as well as several other preprocessing techniques we discuss later on) can be seen as the repeated application of the classical *resolution* rule [Rob65]. Given two clauses $(c_1 \vee \cdots \vee c_m \vee l)$ and $(d_1 \vee \cdots \vee d_n \vee \bar{l})$, the resolution rule allows us to derive the clause $(c_1 \vee \cdots \vee c_m \vee d_1 \vee \cdots \vee d_n)$, which is called the *resolvent* of the two clauses upon l. In case of unit propagation, the removal of the literal $\bar{l}$ from a clause can be seen as a resolution step with the clause (l) and thus is also often called *unit resolution*. In the context of constraint programming, unit propagation is also called *Boolean constraint propagation* (BCP). Unit propagation is essential for the original search-based D(P)LL algorithm [DLL62] as well as for the CDCL procedure described in Chapter 4.

9.2.2. Failed Literals

The notion of a *failed literal* [LB01, LaPMS03] forms the basis of a preprocessing technique known as *failed-literal probing*, which is an integral part of look-ahead solving (see Chapter 5). The origin of failed literals dates back to Zabih and McAllester [ZM88], and is featured in the thesis of Freeman [Fre95]. The related SLUR algorithm by Schlipf et. al. [SAFS95] has been used to characterize a class of polynomially solvable formulas (see also [dV00] for a related line of work).

A literal l is a failed literal with respect to a formula F if unit propagation derives a conflict on $F \wedge (l)$. Clearly, if unit propagation derives a conflict on $F \wedge (l)$, then $F \wedge (l)$ is unsatisfiable and therefore F implies $\bar{l}$. The addition of the unit clause $(\bar{l})$ to F thus preserves equivalence, and the corresponding preprocessing technique that adds the negations of failed literals to a formula is called failed-literal probing.

Example 2. Let $F = (x \vee u) \wedge (\bar{x} \vee u) \wedge (\bar{u} \vee z \vee y) \wedge (\bar{u} \vee z \vee \bar{y})$. Since F does not contain any unit clauses, it cannot be simplified by unit propagation alone. However, $\bar{u}$ is a failed literal because unit propagation derives a conflict on $F \wedge (\bar{u})$, caused by the first two clauses. The unit clause (u) is therefore implied by F and so we can add it to F. If we then perform unit propagation on $F \wedge (u)$, we obtain the simplified formula $G = (z \vee y) \wedge (z \vee \bar{y})$.

In practical preprocessing, a list of candidate literals is considered, and for all the candidate literals it is checked whether or not they are failed literals. It is generally not clear how to order the candidate literals to minimize the number of required decisions. For instance, in Example 2, the literal $\bar{u}$ is the only failed literal with respect to F. In the worst case, we would thus have tested all other literals before $(\bar{u})$. After simplifying F, which yields G, we can see that $\bar{z}$ becomes a failed literal, but z, y, and $\bar{y}$ are still no failed literals. Simplifying a formula until there are no more failed literals can thus require us to test quadratically many literals (with respect to the number of variables) in the worst case. Assuming the strong exponential-time hypothesis (ETH), the quadratic upper bound is tight in terms of both the number of variables and the number of clauses, even for Horn-3-CNF formulas [JK14]. As each test performs unit propagation, this results in cubic accumulated complexity in the worst case.

The cubic complexity can be reduced by marking the literals that were implied by unit propagation in tests since the last time a failed literal has been detected. These literals do not have to be tested until another failed literal is found, since their propagation did not lead to a conflict. As soon as a test yields a conflict—meaning that a new failed literal is found—all the marked literals have to be unmarked again (it is more efficient to use time-stamping techniques though). This optimization was independently discovered and described in [ABS99] and [SNS02].

Another optimization consists of testing only literals that do not occur in binary clauses themselves (but their negation does), thus effectively only testing the roots of the binary implication graph [GS05, Bie17].[2] With these techniques, failed-literal probing can run until completion even on large industrial SAT instances or can be used as preprocessing for splitting and partitioning in distributed SAT solving [HJN10]. Later in this chapter we will discuss an extension, based on look-ahead (see also Chapter 5), that learns binary clauses on the fly by combining *hyper binary resolution* with equivalent-literal reasoning.

9.2.3. Pure Literals

A pure literal is a literal whose negation does not occur in the formula. The corresponding technique of pure-literal elimination removes all clauses that contain a pure literal, because these clauses can be trivially satisfied by making the pure literal true without falsifying any other clauses.

Example 3. Consider the formula $(\bar{x} \vee y \vee z) \wedge (\bar{y} \vee \bar{z})$. The literal $\bar{x}$ is a pure literal in this formula whereas all the other literals are not pure.

[2] When focusing on roots, cyclic literal dependencies, as in $(a \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (a \vee b)$, might prevent detection of some failed literals though. This can be avoided by computing strongly connected components and, for instance, equivalent-literal substitution as discussed in Section 9.3.2.

Pure-literal elimination was introduced independently both by Davis and Putnam [DP60] and by Dunham, Fridshal, and Sward [DFS59]. While a failed literal can be seen as a literal which we make false because making it true would lead to a conflict, a pure literal can be seen as a literal which we make true because making it true cannot lead to a conflict. Moreover, if a formula is unsatisfiable, clauses containing a pure literal cannot be used productively in a resolution proof because pure literals can never be resolved away.

An alternative to removing clauses that contain pure literals is to add the pure literals as unit clauses. The addition of these unit clauses, however, does not always preserve logical equivalence, because it could remove potential models in which a pure literal is false. Nevertheless, adding pure literals as unit clauses does not change the satisfiability status of a formula. This idea of performing transformations that yield equisatisfiable—but not necessarily logically equivalent—formulas is the corner stone of more advanced preprocessing techniques, such as blocked clause addition, which we discuss later.

9.2.4. Subsumption

A clause C is *subsumed* by a clause D if the set of literals in C is a superset of the literals in D. Analogously, a clause is subsumed by a formula if it is subsumed by some clause in the formula. The removal of subsumed clauses is an important technique that is often referred to as *subsumption*. The idea behind subsumption is to eliminate clauses that are redundant in the sense that their removal does not affect the satisfiability of a formula: if a formula contains two clauses C and D, where D subsumes C, the removal of C yields a logically equivalent formula since every assignment that satisfies D must also satisfy C.

Example 4. The clause $(\bar{x} \vee y \vee z)$ is subsumed by the clause $(\bar{x} \vee z)$ since the set $\{\bar{x}, y, z\}$ is a superset of the set $\{\bar{x}, z\}$. Whenever $(\bar{x} \vee z)$ is true, $(\bar{x} \vee y \vee z)$ must also be true.

We have already seen an application of subsumption earlier: During unit propagation, when a unit clause (l) is found, all the clauses containing the literal l are removed from the formula. A corner case is the empty clause, which subsumes all other clauses in a formula.

9.2.4.1. Forward Subsumption

Forward subsumption starts with a given clause C and a formula F, and checks if C is subsumed by F. A simple technique to check if a clause is forward subsumed by a formula F temporarily marks all literals of C as false and then checks whether there is another (usually smaller) clause D in F with only false literals.

In order to find all potential candidates D without iterating over F, at least one literal of every clause in F needs to be "watched". Two-watched literal schemes [ZS00, MMZ+01], as commonly used in CDCL solvers (see Chapter 4), suffice too, while *occurrence lists* (an occurrence list contains *all* the clauses in which a literal occurs) are not necessary. A similar technique has been described in [Bra04] and refined in [Zha05].

9.2.4.2. Backward Subsumption

In contrast to forward subsumption, which checks if a clause C is subsumed by a clause D in F, backward subsumption checks if F contains clauses C that are subsumed by a given clause D, and if so, removes them. Backward subsumption is actually more common in practice; for example, in SAT solvers using clause learning, learned clauses are never forward subsumed, but they might subsume other clauses of the formula, in particular clauses that were learned recently.

As described in Exercise 271 of [Knu15] and its solution, this situation might occur frequently for certain formulas, and it is possible to *eagerly subsume recently learned clauses* efficiently, by reusing the markings of the literals in the learned clause at the end of conflict analysis (see Chapter 4 on CDCL again). Even though this technique is efficient and simple to implement, subsumption algorithms are more useful in combination with variable elimination (see Section 9.3), because variable elimination focuses on irredundant original clauses and is allowed to ignore redundant learned clauses [JHB12].

If an implementation maintains for each literal an occurrence list instead of using a one-watched [Zha05] or two-watched [ZS00, MMZ+01] literal scheme, then the search for a candidate clause C, checked for being backward subsumed by the given clause D, can be restricted to iterate over the occurrences of a single literal in the given clause D, chosen as one with the smallest number of occurrences (shortest occurrence list). This observation, originally made in the context of QBF solving [Bie04], is crucial for scaling backward subsumption to large formulas, and was first used for SAT preprocessing in SatELite [EB05].

Another technique to improve the performance of backward subsumption in practice is based on so-called *signatures* [Bie04], which are an instance of Bloom filters [Blo70]. The basic idea is to avoid costly traversals of occurrence lists in the (frequent) case where no subsumed clauses can be found. This is achieved by checking a signature condition that must hold if there is a clause that can be subsumed by the given clause. If the condition does not hold, the traversal can be avoided. For details, we refer to the original paper.

A related backward-subsumption algorithm is due to Brafman [Bra04]. Given a clause D and a formula F, the algorithm iteratively computes the set

$$S = \bigcap_{l \in D} \mathit{clauses}_F(l),$$

where $\mathit{clauses}_F(l)$ denotes the set of all clauses of F in which the literal l occurs. The final set S contains all clauses of F that are subsumed by D. As soon as the algorithm detects that S is empty, it can conclude that D does not subsume any clauses. The original algorithm does not need to save signatures for clauses and literals, but it needs to save the occurrence lists for all the literals in F. In principle, though, it could also make use of signatures.

Backward subsumption can easily be turned into an algorithm that removes all subsumed clauses from a formula: start with the empty formula and one-by-one add original clauses with largest clauses first. Then check for each newly added clause whether it subsumes any of the previous clauses. This observation is particularly useful for applying subsumption during inprocessing. In practice,

it is still advisable to limit the effort spent on subsumption checks. This can, for instance, be done by limiting the maximum length of traversed occurrence lists.

9.2.4.3. Self-Subsuming Resolution

As observed in [EB05], subsumption is closely related to *strengthening* clauses, more precisely to *self-subsuming resolution* as implemented in SatELite [EB05]. Consider two clauses $C \vee l$ and $D \vee \bar{l}$, which can be resolved upon l. If D subsumes C, then the resulting resolvent is simply C, which in turn subsumes the first antecedent $C \vee l$. Thus, instead of adding the resolvent C, we can replace $C \vee l$ by C, in effect strengthening the clause $C \vee l$ by removing the literal l. This technique is called self-subsuming resolution.

Subsumption algorithms can easily be extended to perform self-subsuming resolution. The only necessary change is to modify the subsumption check between two clauses: when checking if D subsumes C, we simply allow that at most one literal $\bar{l}$ of D occurs negated in C. If this is the case, and all other literals of D occur in C, we can apply self-subsuming resolution to C and D by removing l from C. Checking that the literals of D occur in C (with one possible exception) can be achieved by marking literals (possibly with time stamps) or by a merge-sort-style check, assuming the literals in clauses are sorted [BP11].

