Expressing Symmetry Breaking in DRAT Proofs

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

The University of Texas at Austin

Abstract. An effective SAT preprocessing technique is using symmetry-breaking predicates, i.e., auxiliary clauses that guide a solver away from needless exploration of isomorphic sub-problems. Although symmetry-breaking predicates have been in use for over a decade, it was not known how to express them in proofs of unsatisfiability. Consequently, results obtained by symmetry breaking cannot be validated by existing proof checkers. We present a method to express symmetry-breaking predicates in DRAT, a proof format that is supported by the top-tier solvers. We applied this method to generate proofs of problems that have not been solved without symmetry-breaking predicates. We validated these proofs with both an ACL2-based, mechanically-verified checker and the proof-checking tool of SAT Competition 2014.

1 Introduction

Satisfiability (SAT) solvers can be applied to decide hard combinatorial problems that contain symmetries. Breaking problem symmetries typically boosts solver performance as it prevents a solver from needlessly exploring isomorphic parts of the search space. A common method to eliminate symmetries is to add *symmetry-breaking predicates* [1,2,3]. However, expressing symmetry-breaking predicates in existing SAT proof formats has been an open problem, leaving it hard to validate solver results. We present a method to express the use of symmetry-breaking predicates in the DRAT proof format [4], which is supported by the top-tier SAT solvers and was used to validate the results of SAT Competition 2014.

Recent successes of SAT technology include solving several long-standing open problems such as the Erdős Discrepancy Conjecture [5], computing Van der Waerden numbers [6], and producing minimal sorting networks [7]. Symmetry-breaking techniques can be applied to each of these problems and allows one to solve them more efficiently. Our new method facilitates creating proofs for unsatisfiability results even when symmetry-breaking techniques are applied.

A state-of-the-art tool to break symmetries in SAT problems is `shatter` [8]; it performs *static symmetry-breaking* by adding symmetry-breaking predicates to the problem. The `shatter` tool works as follows: a SAT problem is converted into a graph and all symmetry groups are computed. Symmetry groups are then transformed into predicates and added to the SAT problem. An alternative method, called *dynamic symmetry-breaking* [9], adds symmetric versions

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

of learned clauses to the problem. Dynamic symmetry-breaking is most useful when few symmetries exist, which is the case for graph-coloring problems.

We will demonstrate the expression of symmetry-breaking predicates in the DRAT proof format using unavoidable subgraphs. An *unavoidable subgraph* G, for graphs of order n, is an undirected graph such that any red/blue edge-coloring of the fully-connected, undirected graph of order n contains G in either red or blue. The most famous type of unavoidable subgraphs are cliques (Ramsey numbers), but there are many other types of graphs for which unavoidability has been studied; an online dynamic survey by Stanisław Radziszowski [10] lists over 600 articles on the topic. SAT solvers have severe difficulty solving unavoidable graph problems without symmetry-breaking predicates, but become more powerful tools with them.

Given a satisfiability problem F, we produce a satisfiability-equivalent problem F' that contains symmetry-breaking predicates. The formula F' is similar to the result of applying `shatter` (modulo variable renaming). Additionally, we produce a partial DRAT proof that expresses the conversion from F to F'. We solve F' using an off-the-shelf SAT solver that can emit DRAT proofs — which is the case for all top-tier solvers. Finally, we validate the result by merging the partial proof with the SAT solver proof and applying a checker such as `drat-trim` [4]. We evaluate this method on some hard combinatorial problems.

The remainder of the paper is structured as follows. After some preliminary and background in Section 2, the DRAT proof system is explained in Section 3 and the addition of symmetry-breaking predicates is presented in Section 4. Breaking a single symmetry may require many steps in order to express it in a DRAT proof, and this is discussed is Section 5. In Section 6, we explain how to break multiple symmetries. Our tool chain and evaluation are presented in Sections 7 and 8 and some conclusions are drawn in Section 9.

2 Preliminaries

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

Conjunctive Normal Form For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a finite disjunction of literals, and a CNF *formula* is a finite conjunction of clauses. A clause is a *tautology* if it contains both x and $\bar{x}$ for some variable x. We denote with ϵ the empty clause. Given a clause $C = (l_1 \vee \cdots \vee l_k)$, $\overline{C}$ denotes the conjunction of its negated literals, i.e., $(\bar{l}_1) \wedge \cdots \wedge (\bar{l}_k)$. The set of literals occurring in a CNF formula F is denoted by $\mathrm{LIT}(F)$. A truth assignment for a CNF formula F is a partial function τ that maps literals $l \in \mathrm{LIT}(F)$ to $\{\mathbf{t}, \mathbf{f}\}$. If $\tau(l) = v$, then $\tau(\bar{l}) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. Furthermore:

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

A CNF formula with no satisfying assignments is called *unsatisfiable*. A clause C is *logically implied* by formula F if adding C to F does not change the set of satisfying assignments of F. Two formulas are *logically equivalent* if they have the same set of solutions over the common variables. Two formulas are *satisfiability equivalent* if both have a solution or neither has a solution.

Resolution Given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the resolution rule states that the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We call C the resolvent of C_1 and C_2 and write $C = C_1 \diamond C_2$. C is logically implied by a formula containing C_1 and C_2.

Boolean Constraint Propagation and Asymmetric Tautologies For a CNF formula F, *Boolean constraint propagation* (BCP) (or *unit propagation*) simplifies F based on unit clauses; that is, it repeats the following until it reaches a fixpoint: If there is a unit clause $(l) \in F$, remove all clauses that contain the literal l from the set $F \setminus \{(l)\}$ and remove the literal $\bar{l}$ from all clauses in F. We write $F \vdash_1 \epsilon$ to denote that BCP applied to F can derive the empty clause. A clause C is an *asymmetric tautology* (AT) with respect to formula F if and only if $F \wedge \overline{C} \vdash_1 \epsilon$. Asymmetric tautologies are logically implied by F.

Example 1. Consider the formula $F = (\bar{a} \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$. Clause $C = (\bar{a})$ is an asymmetric tautology with respect to F, because $F \wedge \overline{C} \vdash_1 \epsilon$: BCP removes literal $\bar{a}$, resulting in the new unit clause (b). After removal of the literals $\bar{b}$, two complementary unit clauses (c) and $(\bar{c})$ are created. ■

Blocked Clauses Given a CNF formula F, a clause C, and a literal $l \in C$, the literal l *blocks* C with respect to F if (i) for each clause $D \in F$ with $\bar{l} \in D$, $C \diamond D$ is a tautology, or (ii) $\bar{l} \in C$, i.e., C is itself a tautology. Given a CNF formula F, a clause C is *blocked* with respect to F if there is a literal that blocks C with respect to F. Addition and removal of blocked clauses results in satisfiability-equivalent formulas [11], but not logically equivalent formulas.

Example 2. Consider the formula $(a \vee b) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee c)$. Clause $(a \vee \bar{b} \vee \bar{c})$ is a blocked clause, because its literal a is blocking it: the only resolution possibility is with $(\bar{a} \vee c)$ which results in tautology $(\bar{b} \vee \bar{c} \vee c)$. ■

3 Validating DRAT Proofs

Unsatisfiability proofs come in two flavors: *resolution proofs* and *clausal proofs*. A handful of formats have been designed for resolution proofs [12,13,14], but they all have the same disadvantages: resolution proofs are often huge and it is hard to express several important techniques, such as conflict clause minimization, as resolution steps. Other techniques, such as bounded variable addition, cannot be polynomially simulated by resolution. Clausal proof formats [15,16,4] are syntactically the same: a sequence of clauses that are claimed to be redundant with respect to a given formula. It is important that the redundancy can be checked in polynomial time. Clausal proofs may include deletion information to reduce the validation cost. The `drat-trim` [4] tool can efficiently validate clausal

CNF formula

```
p cnf 4 10
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

DRAT proof