Self-subsuming resolution was independently discovered as *global subsumption* for instantiation-based first-order theorem proving [Kor08] and actually was introduced in the SAT context via the notion of a *subsuming resolvent* in [OGMS02] (though without detailed experiments). In [EB05], it was partially motivated by the fact that clause learning (Chapter 4), and particularly clause minimization [SB09], implicitly rely on it. *On-the-fly subsumption* [HJS10, HS09] is another related technique that uses self-subsuming resolution to strengthen antecedent clauses used for deriving learned clauses in CDCL.

9.2.4.4. Implementations

Finding subsumed clauses in CNF formulas is closely related to finding *extremal sets* in data mining. In this context, Bayardo and Panda [BP11] showed that backward-subsumption algorithms developed in the SAT community [EB05] were competitive for this problem too, but their paper also presents a substantially faster algorithm, which in our setting translates to forward-subsumption checking and yields a substantial performance improvement also for SAT.

The basic idea is to use a one-watched-literal scheme and only watch literals with the fewest occurrences. Clauses are added again one-by-one, but now starting with the smallest clauses first. All one-watched-literal lists of the literals in a processed clause are traversed to find clauses which subsume the processed clause. If this is not the case (i.e., the forward-subsumption check fails), then the clause is added, and the literal with the shortest watch list (at this point) is watched. As already discussed above, the actual subsumption check between a previously added clause and the processed clause can be made faster too (for instance, by sorting literals in clauses [BP11]) and extends to self-subsuming resolution.

In the context of Quantor [Bie04], SatELite [EB05], and MiniSat [ES06] backward subsumption was considered superior. However, forward-subsumption

Figure 1. Hypergraph Representation of a CNF Formula.

algorithms inspired by ideas of [BP11], as implemented in Splatz [Bie16] and CaDiCaL [Bie17], become more important, particularly during the process of removing from a given CNF all subsumed clauses and performing self-subsuming resolution until completion. These forward-subsumption algorithms are much more efficient than previously used backward-subsumption algorithms and can even regularly be applied to learned clauses during inprocessing too. Interleaved with variable elimination (see Section 9.3), which needs full occurrence lists in any case, there is still a benefit in using backward subsumption, since it allows to focus on trying to subsume or strengthen other clauses by newly derived clauses to trigger new variable-elimination attempts (as for instance in [BFFH20]).

9.2.5. Connected Components

The solving process for a formula can be significantly simplified by taking its underlying graph structure into account. One way to represent a formula as a graph is to define a hypergraph where variables correspond to vertices and clauses correspond to hyperedges connecting their variables. For example, the formula

$$(x \vee y \vee \bar{z}) \wedge (\bar{y} \vee z) \wedge (u \vee \bar{v}) \wedge (v \vee \bar{w})$$

can be represented by the hypergraph shown in Figure 1. As we can see in this example, the graph contains two different connected components.

It has been observed that formulas in practical SAT solving often contain multiple connected components [BS06]. Each of these components can be seen as an independent SAT problem, with the whole formula being satisfiable only if all of its components are satisfiable. Splitting a formula into its connected components during preprocessing and then solving these components independently has several advantages: the independent components can be solved in parallel, and as soon as one component is identified as unsatisfiable, it can be concluded that the whole formula is unsatisfiable. Even if all components are satisfiable, the parallelism can speed up the solving process. Similar ideas are used in component caching for model counting [BDP03, SBB+04], where the formula is preprocessed at every search node and decomposed into disconnected components for which the model count can be computed independently. The problem of model counting is also called #SAT and extensively covered in Chapter 25 and 26.

In the context of CDCL SAT solving (Chapter 4), formulas consisting of independent components can be simpler to solve even without relying on parallelism. Suppose a typical CDCL solver tries to find a satisfying assignment for a formula that consists of several independent components. During search, the solver might find an assignment that already satisfies some of the components, but then realize that the assignment falsifies another component. What the solver usually does in

that situation is to backjump and thereby undo some of its variable assignments. However, this could undo some of the assignments that are independent of the falsified component and thereby unassign components that were already satisfied.

By splitting up a formula, such a situation can be avoided. However, the same effect can be achieved by saving the last assigned truth value (also called *phase*) of a variable and then always assigning that saved value when making decisions. This *phase saving* technique was introduced in the RSat solver [PD07] and has been standard in SAT solvers since then, making explicit component decomposition obsolete, at least for sequential plain CDCL solving.

Since 2018 several state-of-the-art SAT solvers quite frequently reset saved phases [Bie18, Bie19, BFFH20, ZC20, SCD+20, SSK+20], which counteracts the effect of phase saving to remember satisfying assignments for components. As an alternative to explicit component decomposition as in [BS06], it was proposed in [BFFH20] to simply remove all clauses satisfied by the largest autarky within saved phases following ideas in [KHB19], an application of autarky reasoning of Chapter 14 within SAT solving.

9.3. Resolution-Based Preprocessing

We continue by discussing preprocessing techniques that rely heavily on resolution, starting with bounded variable elimination, which as high-lighted already in the introduction is still considered the most important preprocessing technique in practical SAT solving.

9.3.1. Bounded Variable Elimination

Bounded variable elimination [Bie03, Bie04, SP04, EB05] is based on the technique of *clause distribution*, which lies at the core of the original Davis-Putnam procedure [DP60]. To perform clause distribution, we choose a variable, add all resolvents upon this variable to the formula, and remove the original clauses containing the variable.

Example 5. Consider the formula

$$F = (x \vee e) \wedge (y \vee e) \wedge (\bar{x} \vee z \vee \bar{e}) \wedge (y \vee \bar{e}) \wedge (y \vee z). \tag{9.1}$$

To perform clause distribution with the variable e, we first add all resolvents upon e. The clauses $(x \vee e)$ and $(y \vee e)$ can both be resolved with $(\bar{x} \vee z \vee \bar{e})$ and $(y \vee \bar{e})$. We thus add the corresponding four resolvents to obtain the formula

$$F \wedge (x \vee \bar{x} \vee z) \wedge (x \vee y) \wedge (y \vee \bar{x} \vee z) \wedge (y).$$

Now we remove all clauses that contain e to obtain

$$(y \vee z) \wedge (x \vee \bar{x} \vee z) \wedge (x \vee y) \wedge (y \vee \bar{x} \vee z) \wedge (y). \tag{9.2}$$

Repeated clause distribution can increase the number of clauses exponentially, rendering its unbounded use as a preprocessing technique unaffordable. Actually, already eliminating a single variable might increase the size quadratically. To

deal with this problem, the original technique of bounded variable elimination applies clause distribution only on variables whose elimination does not increase the number of clauses, which explains the name *bounded* variable elimination.

However, as proposed in the variable-elimination procedure of the QBF solver QUANTOR [Bie04], it has become common in SAT to relax the bound on additional clauses: in *incrementally relaxed bounded variable elimination*, introduced in the SAT solver GLUEMINISAT [NII15], the bound is increased[3] incrementally every time a round of variable elimination has completed without increasing the size of the formula too much. This technique was ported to the highly influential SAT solver COMINISATPS [Oh16] that has formed the basis of the MAPLESAT series of SAT solvers [LGPC16, LLX+17, NR18], which ranked at the top in the SAT competitions from 2016 to 2018.

Since bounded variable elimination produces many redundant clauses, it is often combined with tautology elimination (as in NIVER [SP04]) as well as with subsumption and strengthening (in SATELITE [EB05]). Subsumption might lead to further variables being eliminated, triggered by removing clauses or literals.

Example 6. Consider again the formula (9.2), which resulted from the formula (9.1) by eliminating the variable e. We can observe that the resolvent $(x \vee \bar{x} \vee z)$ is a tautology and that it can therefore be removed from the formula. Moreover, the clauses $(y \vee z)$, $(x \vee y)$, and $(y \vee \bar{x} \vee z)$ are all subsumed by the clause (y), and so they can be eliminated as well. Thus, after eliminating only a single variable, we end up with the trivially satisfiable formula (y).

On the implementation side, clause distribution is best interleaved with subsumption using the generated resolvents immediately to subsume and strengthen existing clauses through backward subsumption. Furthermore, bounded variable elimination can be improved by on-the-fly subsumption during variable elimination [HS09] as follows: Whenever a (non-tautological) resolvent R of two clauses C and D is computed, check if $|R| = |C| - 1$ or $|R| = |D| - 1$ (after removing repeated literals, i.e., interpreting clauses as literal sets). In the former case, C is subsumed by R and so it can be replaced by R; in the latter case, D is subsumed by R and so it can also be replaced by R. Moreover, the replacement can be performed regardless of whether e is actually eliminated or not (in case elimination of e would produce too many new clauses). Variables of removed clauses or literals have to be (re)scheduled as candidates for further elimination attempts.

Another important implementation detail is how to organize the schedule of elimination candidates. Common practice is to use a priority queue implemented as a binary heap in which the variables are ordered by their number of occurrences within the formula. Variables with the smallest number of occurrences are tried first, and the priority queue is updated dynamically as clauses are strengthened, removed, and added. In order to scale variable elimination to large formulas, variables that—when resolved—produce large resolvents or occur in large or in too many clauses, should not be eliminated. Limits vary by implementation, but typically the clause size or the resolvent size is limited to 20 to 100 literals. Variables that occur more than 100 to 1000 times, either negatively or positively, are typically skipped too.

[3] For instance, the bounds form a geometric series $0, 8, 16, 32, \ldots, 8192$.

The effectiveness of bounded variable elimination is highlighted by the observation that it automatically performs the earlier discussed elimination of pure literals, because there are no resolvents upon pure literals. Moreover, if a formula contains a unit clause (e) and either on-the-fly subsumption is performed or the formula contains no clauses that are subsumed by (e), then the result of eliminating the variable e is the same as applying the unit-clause rule with (e).

As observed in [EB05], the elimination of variables for which the CNF contains a functional definition produces redundant clauses; these clauses are actually resolvents of clauses that are not part of the functional definition. Thus, searching for such definitions (see Section 9.6.2) during elimination attempts reduces the number of required resolvents and often allows more variables to be eliminated.

Alternatively, as implemented in LINGELING [Bie10], it is possible to simplify the CNF of resolvents on the fly, for instance by applying "semantically" the dual of Minato's algorithm for producing an irredundant sum-of-products [Min92] to CNF [EMS07]. The idea is to compile the CNF of the clauses containing the variable to be eliminated into a "semantic" representation, such as a binary decision diagram (BDD) [Bry86], eliminate the variable from the BDD by existential quantification, and then as in [EMS07] encode the resulting BDD back into an irredundant CNF (using Minato's algorithm). Instead of using BDDs *function tables* represented as bit-maps as in LINGELING will also do. In any case, this approach avoids searching for functional definitions explicitly and captures all types of functional dependencies. Otherwise it reduces the number of produced clauses at least as much as the explicit method [EB05] discussed above.

For the sake of completeness, it should also be mentioned that there exist symbolic variants of bounded variable elimination that use decision diagrams [CS00, vDEB18], more precisely *zero-suppressed decision diagrams* (ZDD) [Min93]; but in practice they currently work only on very restricted sets of formulas (e.g., pigeon-hole formulas).