```
  -1 0
d -1 2 4 0
   2 0
   0
```

Fig. 1. Left, a formula in DIMACS CNF format, the conventional input format for SAT solvers. Right, a DRAT proof for that formula. Each line in the proof is either an addition step (no prefix) or a deletion step identified by the prefix "`d`". Spacing in both examples is used to improve readability. Clause in the proof should be either asymmetric tautologies or be a RAT clauses using the first literal as the pivot.

proofs in the DRAT format —which is backwards compatible with earlier clausal proof formats— and was used to check the results of SAT Competition 2014.

Resolution asymmetric tautologies (or RAT clauses) [17] are a generalization of both asymmetric tautologies and blocked clauses. A clause C has RAT on l (referred to as the pivot literal) with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$ holds that

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

RAT is a very useful redundancy property because it can be computed in polynomial time and all preprocessing, inprocessing, and solving techniques in state-of-the-art SAT solvers can be expressed in terms of addition and removal of RAT clauses [17]. A DRAT *proof*, short for *Deletion Resolution Asymmetric Tautology*, is a sequence of addition and deletion steps of RAT clauses. Fig. 1 shows an example DRAT proof.

Example 3. Consider the CNF formula $F = (a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (b \vee c \vee \bar{d}) \wedge (\bar{b} \vee \bar{c} \vee d) \wedge (a \vee c \vee d) \wedge (\bar{a} \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee b \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$, which is shown in the DIMACS format in Fig. 1 (left). The first clause in the proof, $(\bar{a})$, is a RAT clause with respect to F, i.e., all possible resolvents are asymmetric tautologies:

$$\begin{aligned} F \wedge (a) \wedge (\bar{b}) \wedge (c) &\vdash_1 \epsilon \quad \text{using} \quad (a \vee b \vee \bar{c}) \\ F \wedge (a) \wedge (\bar{c}) \wedge (\bar{d}) &\vdash_1 \epsilon \quad \text{using} \quad (a \vee c \vee d) \\ F \wedge (a) \wedge (b) \wedge (d) &\vdash_1 \epsilon \quad \text{using} \quad (a \vee \bar{b} \vee \bar{d}) \end{aligned}$$

■

Let F be a CNF formula and P be a DRAT proof for F. The number of lines in a proof P is denoted by $|P|$. For each $i \in \{0, \ldots, |P|\}$, a CNF formula is defined F_P^i below. L_i refers to the lemma (redundant clause) on line i of P.

$$F_P^i := \begin{cases} F & \text{if } i = 0 \\ F_P^{i-1} \setminus \{L_i\} & \text{if the prefix of } L_i \text{ is "}\texttt{d }\text{"} \\ F_P^{i-1} \cup \{L_i\} & \text{otherwise} \end{cases}$$

Each lemma addition step is validated using a RAT check, while lemma deletion steps are ignored as their only purpose is to reduce the validation costs. Let l_i denote the first literal in lemma L_i. The RAT check for lemma L_i in proof P for CNF formula F succeeds if and only if L_i has the property RAT on literal l_i with respect to F_P^{i-1}. Moreover, lemma $L_{|P|}$ must be the empty clause.

4 Symmetries in Propositional Formulas

Two graphs G and H are *isomorphic* if there exists an edge-preserving bijection from the vertices of G to the vertices of H. A *symmetry* (or automorphism) of a graph G is an edge-preserving bijection of G onto itself. Symmetries in the graph representation of SAT problems may cause SAT solvers to explore symmetric parts of the search space again and again. This problem can be avoided by adding symmetry-breaking predicates [1].

The state-of-the-art approach to generate symmetry-breaking predicates for SAT solvers [8] works as follows. The clause-literal graph (explained below) of a given CNF formula is created for which the automorphisms are computed. The automorphisms in turn are converted into the symmetry-breaking predicates.

A *clause-literal graph* of a CNF formula F is an undirected graph containing a vertex for each clause and each literal in F. A literal vertex and a clause vertex are connected if and only if the corresponding literal occurs in the corresponding clause. Two clause vertices are never connected. Two literal vertices are connected if and only if the corresponding literals are each others complement.

We refer to a *symmetry* $\sigma = (x_1, \ldots, x_n)(p_1, \ldots, p_n)$ of a CNF formula F as an edge-preserving bijection of the corresponding clause-literal graph of F, that maps variable x_i onto p_i with $i \in \{1..n\}$. $(p_1, \ldots, p_n)$ is a permutation of $(x_1, \ldots, x_n)$ in which each p_i is potentially negated. If x_i is mapped onto p_i, then $\bar{x}_i$ is mapped onto $\bar{p}_i$. Also the clauses in the clause-literal graph are permuted by a symmetry, but we can ignore this aspect when it comes to symmetry breaking. Breaking σ can be achieved by enforcing that the assignment to literals $x_1, x_2, \ldots, x_n$ is lexicographically less or equal ($\leq$) than the assignment to literals $p_1, p_2, \ldots, p_n$. Instead of $\leq$, we could have used $\geq$.

Example 4. Consider the problem of whether a path of two edges is an unavoidable subgraph of graphs of order 3. We name the vertices a, b, and c. The existence of an edge between a and b, a and c, and b and c is represented by the Boolean variables $x_{a,b}$, $x_{a,c}$, and $x_{b,c}$, respectively. The propositional formula that expresses this problem and the labels of the clauses are shown below.

$$\overbrace{(x_{a,b} \vee x_{a,c})}^{C_1} \wedge \overbrace{(x_{a,b} \vee x_{b,c})}^{C_2} \wedge \overbrace{(x_{a,c} \vee x_{b,c})}^{C_3} \wedge \overbrace{(\bar{x}_{a,b} \vee \bar{x}_{a,c})}^{C_4} \wedge \overbrace{(\bar{x}_{a,b} \vee \bar{x}_{b,c})}^{C_5} \wedge \overbrace{(\bar{x}_{a,c} \vee \bar{x}_{b,c})}^{C_6}$$

The clause-literal graph of this formula is shown in Fig. 2 together with three isomorphic copies that can be obtained by permuting the nodes of the clause-literal graph. The three symmetries are (ignoring the permutation of clauses):

$$\sigma_1 = (x_{a,b}, x_{a,c}, x_{b,c})(\bar{x}_{a,b}, \bar{x}_{a,c}, \bar{x}_{b,c}); \quad \sigma_2 = (x_{a,b})(x_{a,c}); \quad \sigma_3 = (x_{a,c})(x_{b,c}).$$

The symmetry-breaking tool `shatter` will break these symmetry by adding the constraints $x_{a,b}, x_{a,c}, x_{b,c} \leq \bar{x}_{a,b}, \bar{x}_{a,c}, \bar{x}_{b,c}$ (which can be simplified to the constraint $(x_{a,b} \leq \bar{x}_{a,b}) \equiv \bar{x}_{a,b}$); $x_{a,b} \leq x_{a,c}$; and $x_{a,c} \leq x_{b,c}$, respectively. These constraints correspond to the clauses $(\bar{x}_{a,b})$, $(\bar{x}_{a,b} \vee x_{a,c})$, and $(\bar{x}_{a,c} \vee x_{b,c})$. ■

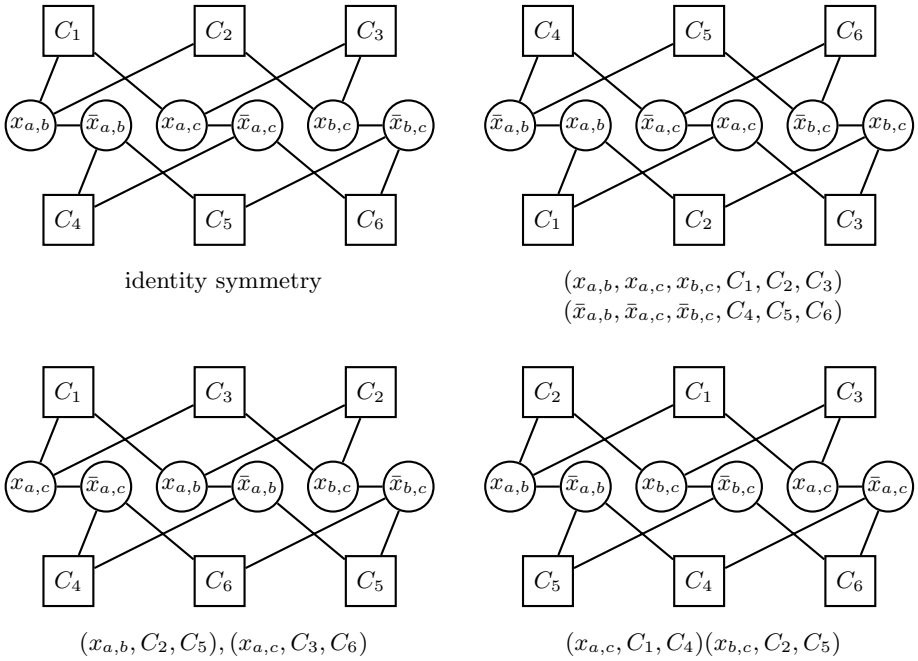


Fig. 2. Four isomorphic clause-literal graphs of the propositional formula of Example 4. Notice that the four graphs are identical modulo the labelling of the nodes. Below each graph, the permutation of the nodes is shown compared to the top-left graph. The permutation of clauses is omitted in symmetries throughout this paper.

Expressing the constraint $x_1, x_2, \ldots, x_n \leq p_1, p_2, \ldots, p_n$ in clauses with (only) variables in F can be done as follows:

$$(\bar{x}_1 \vee p_1) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee p_2) \wedge (p_1 \vee \bar{x}_2 \vee p_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee p_3) \wedge$$
$$(\bar{x}_1 \vee p_2 \vee \bar{x}_3 \vee p_3) \wedge (p_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee p_3) \wedge (p_1 \vee p_2 \vee \bar{x}_3 \vee p_3) \wedge \ldots$$

The above scheme adds $2^n - 1$ clauses. Using auxiliary variables, it requires only a linear number of clauses, i.e., $3n - 2$, to express this constraint:

$$\begin{aligned} &(\bar{x}_1 \vee p_1) \wedge (a_1 \vee \bar{x}_1) \wedge (a_1 \vee p_1) \wedge (\bar{a}_{n-1} \vee \bar{x}_n \vee p_n) \wedge \\ \bigwedge_{i \in \{2..n-1\}} &\big((\bar{a}_{i-1} \vee \bar{x}_i \vee p_i) \wedge (a_i \vee \bar{a}_{i-1} \vee \bar{x}_i) \wedge (a_i \vee \bar{a}_{i-1} \vee p_i)\big) \end{aligned} \tag{1}$$

Optionally some *blocked clauses* [11] can be added: $(\bar{a}_1 \vee x_1 \vee \bar{p}_1)$ and $(\bar{a}_i \vee a_{i-1}) \wedge (\bar{a}_i \vee x_i \vee \bar{p}_i)$ for $i \in \{2..n-1\}$. State-of-the-art SAT solvers, such as `Lingeling`, would remove them during preprocessing since they are useless in practice [18].

The main question that we will answer in the next two sections is how to convert F into a satisfiability-equivalent formula F' such that the above clauses have DRAT with respect to F' and can therefore be added.

5 Breaking a Single Symmetry

This section shows how to express breaking a single symmetry in a DRAT proof. Breaking multiple symmetries is more complicated; this will be discussed in Section 6. Breaking a single symmetry consists of three steps: adding definitions, redefine involved clauses, and adding symmetry-breaking predicates. Below we discuss these three steps in detail using the following notation. The formula F_0 expresses the initial formula for which we want to break a symmetry σ. Formula F_1 expresses the result of adding definitions (step 1); formula F_2 expresses the result after redefining involved clauses in F_1 (steps 1 and 2); and formula F_3 expresses the result after adding symmetry-breaking predicates to F_2 (all steps).