Finally, bounded variable elimination can also be applied "in reverse", resulting in what is called *bounded variable addition* (BVA), as proposed in [MHB12]. Since the elimination of variables can increase the number of clauses, performing variable elimination in reverse (adding variables instead of eliminating them) can potentially shrink the size of a formula. Requiring to distinguish between two types of variables (i.e., original and added) makes the implementation of bounded variable addition much more problematic than implementing bounded variable elimination, particularly in the context of incremental SAT solving, where also the need of "recycling" variables might become an issue. In contrast to bounded variable elimination, bounded variable addition has therefore not yet established itself as a central preprocessing technique.

9.3.2. Techniques Based on Implication Graphs

Analyzing the structure of binary clauses (i.e., clauses that contain exactly two literals) in a CNF formula gives rise to preprocessing techniques through the notion of a (*binary*) *implication graph* [APT79] (BIG). Given a CNF formula, its binary implication graph is a directed graph that is obtained as follows: Introduce a vertex for every literal, and add the edges $(\bar{l}, k)$ and $(\bar{k}, l)$ for every binary clause $(l \vee k)$, where l and k are (possibly negative) literals.

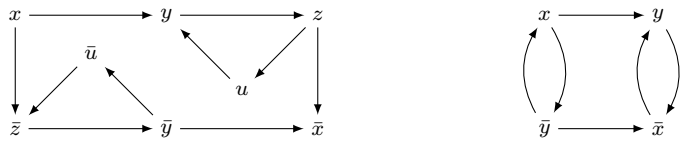

Figure 2. The original (binary) implication graph (BIG) for the formula of Example 7 is shown on the left and on the right the result after one round of equivalent-literal substitution.

Example 7. The implication graph of the formula

$$(\bar{x} \vee y) \wedge (\bar{y} \vee z) \wedge (\bar{z} \vee u) \wedge (\bar{u} \vee y) \wedge (\bar{x} \vee \bar{z}) \wedge (x \vee z \vee u)$$

is shown on the left in Figure 2. Notice that the ternary clause $(x \vee z \vee u)$ does not affect the implication graph.

An edge in the implication graph directly represents an implication that is equivalent to a binary clause in the formula. In the example, x implies y, y implies z, z implies u, and so on. Notice also that for every edge of the form (e_1, e_2), the implication graph contains a corresponding edge $(\bar{e}_2, \bar{e}_1)$, representing the contraposition of the implication.

By construction, every *strongly connected component*[4] of a binary implication graph represents a set of *equivalent literals* [VGT93, Li00], i.e., literals who must have the same truth value in every satisfying assignment of the formula. The set of literals prescribed by a strongly connected component can therefore be replaced by a single representative literal in the underlying CNF formula; this replacement is known as *equivalence reduction* and *equivalent-literal substitution* (ELS).

Example 8. In the implication graph on the left in Figure 2, the literals y, z, and u form a strongly connected component. Their negations $\bar{y}$, $\bar{z}$, and $\bar{u}$ also form a strongly connected component. Since a clause $(x \vee y)$ is represented by the two edges $(\bar{y}, x)$ and $(\bar{x}, y)$ in the binary implication graph, every strongly connected component has a corresponding strongly connected component that consists of its complementary literals. In the example, we can replace the strongly connected component $\{y, z, u\}$ by the vertex y, and the strongly connected component $\{\bar{y}, \bar{z}, \bar{u}\}$ by the vertex $\bar{y}$, to obtain the simplified formula

$$(\bar{x} \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (x \vee y).$$

Note that the clause $(x \vee y)$ stems from the original ternary clause $(x \vee z \vee u)$. The simplified formula corresponds to the implication graph depicted on the right of Figure 2 (with two new strongly connected components).

There are several different implementations for finding and replacing strongly connected components [APT79, dV01, BW03, VG05, GS05] in the context of SAT. For details, we refer to the respective literature, but usually the implementations are based on a depth-first search of the implication graph.

[4] A strongly connected component is a maximal set of vertices such that every vertex in the set is reachable from all other vertices in the set.

In certain cases, strongly connected components can even be used for detecting unsatisfiability: if both a literal l and its complement $\bar{l}$ are contained in the same strongly connected component, then the formula is unsatisfiable. Additionally, the implication graph can be used for the detection of failed literals. In particular, if there is a path from a literal l to its complement $\bar{l}$, then l is a failed literal [VG05].

Example 9. In the binary implication graph shown in Figure 2, x is a failed literal because there is a path from x to $\bar{x}$. Note, however, that the implication graph does not always contain a path from a failed literal l to its complement $\bar{l}$. To see this, consider the formula $F = (\bar{x} \vee y \vee z) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{x} \vee \bar{z})$ and the literal x. Although it can be checked that x is a failed literal with respect to F, there is no path from x to $\bar{x}$ in the implication graph, because the ternary clause $(\bar{x} \vee y \vee z)$, without which x would not be a failed literal, is not considered when constructing the implication graph.

9.3.3. Hyper Binary Resolution

We have already seen in the context of unit propagation how a formula can be simplified by continuously performing inference steps (namely, the *unit-clause rule*) until either a conflict is derived or no more inference steps are applicable. Hyper binary resolution is based on the same idea, but uses stronger inferences.

Towards hyper binary resolution, consider first *binary resolution*. The binary-resolution rule is obtained from the ordinary resolution rule by restricting it to binary clauses. A binary-resolution step can either produce another binary clause—for instance, if we resolve $(x \vee y)$ with $(\bar{x} \vee z)$ to obtain $(y \vee z)$—or it can yield a unit clause—for instance, if we resolve $(x \vee y)$ with $(\bar{x} \vee y)$ to obtain (y). Applying the binary-resolution rule and the unit-clause rule until none of them is applicable anymore can simplify a formula more aggressively than performing only unit propagation.

Example 10. Let $F = (x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y})$. As there are no unit clauses in F, unit propagation has no effect on the formula. However, by performing binary resolution of $(x \vee y)$ with $(\bar{x} \vee y)$, we derive (y). After this, unit propagation can derive a conflict with the clauses (y), $(x \vee \bar{y})$, and $(\bar{x} \vee \bar{y})$.

As observed by Bacchus [Bac02], every unit clause that can be derived by performing binary resolution in combination with unit propagation (as described above) can also be derived by failed-literal detection, but not vice versa.

Example 11. With respect to the formula $F = (\bar{x} \vee y \vee z) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{x} \vee \bar{z})$ from Example 9 above, the literal x is a failed literal since unit propagation derives a conflict on $F \wedge (x)$. Therefore $(\bar{x})$ can be derived by failed-literal detection. However, both unit propagation and binary resolution do not affect F. The reason for this is that they cannot use the first clause, which contains three literals.

Simply performing all resolutions, also with longer clauses, would produce too many clauses. This leads to the definition of hyper binary resolution [Bac02], an instance of *hyper resolution* [Rob74], which contracts several resolution steps into a single inference step.

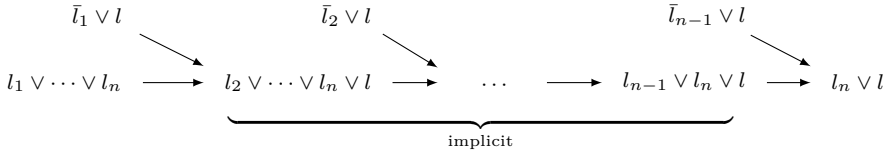

Figure 3. Hyper binary resolution. Only the last resolvent $(l_n \vee l)$ is actually added after a hyper binary resolution step. The intermediate resolvents are implicit.

Definition 12. Given an n-ary clause $(l_1 \vee \cdots \vee l_n)$ and $n-1$ binary clauses $(\bar{l}_1 \vee l), \ldots, (\bar{l}_{n-1} \vee l)$, *hyper binary resolution* allows the derivation of the clause $(l_n \vee l)$, which is called a *hyper binary resolvent*.

Observe that the hyper binary resolvent $(l_n \vee l)$ can be derived from the previous clauses by applying a number of resolution steps. We start with $(l_1 \vee \cdots \vee l_n)$ and first resolve it with $(\bar{l}_1 \vee l)$ to obtain $(l_2 \vee \cdots \vee l_n \vee l)$, which we then resolve with $(\bar{l}_2 \vee l)$ to obtain $(l_3 \vee \cdots \vee l_n \vee l)$ and so on until we finally derive $(l_n \vee l)$. The intermediate resolvents, however, are not derived explicitly. Figure 3 illustrates the (implicit) resolutions involved in a single hyper binary resolution step.

By repeatedly performing hyper binary resolution in combination with unit propagation until no more inference steps are possible (or a conflict is derived), a formula can be simplified significantly. In fact, this combination derives the same literals that would be derived by performing failed-literal detection on all literals in combination with unit propagation [Bac02]. It can, for instance, derive the literal $(\bar{x})$ from the formula F in Example 11 by a single application of the hyper binary resolution rule.

Hyper binary resolution and unit propagation have been used in combination with the detection of binary equivalences (replacing them by single literals, similar to the technique based on strongly connected components discussed in Section 9.3.2). The resulting preprocessing technique can be implemented based on probing failed literals in a depth-first search of the binary implication graph [BW03]. As soon as a large (non-binary) clause is used to derive a unit clause during probing, a hyper binary resolvent is learned. A less general but faster version of this technique, which only probes roots of the binary implication graph, has been implemented by Gershman and Strichman [GS05].

More sophisticated algorithms for hyper binary resolution [HJB13] reuse decisions and propagation effort by scheduling decisions along the structure of the binary implication graph, but using multiple decisions at the same time, similar in spirit to efficient "distillation" on tries [HS07], with more details provided at the end of the next section.

These approaches avoid "transitive" hyper binary resolvents by prioritizing unit propagation over binary clauses. For instance, the set $\{(\bar{a}_i \vee a_{i+1}) \mid 1 \leq i < n\}$ of binary clauses has quadratically many hyper binary resolvents $(\bar{a}_i \vee a_j)$, with $1 \leq i < j \leq n$. Note that these additional transitive clauses are redundant with respect to detecting strongly connected components of equivalent literals as well as unit propagation (see also the discussion of "empowering" in [PD11]) and thus should not be derived (and stored). It is further useful to apply transitive reduc-

tion of the binary implication graph pro-actively [HJB10a]. As first implemented in PrecoSAT [Bie09] and further refined in [HJS11], there is also an on-the-fly variant of hyper binary resolution that learns binary clauses during propagation using dominator analysis on the binary implication graph (see also [HJB13]).

However, removing transitive resolvents does not prevent the worst case of quadratically many non-transitive hyper binary resolvents [HJB13], and running hyper binary resolution until completion is only feasible for certain formulas (mostly small and hard combinatorial problems). For large application formulas, hyper binary resolution has to be preempted, and it is advisable to "forget" hyper binary resolved clauses aggressively [BFFH20] during the "reduce" phase of the SAT solver which deletes useless learned clauses.

9.3.4. Advanced Probing Techniques

We start with asymmetric[5] tautologies, which are a generalization of tautologies based on the concept of an *asymmetric literal*:

Definition 13. A literal l is an *asymmetric literal* in a clause $C \vee l$ with respect to a formula F if unit propagation deduces the unit clause $\bar{l}$ from $F \wedge \overline{C}$, where $\overline{C}$ is the conjunction of the negations of the literals in C.

Accordingly, if l is an asymmetric literal in $C \vee l$ w.r.t. F, then all models that falsify C but satisfy F, falsify l. In this case, l is redundant in $C \vee l$ and $F \wedge (C \vee l)$ can be simplified to $F \wedge C$ by "strengthening" $C \vee l$ and removing l. This simplification is also called *asymmetric literal elimination* (short ALE).