Adding definitions The first step consists of defining auxiliary variables. A pivotal variable is the *primal-swap variable* s_1. For a given symmetry $\sigma = (x_1, \ldots, x_n)(p_1, \ldots, p_n)$, s_1 is false for every assignment for which $x_1, \ldots, x_n \leq p_1, \ldots, p_n$ and true otherwise. Only if s_i is assigned to true, we want to swap x_i and p_i with $i \in \{1..n\}$. We will use auxiliary variables s_i with $i \in \{1..n\}$ and $6n - 3$ clauses to express the potential swap:

$$\begin{aligned}\textstyle\bigwedge_{i\in\{1..n-1\}} \big((s_i \vee \bar{x}_i \vee p_i) \wedge (s_i \vee \bar{x}_i \vee \bar{s}_{i+1}) \wedge (s_i \vee p_i \vee \bar{s}_{i+1}) \wedge \\ (\bar{s}_i \vee x_i \vee \bar{p}_i) \wedge (\bar{s}_i \vee x_i \vee s_{i+1}) \wedge (\bar{s}_i \vee \bar{p}_i \vee s_{i+1})\big) \wedge \\ (s_n \vee \bar{x}_n \vee p_n) \wedge (\bar{s}_n \vee x_n) \wedge (\bar{s}_n \vee \bar{p}_n)\end{aligned}$$

These clauses can be added to the formula by blocked clause addition in the reverse order as listed in the equation above: first add all clauses containing literals s_n and $\bar{s}_n$, second add clauses containing literals s_{n-1} and $\bar{s}_{n-1}$, etc.

Additionally we introduce n auxiliary Boolean variables x'_i with $i \in \{1..n\}$ which are defined as follows. If the primal-swap variable s_1 is assigned to false, then $x'_i \leftrightarrow x_i$, otherwise $x'_i \leftrightarrow p_i$. In clauses this definition is expressed as

$$\bigwedge_{i\in\{1..n\}} \big((x'_i \vee \bar{x}_i \vee s_1) \wedge (\bar{x}'_i \vee x_i \vee s_1) \wedge (x'_i \vee \bar{p}_i \vee \bar{s}_1) \wedge (\bar{x}'_i \vee p_i \vee \bar{s}_1)\big).$$

All these clauses are blocked on the x_i and $\bar{x}_i$ literals. The definitions of x'_1 and p'_1 can be expressed more compactly using only three clauses per definition:

$$\begin{aligned}(x'_1 \vee \bar{x}_1 \vee \bar{p}_1) \wedge (\bar{x}'_1 \vee x_1) \wedge (\bar{x}'_1 \vee p_1) \quad &\equiv \quad x'_1 := \mathrm{AND}(x_1, p_1) \\ (\bar{p}'_1 \vee x_1 \vee p_1) \wedge (p'_1 \vee \bar{x}_1) \wedge (p'_1 \vee \bar{p}_1) \quad &\equiv \quad p'_1 := \mathrm{OR}(x_1, p_1)\end{aligned}$$

The more compact definitions are also blocked on the prime literals. All clauses contain only one prime literal and all clauses are blocked on the prime literal. Therefore, they can be added to the DRAT proof in arbitrary order.

Redefining Involved Clauses In the second step of expressing breaking symmetry $\sigma = (x_1, \ldots, x_n)(p_1, \ldots, p_n)$ as a sequence of DRAT operations, we redefine the *involved clauses* C_j, i.e., those clauses in F_0 that contain at least one literal x_i or $\bar{x}_i$ with $i \in \{1..n\}$ by clauses C'_j, a copy of C_j with all literals x_i and $\bar{x}_i$ replaced by literals x'_i and $\bar{x}'_i$, respectively.

The clauses C'_j do not have RAT with respect to F_1, the formula resulting after adding definitions. However, the clauses $C'_j \cup \{s_1\}$ and $C'_j \cup \{\bar{s}_1\}$ have AT with respect to F_1, with s_1 referring to the primal-swap variable from the prior step. Using this observation, we express redefining C_j into C'_j with $j \in \{1..m\}$ using $4m$ operations: add $C'_j \cup \{s_1\}$, add C'_j, delete $C'_j \cup \{s_1\}$ and delete C_j. Notice that we use $C'_j \cup \{s_1\}$ as an auxiliary clause to add C'_j: C'_j has AT with respect to $F_1 \cup \{C'_j \cup \{s_1\}\}$ because $C'_j \cup \{\bar{s}_1\}$ has AT with respect to F_1.

Adding Symmetry-Breaking Predicates After adding definitions (step 1) and redefining involved clauses (step 2), all assignments for which $x'_1, \ldots, x'_n > p'_1, \ldots, p'_n$ are eliminated: if $x_1, \ldots, x_n > p_1, \ldots, p_n$, then s_1 is assigned to true with will swap x_i and p_i with $i \in \{1..n\}$. To express this knowledge, i.e., $x'_1, \ldots, x'_n \leq p'_1, \ldots, p'_n$, in clauses that have DRAT with respected to F_2, we first introduce auxiliary variables y_i as follows:

$$\begin{aligned}&(y_1 \vee \bar{x}'_1) \wedge (y_1 \vee p'_1) \wedge (\bar{y}_1 \vee x'_1 \vee \bar{p}'_1) \wedge \\ &\bigwedge_{i \in \{2..n-1\}} \big((y_i \vee \bar{y}_{i-1} \vee \bar{x}'_i) \wedge (y_i \vee \bar{y}_{i-1} \vee p'_i) \wedge (\bar{y}_i \vee y_{i-1}) \wedge (\bar{y}_i \vee x'_i \vee \bar{p}'_i)\big)\end{aligned} \tag{2}$$

Notice that all these clauses have RAT on their first literal when added in the order as shown in (2). Afterwards, we add the following clauses:

$$(\bar{x}'_1 \vee p'_1) \wedge \bigwedge_{i \in \{2..n\}} (\bar{y}_{i-1} \vee \bar{x}'_i \vee p'_i) \tag{3}$$

The clauses (3) are logically implied by F_2 after the addition of (2). Notice that the clauses (2) and (3) together are the same as (1), but with the blocked clauses. The blocked clauses are required to add (3), but can be removed afterwards.

6 Breaking Multiple Symmetries

Given k symmetries, tools that add symmetry-breaking predicates simply add the clauses (1) once for each symmetry. However, expressing breaking $k > 1$ symmetries in DRAT cannot simply be done by applying the above procedure (all three steps) only once for each symmetry. In fact, it appears that in worst case the procedure needs to be applied significantly more often than k times. First, we will describe the origin of applying the procedure more than a linear number of times: overlap in the variables in symmetries. Afterwards, we will explore how to apply the procedure as few times as possible. We conclude this section by showing that also the the length of symmetries can significantly increase the number of applications of the procedure.

Overlapping Symmetries It can be quite costly to express breaking multiple symmetries in DRAT, even in case all symmetries have length 1. The example below illustrates the difficulties using two symmetries.

Example 5. Consider the formula $F = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \vee (x_2 \vee x_3) \vee (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ which has two symmetries: $\sigma_1 = (x_1)(x_2)$ and $\sigma_2 = (x_2)(x_3)$. Breaking symmetry σ_1 would result in adding the clauses

$$(x'_1 \vee \bar{x}_1 \vee \bar{x}_2), (\bar{x}'_1 \vee x_1), (\bar{x}'_1 \vee x_2), (\bar{x}'_2 \vee x_1 \vee x_2), (x'_2 \vee \bar{x}_1), (x'_2 \vee \bar{x}_2).$$

Applying the definitions, F can be converted to $F' = (x'_1 \vee x'_2) \wedge (x'_1 \vee x_3) \vee (x'_2 \vee x_3) \vee (\bar{x}'_1 \vee \bar{x}'_2 \vee \bar{x}_3)$. From the definitions it follows that $x'_1 \leq x'_2$, or $(\bar{x}'_1 \vee x'_2)$. Now, let us break symmetry σ_2 by adding the clauses

$$(x''_2 \vee \bar{x}'_2 \vee \bar{x}_3), (\bar{x}''_2 \vee x'_2), (\bar{x}''_2 \vee x_3), (\bar{x}'_3 \vee x'_2 \vee x_3), (x'_3 \vee \bar{x}'_2), (x'_3 \vee \bar{x}_3).$$

Again, applying the definitions, F' can be converted to $F'' = (x'_1 \vee x''_2) \wedge (x'_1 \vee x'_3) \vee (x''_2 \vee x'_3) \vee (\bar{x}'_1 \vee \bar{x}''_2 \vee \bar{x}'_3)$. Notice that $(\bar{x}'_1 \vee x''_2)$, i.e., $x'_1 \leq x''_2$ does not hold. Consider the satisfying assignment $x_1 = 1$, $x_2 = 1$, $x_3 = 0$. Following the definitions, $x'_1 = 1$, $x'_2 = 1$ and $x''_2 = 0$, $x'_3 = 1$. Observe that $0 = x''_2 < x'_1 = 1$. In order to break both σ_1 and σ_2, we need to break σ_1 again. ■

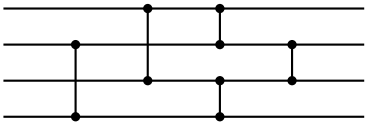
The problem in Example 5 is caused by the overlap (shared variables) in the symmetries. In worst case, all symmetries overlap which requires applying the symmetry-breaking procedure, i.e., the three steps to break a single symmetry, again and again.

The symmetry-breaking procedure presented in Section 5 changes and arbitrary assignment A into an assignment A' that is lexicographically smaller or equal than A according to the symmetry at hand. This procedure has similarities with sorting numbers if certain dependencies between symmetries are missing (which will be discussed at the end of this section). Consequently, we can use sorting technology to perform the procedure as few times as possible.

Sorting Networks A *sorting network*, consisting of k wires and c comparators, sorts k values using c comparisons. The values flow across the wires. A comparator connects two wires, compares the incoming values, and sorts them by outputting the smaller value to one wire, and the larger to the other. The left part of Fig. 3 shows a sorting network of four wires (horizontal lines) and five comparators (vertical lines). This sorting network is optimal: there does not exist a sorting network of four wires with less than five comparators. The table in Fig. 3 shows the size of optimal sorting networks with few wires. Recently it was shown that the smallest sorting network with ten wires requires 29 comparators [7].