Example 14. Consider the formula $F = (a \vee \bar{x}) \wedge (x \vee \bar{l})$ and the clause $C = (a \vee b)$. Unit propagation on $F \wedge \overline{C} = F \wedge (\bar{a}) \wedge (\bar{b})$ produces the unit clause $(\bar{l})$. Therefore, l is an asymmetric literal in $C \vee l$ with respect to F, and $(a \vee \bar{x}) \wedge (x \vee \bar{l}) \wedge (a \vee b \vee l) = F \wedge (C \vee l)$ can be simplified to $(a \vee \bar{x}) \wedge (x \vee \bar{l}) \wedge (a \vee b) = F \wedge C$ through asymmetric literal elimination.

In order to deduce that clauses are "redundant" based on this notion, we add literals instead of removing them, and during this process obtain a tautological clause [HJB10a].

Definition 15. A clause C is an *asymmetric tautology* (AT) with respect to F if there is a sequence $l_1, \ldots, l_n$ of literals such that $C \vee l_1 \vee \cdots \vee l_n$ is a tautology and each l_i is an asymmetric literal in $C \vee l_1 \vee \cdots \vee l_i$ with respect to F.

Example 16. Consider the formula $F = (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z})$ and the clause $C = (\bar{x} \vee \bar{y})$. The literal $\bar{z}$ is an asymmetric literal in $(\bar{x} \vee \bar{y} \vee \bar{z})$ w.r.t. F since unit propagation on $F \wedge (x) \wedge (y)$ deduces the unit clause (z) (propagation of $F \wedge (x)$ is enough). Then z is an asymmetric literal in $(\bar{x} \vee \bar{y} \vee \bar{z} \vee z)$ w.r.t. F as unit propagation deduces also $\bar{z}$ from $F \wedge (x) \wedge (y) \wedge (z)$ (again already unit propagation of $F \wedge (y)$ suffices). At this point we can stop, since we have already shown that C in combination with F is equivalent to a tautology, thus an asymmetric tautology.

[5] The term "asymmetric" has its origin in the notion of "asymmetric (tableau) rules" [DM94], which apparently inspired the MiniSat authors [ES03] to name a command line option "`-asymm`" used to enable a "strengthening" procedure now called "asymmetric literal elimination".

Note that in this example we used the fact that unit propagation is confluent in the sense that the order in which the unit-clause rule is applied is irrelevant to the outcome; it is also monotonic in the sense that every unit clause that can be derived from a formula F can also be derived from every larger formula that contains all the clauses of F.

It can further be shown that if a clause C is an asymmetric tautology w.r.t. a formula F, then F implies C. This in turn means that C can be safely eliminated from the formula $F \wedge C$. Moreover, if only asymmetric literals from binary clauses need to be added to turn a clause into a tautology, then that clause is also called a *hidden tautology*, and the corresponding clause-elimination technique is called *hidden tautology elimination* (HTE) [HJB10a].

The related technique of *unhiding* [HJB11] avoids quadratic computation during repeated unit propagation. It is based on randomized depth-first traversal of the binary implication graph and relies on the *parenthesis theorem* to detect in almost constant time if a clause is a hidden tautology or if clauses can be strengthened by removing hidden literals. This form of literal removal is also known as *hidden literal elimination* (HLE), which can be seen a restricted variant of asymmetric literal elimination which only propagates over binary clauses.

It is well-known that a clause C is an asymmetric tautology with respect to a formula F if and only if it is a *reverse unit propagation* (RUP) clause [VG12].

Definition 17. A clause C is a RUP clause with respect to a formula F if unit propagation derives a conflict on $F \wedge \overline{C}$.

To test if a clause is an asymmetric tautology with respect to a formula F, it thus suffices to check if propagating its negation leads to a conflict. Note that RUP clauses are a generalization of the earlier-mentioned failed literals: a unit clause (l) is a RUP clause with respect to a formula F if and only if $\bar{l}$ is a failed literal with respect to F. From the RUP definition it also becomes clear why asymmetric tautologies are implied, since unit propagation can only derive a conflict if the conjunction of the formula with the negated clause is unsatisfiable, which is the case if and only if the clause is implied by the formula.

Failed-literal probing alone is often already quite expensive. Thus checking all clauses for being asymmetric tautologies is even more costly. Still, removing them can benefit other preprocessing algorithms such as variable elimination.

The basic approach of asymmetric tautology checking takes a clause C, then assigns all its literals to false, followed by unit propagation. Advanced probing techniques like *distillation* [HS07] and *vivification* [PHS08] interleave assignments and propagations instead. This works as follows.

The literals of C are still assigned to be false in an arbitrary fixed order, but a complete round of propagation is started immediately after every single assumed assignment. If during such a propagation round a conflict is found or another not yet assigned literal of C is forced to true, then the clause is an asymmetric tautology and can in principle be removed. Alternatively, particularly if applied to learned clauses in CDCL [LLX+17, LXL+20], such asymmetric tautologies can be replaced by clauses learned through the standard CDCL conflict analysis, if, for instance, the learned clause turns out to be shorter. If, however, during propagation another not yet assigned literal of C is forced to false, then this

literal is actually an asymmetric literal and the clause C can be strengthened, by removing the literal from C. This presents the real benefit of interleaving assignments and propagation in vivification and distillation.

A disadvantage of clause-based probing, in contrast to literal based probing discussed before, is that potentially many redundant propagations are performed, for instance when many clauses share the same literals. These literals are then repeatedly assigned to false and propagated. An attempt to reduce this redundant propagation effort was made in distillation by reorganizing the clauses in a trie (i.e., as a shared prefix tree) and reusing propagations along the same prefix [HS07]. The implementation of vivification in CaDiCaL [Bie17] and Kissat [BFFH20] achieves the same effect by sorting clauses and literals, thus simulating a trie structure on a plain CNF formula.

Another important question is which clauses should be checked, particularly if these techniques are applied to learned clauses during inprocessing [JHB12]. One proposal was to check learned clauses in parallel [WH13] in a separate thread. Another, which was actually the basis for the revival of these techniques [LLX+17], was to focus on learned clauses, check all clauses at most once, and check only those which have a high chance of being used (e.g., clauses of small LBD [AS09]). For more details see [LXL+20].

9.4. CNF Preprocessing Beyond Resolution

So far, we have only considered the elimination of clauses that are implied. However, as we discussed in the section on subsumption, in SAT solving often a more general notion of redundancy is used, requiring only that the elimination of a clause has no effect on the satisfiability status of a formula (see [JHB12]).

One of the most important types of clauses that are redundant but not necessarily implied, are *blocked clauses*. Intuitively, a clause is blocked if it contains a literal such that all resolvents upon this literal are tautologies [Kul99].

Definition 18. Given a formula F and a literal l, we denote by F_l the set of clauses of F that contain l. Then a clause C is a *blocked clause* in a formula F if it contains a literal l such that for every clause $D \in F_{\bar{l}}$, the resolvent of C and D upon l is a tautology. We say that l blocks C in F.

Example 19. Consider the formula $F = (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{x})$ and the clause $(x \vee y)$. The literal x does not block $(x \vee y)$ in F since the resolvent $(y \vee z)$ of $(x \vee y)$ and $(\bar{x} \vee z)$ is not a tautology. However, the literal y blocks $(x \vee y)$ in F since the only clause in $F_{\bar{y}}$ is the clause $(\bar{y} \vee \bar{x})$, and the resolvent $(x \vee \bar{x})$ of $(x \vee y)$ and $(\bar{y} \vee \bar{x})$ is a tautology. Therefore, $(x \vee y)$ is blocked in F.

Blocked clauses were initially introduced by Kullmann [Kul99] as a generalization of the definition clauses in extended resolution [Tse68], and were later proposed as a basis for the preprocessing technique of blocked clause elimination [JBH10] (BCE). As further shown in [JBH12], blocked clause elimination can simulate several structural simplification techniques (see also Section 9.6) on the CNF-level, in particular *non-shared-input elimination*, *monotone-input reduction*, and *cone-of-influence reduction*. Moreover, the effectiveness of bounded variable elimination can be increased by interleaving it with blocked clause elimination.

Blocked clauses have been generalized in several ways. One well-known generalization are *resolution asymmetric tautologies*, better known as RATs. The RAT definition is obtained from the blocked-clause definition by replacing *tautologies* with *asymmetric tautologies* [JHB12].

Definition 20. A clause C is a *resolution asymmetric tautology* (RAT) in a formula F if it contains a literal l such that for every clause $D \in F_{\bar{l}}$, the resolvent of C and D upon l is an asymmetric tautology with respect to F.

From this definition it immediately follows that every blocked clause in formula F is also a RAT with respect to F. Although the explicit elimination of RATs has not yet been shown to boost solver performance in SAT, generalizations of RAT elimination for quantified Boolean formulas (QBFs) have led to significant performance improvements in QBF solving [LE18].

Moreover, RAT lies at the core of the well-known DRAT proof system, which is the de-facto standard in modern SAT solving. Other generalizations of blocked clauses and RATs (e.g., *set-blocked clauses* [KSTB16, KSTB18] and *propagation-redundant clauses* (PR) [HKB17, HKB20]) have also not yet been used for clause elimination but instead for clause addition during solving and for proof generation, which we are not going to discuss here.

One type of clauses that generalizes blocked clauses, and whose elimination has shown some performance improvements in the SAT solver Lingeling [Bie17], are *covered clauses*. Intuitively, a clause is a covered if it can be turned into a blocked clause by adding so-called *covered literals* [HJB10b].

Definition 21. A literal k is a *covered literal* in a clause C with respect to a formula F if C contains a literal l such that all non-tautological resolvents of C upon l with clauses D in $F_{\bar{l}}$ contain k.

The addition of covered literals preserves satisfiability in the sense that $F \wedge C$ and $F \wedge (C \vee k)$ are equisatisfiable if k is covered by C in F; before an example, we give the definition of covered clauses [HJB10b] based on covered literals.

Definition 22. A clause C is a *covered clause* with respect to a formula F if there exists a sequence $k_1, \ldots, k_n$ of literals such that each k_i is covered by $C \vee k_1 \vee \cdots \vee k_{i-1}$ with respect to F and $C \vee k_1 \vee \cdots \vee k_n$ is blocked in F.

Example 23. Consider the formula $F = (\bar{x} \vee y) \wedge (\bar{y} \vee z)$ and the clause $C = (x \vee \bar{z})$. Although C is not blocked in F, we can add the literal y since it is contained in the only resolvent upon x, namely $(y \vee \bar{z})$, obtained by resolving with $(\bar{x} \vee y)$. The resulting clause $(x \vee \bar{z} \vee y)$ is then blocked as there is only the tautological resolvent $(x \vee \bar{z} \vee z)$ upon y, obtained by resolving with $(\bar{y} \vee z)$. Thus, C is a covered clause with respect to F.

There exist even more general types of redundant clauses, such as *resolution-subsumed clauses* [JHB12] ("the clause contains a literal such that all resolvents upon the literal are tautological or subsumed"), and asymmetric variants of blocked as well as covered clauses ("the addition of asymmetric or covered literals turns the clause into a blocked clause") [HJL+15, JHB12], thus extending *blocked clause elimination* (BCE) and *covered clause elimation* (CCE) to the more general concept of *asymmetric covered clause elimination* (ACCE).