Sorting networks have been studied for over sixty years. There exists several algorithms that produce small sorting networks. One of the most best algorithms is Batcher's Merge-Exchange algorithm [19] which produces sorting networks with $\mathcal{O}(k \log^2 k)$ comparators with k being the number of wires. There exists a construction method the uses only $\mathcal{O}(k \log k)$ comparators [20], but this method produces larger networks compared to Batcher's Merge-Exchange for small k.

If symmetries are broken using a sorting network scheme, we obtain the same bounds for the number of swaps unless certain dependencies between the symmetries exist. Below we discuss the exception.



k	2	3	4	5	6	7	8	9	10
optimal	1	3	5	9	12	16	19	25	29
$k(k-1)/2$	1	3	6	10	15	21	28	36	45

Fig. 3. Left, an optimal (fewest wires) sorting network with four inputs and five wires. Right, a table showing the size of optimal k-input sorting network for small k [7].

Breaking Multiple Long Symmetries To this point, we considered symmetries of length 1, which lack certain dependencies. In general, however, symmetries tend to be larger. For k symmetries with length $n > 1$, it may require more than $\mathcal{O}(k \log k)$ swaps to fully break the symmetries.

Example 6. Consider a formula with two symmetries: $\sigma_1 = (x_1, x_4)(x_2, x_5)$ and $\sigma_2 = (x_2, x_4)(x_3, x_6)$. Notice that the variables are not uniquely mapped to each other: x_4 is mapped to x_5 in σ_1 and to x_6 in σ_2. When variables are not uniquely mapped to each other, which is the case for unavoidable graph problems, then more than $k \log k$ swaps may be needed to convert each assignment such that it is lexicographically smallest with respect to all k symmetries. Given two symmetries such that the variables are uniquely mapped, then we would need at most 3 swaps (the size of the smallest sorting network with $(k+1) = 3$ wires). However, the sequence below shows that with σ_1 and σ_2 we need four swaps for the assignment $x_1 = x_2 = x_4 = x_6 = 1$ and $x_3 = x_5 = 0$.

$$\begin{array}{c|c|c} x_1 & x_2 & x_3 \\ \hline 1 & 1 & 0 \\ \hline 1 & 0 & 1 \\ x_4 & x_5 & x_6 \end{array} \xrightarrow{\sigma_1} \begin{array}{c|c|c} x'_1 & x'_2 & x_3 \\ \hline 1 & 1 & 0 \\ \hline 0 & 1 & 1 \\ x'_4 & x'_5 & x_6 \end{array} \xrightarrow{\sigma_2} \begin{array}{c|c|c} x'_1 & x''_2 & x'_3 \\ \hline 1 & 0 & 1 \\ \hline 1 & 1 & 0 \\ x''_4 & x'_5 & x'_6 \end{array} \xrightarrow{\sigma_1} \begin{array}{c|c|c} x''_1 & x'''_2 & x'_3 \\ \hline 0 & 1 & 1 \\ \hline 1 & 1 & 0 \\ x'''_4 & x''_5 & x'_6 \end{array} \xrightarrow{\sigma_2} \begin{array}{c|c|c} x''_1 & x''''_2 & x''_3 \\ \hline 0 & 1 & 1 \\ \hline 0 & 1 & 1 \\ x''''_4 & x''_5 & x''_6 \end{array}$$

■

We experimented with unavoidable graph problems and observed that given k symmetries of length n, $nk \log k$ swaps are required: apply the $k \log k$ steps n times. We conjecture that this is the worst case.

Conjecture 1. Given a CNF formula F with k symmetries of length n, it requires in worst case $\mathcal{O}(nk \log k)$ swaps to convert a symbolic assignment A into an assignment A' for which hold that A' is the lexicographically smallest assignment according to all k symmetries.

7 Tools and Evaluation

Producing a DRAT proof for a given formula F that incorporates symmetry breaking involves the use of several tools. Fig. 4 shows an overview of the tool chain. Six tools are used: a formula-to-graph converter, a symmetry extractor, a symmetry-breaking converter, a SAT solver, and a DRAT proof checker. These tools are used in four phases:

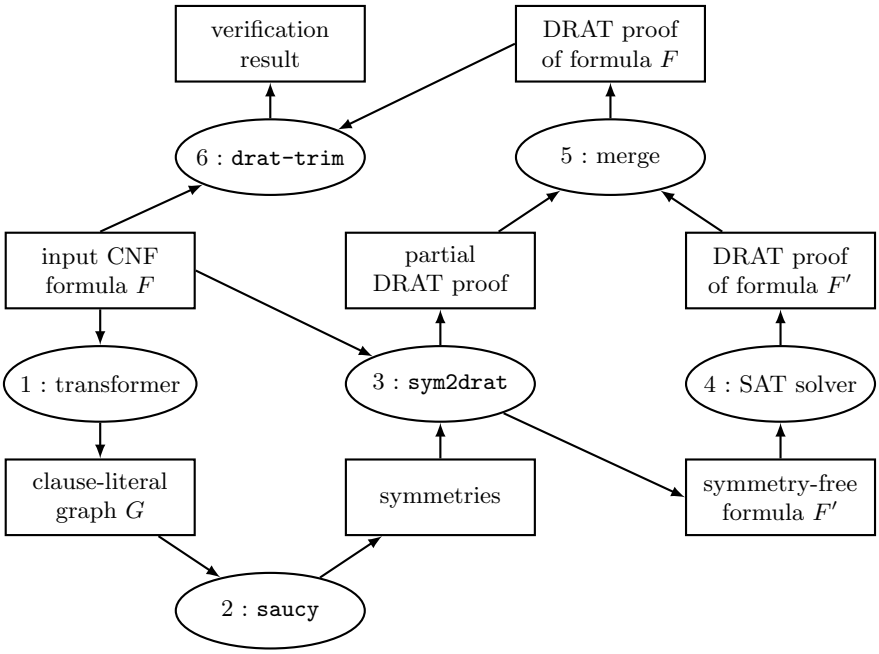


Fig. 4. Tool chain to produce DRAT proofs that incorporate symmetry breaking. The rectangle boxes are files, while the round boxes are tools. Phase I consists of the tools transformer and `saucy`, phase II consists of the `sym2drat` tool, phase III uses an off-the-shelf SAT solver, and phase IV consists of tools to merge and validate the proofs.

I The symmetries σ of F are computed by transforming F into a clause-literal graph (see Section 4). A symmetry-extraction tool, such as `saucy` [21], can be used to obtain the symmetries.

II Formula F is converted into a satisfiability-equivalent formula F', a copy of F for which the symmetries σ are broken. F' is equivalent (modulo variable renaming) to adding symmetry-breaking predicates to F using a symmetry-breaking tool, such as `shatter` [8]. Additionally, the conversion from F to F' is expressed as a partial DRAT proof. Our new tool, `sym2drat`, implements this second phase, i.e., computing F' and a partial DRAT proof.

III The formula F' is solved by a SAT solver, which produces a DRAT proof. Most state-of-the-art SAT solvers now support emission of such proofs.