It is, however, not clear if the elimination of such clauses can lead to noticeable performance improvements in practical SAT solving. The same applies to the elimination of *globally blocked clauses* [KHB19] motivated by circuit preprocessing techniques. There is also some renewed interest in covered clauses, which were recently shown to have orthogonal strength to propagation-redundant clauses [BCB20].

9.5. Solution Reconstruction

For many practical applications it is not sufficient to regard SAT as a mere decision problem. Consider, for instance, bounded model checking (Chapter 18): If a SAT solver determines that a propositional encoding of a safety property is satisfiable, we know that a bad state is reachable and thus that the safety property is violated. Without a concrete satisfying assignment, however, we cannot obtain an execution trace to analyze and diagnose the reason for the property violation. A similar situation occurs in SAT-based planning (Chapter 19), where a simple yes/no answer without the ability to generate actual plans is (mostly) useless.

Other applications require proofs after the SAT solver determined a formula to be unsatisfiable (e.g., to extract *clausal cores* or to generate *interpolants*). Preprocessors thus also need to make sure that valid proofs of a simplified formula can be turned into valid proofs of the original formula. For clausal proofs, which have been the dominant proof variant in the SAT competition since 2013, this is often easy to achieve [JHB12], and only requires that added clauses can be derived via a sequence of simple proof steps (for many techniques, a simple proof step via a RAT addition often suffices). Producing deletion information, used to speed up proof checking for DRAT proofs [HHJW13, WHHJ14], is slightly more involved, as additional care needs to be taken to avoid that clauses are deleted too early. See Chapter 15 on proofs of unsatisfiability for more details.

In the rest of this section, we focus on satisfiable formulas, and, following the literature [JB10], we use the terms *satisfying assignment*, *model*, and *solution* interchangeably. Moreover, we interpret *solution reconstruction* as the process of turning a satisfying assignment of the simplified formula (i.e., the formula produced by preprocessing) into a satisfying assignment of the original formula.

In order to obtain solution reconstruction for unit propagation (Section 9.2.1), it is common practice to simply keep unit literals permanently assigned. That is also the reason why they are sometimes called *root-level assigned*, *root-level forced* or simply *fixed* literals, where the *root level* is the top-most decision level (where no decisions have been made) in a CDCL solver (Chapter 4). This covers also the effect of failed-literal probing. Solvers might actually contain a global Boolean flag to record that the formula is *root-level inconsistent*, meaning that either the input CNF contains the empty clause or the empty clause has been derived (through preprocessing or in conflict analysis during CDCL search).

In industrial applications, it is not uncommon that preprocessing removes around 80% of the variables [EB05]. Focusing on the remaining "working set" of *active variables* can therefore improve cache efficiency and reduce memory consumption. Variables become inactive during search or inprocessing if they are for instance fixed (or eliminated). Accordingly, several SAT solvers distinguish

internal variables and *external variables* in order to keep the working set compact. External variables are those seen by the user of the (incremental) SAT solver and are mapped (and occasionally remapped during inprocessing) to the remaining internal variables on which the SAT solver works internally. Fixed external variables can either be recorded to be fixed (to a certain Boolean value) in this mapping directly, or can be mapped to a unique internal fixed variable (positively or negated), which represents all the fixed (external) variables.

All forms of strengthening and subsumption considered in this chapter, including vivification and distillation, preserve logical equivalence and thus can be ignored from the perspective of solution reconstruction (for satisfiable formulas). For pure literals solution reconstruction can be achieved by simply adding the corresponding unit clauses (which are RAT). However, as discussed in Section 9.2.3, this method does not preserve logical equivalence, since the formula might also allow models where pure literals are false. This is considered problematic for incremental SAT, since the user could be interested in such models in consecutive SAT queries. An alternative is to use a reconstruction stack as discussed further down, in essence treating pure literals in the same way as eliminated variables.

Equivalent-literal substitution (ELS) (see Section 9.3.2) also only preserves satisfiability, since substituted literals do not occur in the simplified formula anymore and can assume any value. One way of dealing with this situation, as implemented in Lingeling [Bie11], is to maintain a global union-find data structure [Tar79] of substituted literals which maps literals to the representative literal of their equivalent-literal class. The user can then obtain the solution value of a literal from its representative literal. This has the additional benefit that users can rather cheaply query the SAT solver, whether two literals have already been determined to be equivalent. This union-find data structure can also be shared among several solver threads efficiently as in Plingeling [Bie11]. Fixed literals (units) can be handled within the same scheme by adding a pseudo-representative constant literal (for, say, the Boolean constant *false*).

9.5.1. Reconstruction Stack

Bounded variable elimination, is on the one hand the most important preprocessing technique, but on the other hand does not preserve logical equivalence, by the same argument as for equivalent-literal substitution: after elimination, the eliminated variable is not restricted anymore and can assume any value. Therefore, we really need a method for solution reconstruction.

One way to perform solution reconstruction for variable elimination works as follows. Take the given solution of the simplified formula (restricted to the remaining variables) and add satisfied literals as unit clauses to the original formula. Alternatively, add back the eliminated clauses to the simplified formula plus the satisfied literals of the solution as units to obtain the augmented formula. Then call a second time a (plain) SAT solver on this augmented formula, which is guaranteed to be satisfiable, and that way compute a satisfying assignment of the original formula. The hope is that the additional unit clauses make it easy to solve the augmented formula, which is not guaranteed though. A similar scheme was employed in the SAT preprocessor SatELite [EB05], using temporary (bi-

nary) files for communicating the simplified CNF and the eliminated clauses to the second SAT solver.

A more sophisticated approach, guaranteed to have even linear complexity in the number of eliminated clauses, which was first described in the literature in [JB10], goes back to Niklas Sörensson. He observed that the given solution of the simplified formula can be "extended" to a solution of the original formula by propagating along the eliminated clauses in a specific way. The corresponding reconstruction algorithm can be described as follows.

Assume that whenever a variable is eliminated, the clauses in which the variable occurs are pushed on a stack, called the *reconstruction stack*. Further, make sure that the first literal in each pushed clause is either the eliminated variable or its negation, in order to map eliminated clauses to eliminated variables.

After obtaining a solution of the simplified formula, it is first extended by assigning an arbitrary value (e.g., false) to all eliminated variables. Then, solution reconstruction goes over the clauses on the reconstruction stack in reverse order, starting from the last pushed clause. Each clause is checked to be satisfied, and if not (i.e., all its literals are assigned to false), then the value of its first literal, which corresponds to the eliminated variable, is flipped (assigned opposite value).

Example 24. Consider the formula $(\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (b \vee c) \wedge (\bar{b} \vee \bar{c})$ which is satisfiable and encodes that a and b need to have the same truth value but b and c need to have opposite values. Now assume that variable elimination first eliminates a, then b, and during this process pushes these four clauses in the given order onto the reconstruction stack. The simplified formula consists of the empty CNF and has as extended solution the assignment where all variables are set to false. Now going backward over the reconstruction stack the algorithm first checks whether the last clause $(\bar{b} \vee \bar{c})$ is satisfied, which is the case. The next clause in reverse order $(b \vee c)$ is falsified however. Thus we flip the value of its first literal b and set b to true. The next clause to be checked $(a \vee \bar{b})$ is also falsified (since a is still false and b just became true). Accordingly, we flip the value of its first literal a and set a to true. The last clause to be checked, $(\bar{a} \vee b)$, is satisfied. Thus the reconstructed solution has both a and b set to true (assigned to the same value), while c remains false (assigned differently).

In fact, this simple algorithm also works for more advanced clause elimination procedures: Blocked clauses are just pushed on the reconstruction stack with the blocking literal as first *witness literal*. The same principle extends without modification to more powerful clause redundancy properties [HJB10a, HJB10b, HJL+15], including RAT, CCE and ACCE [JHB12], the latter two with the caveat of requiring to potentially push multiple clauses on the reconstruction stack when eliminating a clause [BCB20]. This scheme also allows to handle autarky elimination [BFFH20] as well as equivalent-literal substitution elegantly [FBS19], instead of using a union-find data structure. Finally, solution reconstruction for propagation-redundant clauses [HKB17, HKB20] can be achieved in a similar way, using a reconstruction stack, even for incremental SAT solving [FBS19]. The only change needed, as implemented in CaDiCaL to support globally blocked clauses [KHB19], is to replace the single witness literal by a set of witness literals, which are all set to true if the corresponding clause is falsified.

9.6. Structure-Based Preprocessing

So far we have focused on preprocessing techniques developed specifically for formulas in conjunctive normal form. We now extend the discussion to preprocessing techniques that were proposed for more expressive constraints and structural representations. In particular, we discuss parity-reasoning techniques and preprocessing techniques applicable on Boolean circuits, which offer a compact representation of arbitrary propositional formulas.

9.6.1. Parity Reasoning

Although the CNF representation of propositional formulas has many advantages, it can be suboptimal in some cases. One such case, which occurs frequently, is when formulas encode so-called *XOR constraints* or *equivalence clauses*. An XOR constraint is of the form

$$l_1 \oplus l_2 \oplus \ldots \oplus l_k = b,$$

where $l_1, \ldots, l_k$ are propositional literals and b is either 0 or 1. Following the standard semantics of the $\oplus$ operator ($x \oplus y$ is true if and only if x and y are assigned different truth values), an assignment satisfies an XOR constraint with $b = 1$ if and only if it satisfies an odd number of the literals $l_1, \ldots, l_k$. Respectively, it satisfies an XOR constraint with $b = 0$ if it satisfies an even number of the literals. Equivalence clauses are closely related to XOR constraints. An equivalence clause is of the form

$$l_1 \leftrightarrow l_2 \leftrightarrow \ldots \leftrightarrow l_k.$$

An assignment satisfies an equivalence clause if and only if it falsifies an even number of the literals $l_1, \ldots, l_k$. Observe that an equivalence clause of the form $l_1 \leftrightarrow \ldots \leftrightarrow l_{2i+1}$, having an odd number of literals, is equivalent to the XOR constraint $l_1 \oplus \ldots \oplus l_{2i+1} = 1$. Likewise, an equivalence clause $l_1 \leftrightarrow \ldots \leftrightarrow l_{2i}$ with an even number of literals is equivalent to the XOR constraint $l_1 \oplus \ldots \oplus l_{2i} = 0$. Thus equivalence clauses and XOR constraints are essentially the same concept.

An effective strategy for dealing with XOR constraints is to first scan the CNF encoding of a formula for sets of clauses that represent XOR constraints, and to then handle these constraints differently than ordinary clauses [Li00, HvM04, SNC09, Che09, Soo10, LJN10, LJN12a, LJN12b, Bie12, HJ12, SM19, SGM20].

The naive CNF encoding of an XOR constraint with k literals consists of 2^{k-1} clauses, where each clause rules out exactly one assignment that would falsify the constraint. More specifically, the CNF encoding of an XOR constraint $l_1 \oplus \ldots \oplus l_k = b$ consists of all k-sized clauses over $l_1, \ldots, l_k$, where an even number of literals occurs negated if $b = 1$ and an odd number occurs negated if $b = 0$.

Example 25. The XOR constraint

$$l_1 \oplus l_2 \oplus l_3 = 1$$

is falsified by every assignment that satisfies an even number of the literals l_1, l_2, l_3. Its CNF encoding thus rules out all such assignments, leading to the clauses $(l_1 \vee l_2 \vee l_3), (\bar{l}_1 \vee \bar{l}_2 \vee l_3), (l_1 \vee \bar{l}_2 \vee \bar{l}_3), (\bar{l}_1 \vee l_2 \vee \bar{l}_3)$.

A popular method for dealing with XOR constraints, which is used by several solvers, including CryptoMiniSat [SNC09, Soo10], Lingeling [Bie12], and March [HvM04], is *Gaussian elimination*. Gaussian elimination is based on the observation that XOR constraints can be viewed as equations over the finite field GF(2) (also denoted $\mathbb{F}_2$ or $\mathbb{Z}_2$), where the $\oplus$ operation corresponds to addition modulo 2 (and the $\wedge$ operation corresponds to multiplication modulo 2).

A set of XOR constraints can then be seen as a system of linear equations, and we can derive new equations by adding together existing ones, just as known from high school algebra. If Gaussian elimination is performed in a preprocessing phase, it can possibly derive new unit clauses.

Example 26. Consider the clauses

$$(\bar{x} \vee y \vee z), (x \vee \bar{y} \vee z), (x \vee y \vee \bar{z}), (\bar{x} \vee \bar{y} \vee \bar{z}),$$
$$(y \vee z), (\bar{y} \vee \bar{z}).$$

These clauses correspond to the two XOR constraints

$$x \oplus y \oplus z = 0, \tag{9.3}$$
$$y \oplus z = 1. \tag{9.4}$$

As there are no unit clauses, unit propagation would not have an effect on the clauses. However, by adding the XOR constraint 9.3 to 9.4, and by using the fact that $l \oplus l = 0$ and $l \oplus 0 = l$ for every literal l, we can derive the XOR constraint

$$x = 1, \tag{9.5}$$

which is equivalent to the unit clause (x).

Gaussian elimination can even show the unsatisfiability of particular formulas, as illustrated next.

Example 27. Consider the XOR constraints

$$x \oplus y = 1, \tag{9.6}$$
$$y \oplus z = 1, \tag{9.7}$$
$$x \oplus z = 1. \tag{9.8}$$

By adding 9.6 to 9.7, we derive

$$x \oplus z = 0. \tag{9.9}$$

If we add 9.8 to 9.9, we end up with the trivially unsatisfiable constraint $0 = 1$.

The use of Gaussian elimination can lead to exponential speedups in SAT solving. It has been shown that Tseitin formulas [Tse68] over expander graphs have only exponential-size resolution proofs [Urq87], meaning that CDCL solvers require exponential time to solve them. In contrast, Gaussian elimination can show the unsatisfiability of these formulas in polynomial time.

Moreover, Gaussian elimination can be expressed polynomially with extended resolution [SB06] and thus in the DRAT proof system [PR16]. It is still unclear though how many new variables are needed. For simple problems which require parity reasoning new variables can be avoided by reusing eliminated variables [CH20]. Note that Gaussian elimination can be seen as an instance of the more general approach of *algebraic* SAT solving in Chapter 7 on proof complexity.

There exist several approaches that extract and utilize equivalence clauses, especially in preprocessing. Already in 1998, Warners and van Maaren [WvM98] presented a preprocessor that utilizes linear programming to extract equivalence clauses from CNF formulas. By reasoning over these equivalence clauses in an initial solving phase and then running a basic DPLL procedure, their approach was the first to solve the notorious *parity formula* [CKS94] within reasonable time. Heule and van Maaren [HvM04] later improved this approach and integrated equivalence reasoning into the look-ahead solver March (see Chapter 5 for details on look-ahead based solvers), which could achieve a considerable speedup by avoiding look-ups on variables for which look-ups on equivalent literals (as indicated via binary equivalences) had already been performed.

Another approach that relies on inference rules over equivalence clauses was presented by Li [Li00]. In an inprocessing fashion, he incorporated these inference rules—which apply to equivalence clauses with at most three literals—into the solver Satz. The resulting solver, EqSatz, was able to outperform the approach of Warners and van Maaren on some, but not all, of the parity formulas.

The current state of the art in parity reasoning, applied to approximate model counting, focuses on parity reasoning during search [SM19, SGM20], where much more efficient algorithms are required, extending ideas from [HJ12].

9.6.2. Circuit-Level Reasoning

Propositional encodings tend to be more natural to develop using the full language of propositional logic rather than aiming directly at a CNF-level encoding. Boolean circuits offer a succinct structural representation for propositional formulas. Succinctness comes from refining syntax trees of propositional formulas to directed acyclic graph structures by allowing *structural hashing*, i.e., by representing each subcircuit (subformula) only once. This is a common implementation technique for representing immutable data structures, also called *hash consing* in the context of functional languages; it is used in most BDD libraries (in form of a "unique table") and related to *common subexpression elimination* in optimizing compilers. In practical implementations of circuit-level reasoning, restricted classes of circuits are used, with *and-inverter graphs* (AIGs) [KGP01] as one prominent example.

While SAT solvers working directly on the level of circuits have been developed (see, for instance, [Lar92] and in general Chapter 27), SAT solvers today tend to be implemented specifically for CNF formulas. In fact, standard Boolean constraint propagation achieved on the level of circuits is equivalently achieved by unit propagation on the CNF obtained via the Tseitin encoding [Tse68].

However, circuits offer a view to developing further structure-based CNF encodings and simplification techniques, by exploiting functional dependencies

explicit in circuits. A classical argument for the necessity of such structural information and against CNF is that this information is lost in the CNF encoding phase. While structural information is not evident in CNF, it has turned out that such arguments are partly wrong. In particular, CNF-level reasoning can achieve the same effects for various forms of circuit-level reasoning techniques.

One example is the Plaisted-Greenbaum CNF encoding of propositional formulas [PG86] which, based on the notion of polarities of subformulas, refines the Tseitin encoding [Tse68]. It allows to drop clauses representing one direction of the bi-implications used in the Tseitin encoding, thus locally encoding a unipolar subformula. While subformula polarities are computed following the circuit structure of a formula, it has been shown [JBH12] that blocked clause elimination, working "blindly" on the CNF, can remove all clauses left out by the Plaisted-Greenbaum encoding. This implies that blocked clause elimination from Section 9.4 achieves monotone-input reduction, i.e., it detects if the underlying circuit of the formula is monotone in terms of a particular input variable (input gate or primary input). As blocked clause elimination can be implemented efficiently, this renders the Plaisted-Greenbaum encoding on its own unnecessary.

Evidently, there are limits to what can be achieved (both in theory and in practice) in terms of structure-based simplifications solely on the CNF-level, but these limits are to an extent unclear. Structural hashing, for example, can be simulated by hyper binary resolution of Section 9.3.3 on the Tseitin encoding of AIGs, as shown in [HJB13]. However, for other gate types such as XORs and *if-then-else*, also called *multiplexer* (MUX), hyper binary resolution does not achieve structural hashing [HJB13].

More general hashing for minimizing circuit representations through detection of logically equivalent substructures can be achieved on the circuit level through BDD sweeping [KK97] and SAT sweeping [Kue04, MCJB05, ZKKS06, WKK12]. Stålmarck's procedure [SS00] and Recursive Learning [KP92] are earlier circuit-level preprocessing techniques with similar effects.

There are CNF-level versions of Stålmarck's procedure [GW00], Recursive Learning [MSG99] or more general SAT sweeping [HJB13, JLaMS15, CFM13]. These procedures allow to deduce literal equivalences implied by the CNF as well as literals forced to a specific truth value, because they have that same truth value in all models (so-called "backbones" [JLaMS15]). In contrast to hyper binary resolution, which also produces equivalences, sweeping is able to find more of these equivalences or even all, if required by some applications. These equivalences are typically used to reduce the CNF as described in Section 9.3.2.

The current state of the art in encoding circuits into CNF [EMS07] is based on techniques from circuit synthesis: *circuit-level rewriting* [BB04, BB06] with *cut enumeration* [MCB06a] and *technology mapping* [MCB06b]. Related work uses similar criteria [MV07, CMV09]. We are not aware of any CNF-level preprocessing technique able to simulate these optimizations.

When having to start with CNF, one approach to exploit circuit-level techniques is to first attempt to algorithmically recover a circuit-level representation potentially underlying the CNF formula, on which circuit-level techniques can then be applied. This process is also called *gate extraction* or *mining functional definitions*. Even though there is already a substantial amount of work

in this direction starting with [OGMS02, RMB04, FM07, IKS17] and most recently [LLM20, Sli20] with more references, the task of the recovery step is in general non-trivial and rarely used in practical SAT solving.

There is a related technique, called blocked clause decomposition [HB13, BFHB14, Che15], with a similar purpose: it partitions a given CNF into a set of blocked clauses (which is maximized) and some remaining clauses. The set of blocked clauses has similar properties as a circuit, i.e., clauses are ordered and models can be constructed polynomially, which has applications usually attributed to circuit-level reasoning, including SAT sweeping [HB13].

Beside circuits and XOR constraints already discussed in Section 9.6.1, there are other high-level representations or constraints for which dedicated preprocessing algorithms exist, but currently no (efficient) CNF-level technique is known. As explained for parity reasoning, a common approach is to extract such constraints, apply high-level preprocessing, and then encode the resulting constraints back to CNF. Closely related to XOR constraints are polynomials modulo 2 as considered in [CK07, CSCM19], and using Gröbner bases theory for preprocessing and inprocessing. Regarding pseudo-Boolean and cardinality constraints (see also Chapter 28) a similar argument applies [BLBLM14].

9.7. Conclusion

The development of novel types of preprocessing techniques and the optimized implementation of these techniques has made preprocessing a central part of the SAT solving workflow. In this chapter, we have discussed major developments in preprocessing in SAT, focusing mostly on CNF-level preprocessing techniques, but also considering some techniques on other representations.

In practice, the role of preprocessing is intertwined on one hand with the encoding phase, providing an automated way to re-encode CNF instances, and on the other hand with CDCL SAT solving. Developments in the latter direction have brought on the influential inprocessing SAT solving paradigm, which interleaves core CDCL search with complex combinations of preprocessing techniques.

Arguably, bounded variable elimination [EB05] is still the most important practical preprocessing technique today. However, the many other types of preprocessing techniques, developed in particular during the 21$^{\text{st}}$ century, play an evident role as well. In terms of supporting variable elimination, various other techniques are important for triggering further elimination steps of bounded variable elimination, by performing crucial redundancy-elimination steps that are not enabled by bounded variable elimination alone. Other individual techniques and their combinations can be central in different types of problem-specific applications of SAT solvers.

Going further, the various developments in CNF-level preprocessing have had a wide impact, well beyond mere preprocessing. By making use of polynomial-time checkable clause-redundancy properties, inprocessing SAT solving breaks the long-standing "resolution barrier" of standard CDCL SAT solvers, making the solvers more powerful than the resolution proof system. Clause redundancy properties have also resulted in generally applicable practical proof checking mechanisms, enabling the automated verification of proofs constructed by SAT solvers,

thus lifting the trustworthiness of "no" answers provided by SAT solvers to new levels (see Chapter 15).

There is much ongoing work on combining high-level and CNF-level preprocessing, particularly for non-linear algebraic reasoning [CSCM19, KBK19], while producing proofs for even parity reasoning is still a challenge [CH20]. In this context, revisiting techniques using binary decision diagrams might also be interesting [MM02, FKS+04, SB06, WFS06, BD09, vDEB18]. We also consider parallelization of preprocessing to still be in its infancy [WH13, GM13, HW13, BS18, OW19].

Finally, we want to point out that many of the preprocessing techniques discussed in this chapter have been lifted to more general logics and problems, such as QBF, MaxSAT, #SAT, and first-order logic, but instead of listing these extensions here, we refer to the chapters on these extensions in this handbook or the corresponding literature instead.

Acknowledgments

We would like to thank Lee Barnett, Katalin Fazekas, and Mathias Fleury for providing valuable feedback on late drafts of the chapter.

References

[ABS99] G. Audemard, B. Benhamou, and P. Siegel. La méthode d'avalanche AVAL: Une méthode énumérative pour SAT. *Journées Nationales de la Résolution Pratique des problèmes NP-Complets (JNPC)*, pages 17–25, 1999.

[ALS13] G. Audemard, J. Lagniez, and L. Simon. Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In M. Järvisalo and A. Van Gelder, editors, *SAT*, volume 7962 of *LNCS*, pages 309–317. Springer, 2013.

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.

[AS09] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In C. Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.

[Bac02] F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In R. Dechter, M. J. Kearns, and R. S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 613–619. AAAI Press / The MIT Press, 2002.

[BB04] P. Bjesse and A. Borälv. DAG-aware circuit compression for formal verification. In *2004 International Conference on Computer-Aided Design, ICCAD 2004, San Jose, CA, USA, November 7-11, 2004*, pages 42–49. IEEE Computer Society / ACM, 2004.

[BB06] R. Brummayer and A. Biere. Local two-level And-Inverter graph minimization without blowup. In *Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2006)*, 2006.

[BBIS16] T. Balyo, A. Biere, M. Iser, and C. Sinz. SAT Race 2015. *Artificial Intelligence*, 241:45–65, 2016.

[BCB20] L. A. Barnett, D. Cerna, and A. Biere. Covered clauses are not propagation redundant. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2020.

[BD09] M. Brickenstein and A. Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *J. Symb. Comput.*, 44(9):1326–1345, 2009.

[BDP03] F. Bacchus, S. Dalmao, and T. Pitassi. DPLL with caching: A new algorithm for #SAT and Bayesian inference. *Electronic Colloquium on Computational Complexity (ECCC)*, 10(003), 2003.

[BFFH20] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[BFHB14] T. Balyo, A. Fröhlich, M. J. H. Heule, and A. Biere. Everything you always wanted to know about blocked sets (but were afraid to ask). In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2014.

[Bie03] A. Biere. About the SAT solvers Limmat, Compsat, Funex and the QBF solver Quantor, 2003. Presentation for the SAT'03 SAT Solver Competition.

[Bie04] A. Biere. Resolve and expand. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2004.

[Bie09] A. Biere. P{re,i}coSAT@SC'09. In *SAT 2009 Competitive Event Booklet*, 2009.

[Bie10] A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Johannes Kepler University Linz, 2010.

[Bie11] A. Biere. Lingeling and friends at the SAT Competition 2011. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040

Linz, Austria, 2011.

[Bie12] A. Biere. Lingeling and friends entering the SAT Challenge 2012. In *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Dept. of Computer Science Series of Publications B, University of Helsinki*, 2012.

[Bie14] A. Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver Lingeling. In D. Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, volume 27 of *EPiC Series in Computing*, page 88. EasyChair, 2014.

[Bie16] A. Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT Competition 2016. In T. Balyo, M. J. H. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2016: Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 44–45. University of Helsinki, 2016.

[Bie17] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT Competition 2017. In T. Balyo, M. J. H. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2017: Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[Bie18] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2018. In M. J. H. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*, pages 13–14. University of Helsinki, 2018.

[Bie19] A. Biere. CaDiCaL at the SAT Race 2019. In M. J. H. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Race 2019: Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.

[BKS04] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.

[BLBLM14] A. Biere, D. Le Berre, E. Lonca, and N. Manthey. Detecting cardinality constraints in CNF. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 285–301. Springer, 2014.

[Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BP11] R. J. Bayardo and B. Panda. Fast algorithms for finding extremal

sets. In *Proceedings of the Eleventh SIAM International Conference on Data Mining, SDM 2011, April 28-30, 2011, Mesa, Arizona, USA*, pages 25–34. SIAM / Omnipress, 2011.

[Bra04] R. I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 34(1):52–59, 2004.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS06] A. Biere and C. Sinz. Decomposing SAT problems into connected components. *J. Satisf. Boolean Model. Comput.*, 2(1-4):201–208, 2006.

[BS18] T. Balyo and C. Sinz. Parallel satisfiability. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer, 2018.

[BW03] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In E. Giunchiglia and A. 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*, pages 341–355. Springer, 2003.

[CFM13] M. Codish, Y. Fekete, and A. Metodi. Backbones for equality. In V. Bertacco and A. Legay, editors, *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, volume 8244 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2013.

[CH20] L. Chew and M. J. H. Heule. Sorting parity encodings by reusing variables. In L. Pulina and M. Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2020.

[Che09] J. Chen. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In O. 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*, pages 298–311. Springer, 2009.

[Che15] J. Chen. Fast blocked clause decomposition with high quality. *CoRR*, abs/1507.00459, 2015.

[CK07] C. Condrat and P. Kalla. A Gröbner basis approach to CNF-formulae preprocessing. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.

[CKS94] J. M. Crawford, M. J. Kearns, and R. E. Schapire. The minimal disagreement parity problem as a hard satisfiability problem. Technical report, Computational Intelligence Research Laboratory and AT&T Bell Labs, 1994.

[CMV09] B. Chambers, P. Manolios, and D. Vroon. Faster SAT solving with better CNF generation. In L. Benini, G. De Micheli, B. M. Al-Hashimi, and W. Müller, editors, *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 1590–1595. IEEE, 2009.

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

[CS00] P. Chatalic and L. Simon. ZRES: The old Davis-Putnam procedure meets ZBDD. In D. A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*, pages 449–454. Springer, 2000.

[CSCM19] D. Choo, M. Soos, K. M. A. Chai, and K. S. Meel. Bosphorus: Bridging ANF and CNF solvers. In J. Teich and F. Fummi, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 468–473. IEEE, 2019.

[DFS59] B. Dunham, R. Fridshal, and G. L. Sward. A non-heuristic program for proving elementary logical theorems. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959*, pages 282–284. UNESCO (Paris), 1959.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DM94] M. D'Agostino and M. Mondadori. The taming of the cut. classical refutations with analytic cut. *J. Log. Comput.*, 4(3):285–319, 1994.

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

[dV00] A. del Val. On 2-SAT and renamable Horn. In H. A. Kautz and B. W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, pages 279–284. AAAI Press / The MIT Press, 2000.

[dV01] A. del Val. Simplifying binary propositional theories into connected components twice as fast. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2001.

[EB05] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International*

Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[EMS07] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In J. a. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.

[ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[ES06] N. Eén and N. Sörensson. MiniSat 2.0 beta. In *Solver Description SAT Race 2006*, 2006.

[FBS19] K. Fazekas, A. Biere, and C. Scholl. Incremental inprocessing in SAT solving. In M. Janota and I. 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 136–154. Springer, 2019.

[FKS+04] J. V. Franco, M. Kouril, J. S. Schlipf, S. A. Weaver, M. R. Dransfield, and W. M. Vanfleet. Function-complete lookahead in support of efficient SAT search heuristics. *J. Univers. Comput. Sci.*, 10(12):1655–1695, 2004.

[FM07] Z. Fu and S. Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *20th International Conference on VLSI Design (VLSI Design 2007), Sixth International Conference on Embedded Systems (ICES 2007), 6-10 January 2007, Bangalore, India*, pages 37–42. IEEE Computer Society, 2007.

[Fre95] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, PA, USA, 1995.

[GM13] K. Gebhardt and N. Manthey. Parallel variable elimination on CNF formulas. In I. J. Timm and M. Thimm, editors, *KI 2013: Advances in Artificial Intelligence - 36th Annual German Conference on AI, Koblenz, Germany, September 16-20, 2013. Proceedings*, volume 8077 of *Lecture Notes in Computer Science*, pages 61–73. Springer, 2013.

[GS05] R. Gershman and O. Strichman. Cost-effective hyper-resolution for preprocessing CNF formulas. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 423–429. Springer, 2005.

[GW00] J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. *J. Autom. Reasoning*, 24(1/2):101–125, 2000.

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

[HB13] M. J. H. Heule and A. Biere. Blocked clause decomposition. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2013.

[HHJW13] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.

[HJ12] C. Han and J. R. Jiang. When Boolean satisfiability meets Gaussian elimination in a Simplex way. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 410–426. Springer, 2012.

[HJB10a] M. J. H. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for CNF formulas. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2010.

[HJB10b] M. J. H. Heule, M. Järvisalo, and A. Biere. Covered clause elimination. In A. Voronkov, G. Sutcliffe, M. Baaz, and C. G. Fermüller, editors, *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short, Yogyakarta, Indonesia, October 10-15, 2010*, volume 13 of *EPiC Series in Computing*, pages 41–46. EasyChair, 2010.

[HJB11] M. J. H. Heule, M. Järvisalo, and A. Biere. Efficient CNF simplification based on binary implication graphs. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2011.

[HJB13] M. J. H. Heule, M. Järvisalo, and A. Biere. Revisiting hyper binary resolution. In C. P. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2013.

[HJL+15] M. J. H. Heule, M. Järvisalo, F. Lonsing, M. Seidl, and A. Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53:127–168, 2015.

[HJN10] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intel-*

ligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings, volume 6397 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2010.

[HJS10] Y. Hamadi, S. Jabbour, and L. Sais. Learning for dynamic subsumption. *Int. J. Artif. Intell. Tools*, 19(4):511–529, 2010.

[HJS11] H. Han, H. Jin, and F. Somenzi. Clause simplification through dominator analysis. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 143–148. IEEE, 2011.

[HKB17] M. J. H. Heule, B. Kiesl, and A. Biere. Short proofs without new variables. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2017.

[HKB20] M. J. H. Heule, B. Kiesl, and A. Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020.

[Hoo93] J. N. Hooker. Solving the incremental satisfiability problem. *J. Log. Program.*, 15(1&2):177–186, 1993.

[HS07] H. Han and F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 582–587. IEEE, 2007.

[HS09] H. Han and F. Somenzi. On-the-fly clause improvement. In O. 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*, pages 209–222. Springer, 2009.

[HvM04] M. J. H. Heule and H. van Maaren. Aligning CNF- and equivalence-reasoning. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2004.

[HW13] Y. Hamadi and C. M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Mag.*, 34(2):99–106, 2013.

[IKS17] M. Iser, F. Kutzner, and C. Sinz. Using gate recognition and random simulation for under-approximation and optimized branching in SAT solvers. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 1029–1036. IEEE Computer Society, 2017.

[JB10] M. Järvisalo and A. Biere. Reconstructing solutions after blocked clause elimination. In O. Strichman and S. Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 340–345. Springer, 2010.

[JBH10] M. Järvisalo, A. Biere, and M. J. H. Heule. Blocked clause elimination. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.

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

[JHB12] M. Järvisalo, M. J. H. Heule, and A. Biere. Inprocessing rules. In B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.

[JK14] M. Järvisalo and J. H. Korhonen. Conditional lower bounds for failed literals and related techniques. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 75–84. Springer, 2014.

[JLaMS15] M. Janota, I. Lynce, and J. ao Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28(2):161–177, 2015.

[KBK19] D. Kaufmann, A. Biere, and M. Kauers. Verifying large multipliers by combining SAT and computer algebra. In C. W. Barrett and J. Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 28–36. IEEE, 2019.

[KGP01] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 232–237. ACM, 2001.

[KHB19] B. Kiesl, M. J. H. Heule, and A. Biere. Truth assignments as conditional autarkies. In Y. Chen, C. Cheng, and J. Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2019.

[KK97] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In E. J. Yoffa, G. De Micheli, and J. M. Rabaey, editors, *Proceedings of the 34st Conference on Design Automation, Anaheim, California, USA, Anaheim Convention Center, June 9-13, 1997.*, pages 263–268. ACM Press, 1997.

[KN11] Z. Khasidashvili and A. Nadel. Implicative simultaneous satisfiability and applications. In K. Eder, J. ao Lourenço, and O. Shehory, editors, *Hardware and Software: Verification and Testing - 7th In-*

ternational Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers, volume 7261 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2011.

[KNPH05] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna. Simultaneous SAT-based model checking of safety properties. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, volume 3875 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2005.

[Knu15] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 2015.

[Kor08] K. Korovin. iProver — an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[KP92] W. Kunz and D. K. Pradhan. Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. In *Proceedings IEEE International Test Conference 1992, Discover the New World of Test and Design, Baltimore, Maryland, USA, September 20-24, 1992*, pages 816–825. IEEE Computer Society, 1992.

[KSS+17] B. Kiesl, M. Suda, M. Seidl, H. Tompits, and A. Biere. Blocked clauses in first-order logic. In T. Eiter and D. Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 31–48. EasyChair, 2017.

[KSTB16] B. Kiesl, M. Seidl, H. Tompits, and A. Biere. Super-blocked clauses. In N. Olivetti and A. Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2016.

[KSTB18] B. Kiesl, M. Seidl, H. Tompits, and A. Biere. Local redundancy in SAT: Generalizations of blocked clauses. *Log. Methods Comput. Sci.*, 14(4), 2018.

[Kue04] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *2004 International Conference on Computer-Aided Design, ICCAD 2004, San Jose, CA, USA, November 7-11, 2004*, pages 50–57. IEEE Computer Society / ACM, 2004.

[Kul99] O. Kullmann. On a generalization of extended resolution. *Discret. Appl. Math.*, 96-97:149–176, 1999.

[KWS00] J. Kim, J. Whittemore, and K. A. Sakallah. On solving stack-based incremental satisfiability problems. In *Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors, ICCD '00, Austin, Texas, USA, September 17-20, 2000*, pages 379–382. IEEE Computer Society, 2000.

[LaPMS03] I. Lynce and J. ao P. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2003), 3-5 November 2003, Sacramento, California, USA*, pages 105–110. IEEE Computer Society, 2003.

[Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 11(1):4–15, 1992.

[LB01] D. Le Berre. Exploiting the real power of unit propagation lookahead. *Electron. Notes Discret. Math.*, 9:59–80, 2001.

[LE18] F. Lonsing and U. Egly. QRAT+: Generalizing QRAT by a more powerful QBF redundancy property. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2018.

[LGPC16] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016.

[Li00] C. M. Li. Integrating equivalency reasoning into Davis-Putnam procedure. In H. A. Kautz and B. W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, pages 291–296. AAAI Press / The MIT Press, 2000.

[LJN10] T. Laitinen, T. A. Junttila, and I. Niemelä. Extending clause learning DPLL with parity reasoning. In H. Coelho, R. Studer, and M. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 21–26. IOS Press, 2010.

[LJN12a] T. Laitinen, T. A. Junttila, and I. Niemelä. Classifying and propagating parity constraints. In M. Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2012.

[LJN12b] T. Laitinen, T. A. Junttila, and I. Niemelä. Extending clause learning SAT solvers with complete parity reasoning. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 65–72. IEEE Computer Society, 2012.

[LLM20] J. Lagniez, E. Lonca, and P. Marquis. Definability for model count-

ing. *Artif. Intell.*, 281:103229, 2020.

[LLX+17] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 703–711. ijcai.org, 2017.

[LXL+20] C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.*, 279, 2020.

[Man12] N. Manthey. Coprocessor 2.0 - A flexible CNF simplifier - (tool presentation). In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 2012.

[MCB06a] A. Mishchenko, S. Chatterjee, and R. K. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In E. Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 532–535. ACM, 2006.

[MCB06b] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Improvements to technology mapping for LUT-based FPGAs. In S. J. E. Wilton and A. DeHon, editors, *Proceedings of the ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays, FPGA 2006, Monterey, California, USA, February 22-24, 2006*, pages 41–49. ACM, 2006.

[MCJB05] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, ERL Technical Report, 2005.

[MHB12] N. Manthey, M. J. H. Heule, and A. Biere. Automated reencoding of Boolean formulas. In A. Biere, A. Nahir, and T. E. J. Vos, editors, *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012.

[Min92] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange (SASIMI'92)*, pages 64–73, 1992.

[Min93] S. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC*, pages 272–277. ACM Press, 1993.

[MM02] D. B. Motter and I. L. Markov. A compressed breadth-first search for satisfiability. In D. M. Mount and C. Stein, editors, *Algorithm Engineering and Experiments, 4th International Workshop, ALENEX 2002, San Francisco, CA, USA, January 4-5, 2002, Revised Papers*, volume 2409 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2002.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[MSG99] J. a. P. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *1999 Design, Automation and Test in Europe (DATE '99), 9-12 March 1999, Munich, Germany*, pages 145–149. IEEE Computer Society / ACM, 1999.

[MV07] P. Manolios and D. Vroon. Efficient circuit to CNF conversion. In J. ao Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 4–9. Springer, 2007.

[NII15] H. Nabeshima, K. Iwanuma, and K. Inoue. GlueMiniSat 2.2.10 & 2.2.10-5, 2015. SAT-Race 2015.

[NR12] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012.

[NR18] A. Nadel and V. Ryvchin. Chronological backtracking. In O. Beyersdorff and C. M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018.

[NRS12] A. Nadel, V. Ryvchin, and O. Strichman. Preprocessing in incremental SAT. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 256–269. Springer, 2012.

[NRS14] A. Nadel, V. Ryvchin, and O. Strichman. Ultimately incremental SAT. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2014.

[OGMS02] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from CNF formulas. In P. V. Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.

[Oh16] C. Oh. *Improving SAT solvers by exploiting empirical characteristics of CDCL*. PhD thesis, New York University, NY, USA, 2016.

[OW19] M. Osama and A. Wijs. Parallel SAT simplification on GPU architectures. In *TACAS (1)*, volume 11427 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2019.

[PD07] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. ao Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[PD11] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[PHS08] C. Piette, Y. Hamadi, and L. Sais. Vivifying propositional clausal formulae. In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529. IOS Press, 2008.

[PR16] T. Philipp and A. Rebola-Pardo. DRAT proofs for XOR reasoning. In L. Michael and A. C. Kakas, editors, *Logics in Artificial Intelligence - 15th European Conference, JELIA 2016, Larnaca, Cyprus, November 9-11, 2016, Proceedings*, volume 10021 of *Lecture Notes in Computer Science*, pages 415–429, 2016.

[RMB04] J. A. Roy, I. L. Markov, and V. Bertacco. Restoring circuit structure from SAT instances. In *Proceedings of International Workshop on Logic and Synthesis (IWLS)*, pages 663–678, 2004.

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

[Rob74] J. A. Robinson. Automatic deduction with hyper-resolution. *J. Symb. Log*, 39(1):189–190, 1974.

[SAFS95] J. S. Schlipf, F. S. Annexstein, J. V. Franco, and R. P. Swaminathan. On finding solutions for extended Horn formulas. *Inf. Process. Lett.*, 54(3):133–137, 1995.

[SB06] C. Sinz and A. Biere. Extended resolution proofs for conjoining BDDs. In D. Grigoriev, J. Harrison, and E. A. Hirsch, editors, *Computer Science - Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.

[SB09] N. Sörensson and A. Biere. Minimizing learned clauses. In O. 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*, pages 237–243. Springer, 2009.

[SBB+04] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Com-

bining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[SCD+20] M. Soos, S. Cai, J. Devriendt, S. Gocht, A. Shaw, and K. Meel. CryptoMiniSAT with CCAnr at the SAT Competition 2020. In T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 27–28. University of Helsinki, 2020.

[SGM20] M. Soos, S. Gocht, and K. S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 463–484. Springer, 2020.

[Sli20] F. Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 508–528. Springer, 2020.

[SM19] M. Soos and K. S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 1592–1599. AAAI Press, 2019.

[SNC09] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In O. 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*, pages 244–257. Springer, 2009.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[Soo10] M. Soos. Enhanced Gaussian elimination in DPLL-based SAT solvers. In D. Le Berre, editor, *POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010*, volume 8 of *EPiC Series in Computing*, pages 2–14. EasyChair, 2010.

[SP04] S. Subbarayan and D. K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[SS00] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods Syst. Des.*, 16(1):23–58, 2000.

[SSK+20] M. Soos, B. Selman, H. Kautz, J. Devriendt, and S. Gocht. CryptoMiniSAT with WalkSAT at the SAT Competition 2020. In T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 29–30. University of Helsinki, 2020.

[Tar79] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979.

[Tse68] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Mathematics and Mathematical Logic*, 2:115–125, 1968.

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

[vDEB18] T. van Dijk, R. Ehlers, and A. Biere. Revisiting decision diagrams for SAT. *CoRR*, abs/1805.03496, 2018.

[VG05] A. Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Ann. Math. Artif. Intell.*, 43(1):239–253, 2005.

[VG12] A. 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.

[VGT93] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 559–586. DIMACS/AMS, 1993.

[WFS06] S. A. Weaver, J. V. Franco, and J. S. Schlipf. Extending existential quantification in conjunctions of BDDs. *J. Satisf. Boolean Model. Comput.*, 1(2):89–110, 2006.

[WH13] S. Wieringa and K. Heljanko. Concurrent clause strengthening. In M. Järvisalo and A. Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2013.

[WHHJ14] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.

[WKK12] T. Welp, S. Krishnaswamy, and A. Kuehlmann. Generalized SAT-sweeping for post-mapping optimization. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation*

Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012, pages 814–819. ACM, 2012.

[WKS01] J. Whittemore, J. Kim, and K. A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, pages 542–545. ACM, 2001.

[WvdGS13] A. Wotzlaw, A. van der Grinten, and E. Speckenmeyer. Effectiveness of pre- and inprocessing for CDCL-based SAT solving. *CoRR*, abs/1310.4756, 2013.

[WvM98] J. P. Warners and H. van Maaren. A two-phase algorithm for solving a class of hard satisfiability problems. *Oper. Res. Lett.*, 23(3-5):81–88, 1998.

[ZC20] X. Zhang and S. Cai. Relaxed backtracking with rephasing. In T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 15–16. University of Helsinki, 2020.

[Zha05] L. Zhang. On subsumption removal and on-the-fly CNF simplification. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 482–489. Springer, 2005.

[ZKKS06] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. L. Sangiovanni-Vincentelli. SAT sweeping with local observability don't-cares. In E. Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 229–234. ACM, 2006.

[ZM88] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In H. E. Shrobe, T. M. Mitchell, and R. G. Smith, editors, *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988.*, pages 155–160. AAAI Press / The MIT Press, 1988.

[ZS00] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *J. Autom. Reason.*, 24(1/2):277–296, 2000.