IV The last step consists of verifying the result of both the symmetry-breaking tool and the SAT solver. The partial DRAT proof and the DRAT proof of F' are merged, which can be simply done by concatenating the proofs. A proof checker, such as `drat-trim` [4], validates whether the merged proof is a refutation for the input formula F.

Below we will discuss the tools that we developed for phases II and IV. We used off-the-shelf tools for the phases I and III.

7.1 The tool `sym2drat`

The main tool that we developed for expressing symmetry-breaking as DRAT proofs is `sym2drat`. This tool requires two inputs: a CNF formula F and a set of symmetries of F. Two files are emitted by `sym2drat`: a CNF formula F' with symmetry-breaking predicates and a partial DRAT proof that expresses the conversion of F into F'. Our tool `sym2drat` constructs sorting networks based on the Batcher's Merge-Exchange algorithm [19] which reduces the number of swaps (and thus the size of the partial DRAT proof) by roughy a factor of two compared to bubble sort on most problems we experimented with, i.e., problems containing around 20 symmetries. The `sym2drat` tool is still in a prototype phase as we do not know yet the exact bounds for the number of swaps required to break all symmetries. Throughout our experimentation we determined manually whether to apply a sorting network once or multiple times.

7.2 Improving DRAT proof-checking tools

Apart from implementing `sym2drat`, we improved two tools that validate DRAT proofs. The first tool we improved is `drat-trim`: the fast DRAT proof checking written in C that was used to validate the results of SAT Competition 2014. The current version of `drat-trim` does not support validating *partial proofs*: a sequence of clauses that are all redundant with respect to a given formula, but that does not terminate with the empty clause. Being able to check partial proofs allows the validation of the output of `sym2drat`. We extended `drat-trim` with the option to validate partial proofs[1]. This feature was very useful for developing our method to express using symmetry-breaking in DRAT proofs. We expect this feature to be helpful to discover how other techniques, such as Gaussian Elimination and cardinality resolution, can be expressed with DRAT proofs. Moreover, this feature allows checking partial runs of SAT solvers and it can be used for checking runs on satisfiable problems.

Our mechanically-verified, RAT validation tool [22], written in ACL2, has undergone significant improvements. This tool was originally designed to demonstrate the soundness of a basic algorithm used to validate RAT proofs. Efficiency of the tool was not a priority. Recent work [23] has been devoted to improving the performance of this tool while maintaining its proof of correctness (soundness). The underlying data structures have been moved from `cons`-based lists to ACL2 `STOBJ`s (Single Thread OBJects) which offer support for LISP arrays, reducing the linear-time cost for accesses and updates to constant-time. This seemingly small change has a large impact on performance but also required a substantial proof effort. A new ACL2 data structure, called `farray`, was developed to facilitate proof development with `STOBJ`s. A mechanical proof of equivalence was established to show that the new tool behaves exactly the same as the original tool, preserving much of the proof of correctness of the original tool.

[1] available at `http://www.cs.utexas.edu/~marijn/drat-trim/`

8 Evaluation

We evaluate the usefulness of our new method by computing and validating "compact" DRAT proofs[2] of some hard combinatorial problems.

Ramsey number four. Ramsey theory addresses unavoidable patterns. The most well-known pattern is unavoidable cliques. The size of the smallest graph that has an unavoidable clique of size k is called Ramsey number k. Ramsey number four is 18. Showing that any graph of size 18 has and unavoidable clique of size 4 can be encoded using a formula consisting of $2 \cdot \binom{18}{4} = 6120$ clauses, each of length 6. We tried to solve this formula using the SAT solvers `Lingeling` and `glucose`, but both solvers were unable to determine in 24 hours that the formula is unsatisfiable.

The CNF formula F that encodes Ramsey number four has 18 symmetries: any permutation of vertices and complementing the graph. SAT solvers can determine that formula F' with symmetry-breaking predicates, produced by `sym2drat`, is unsatisfiable in less than a second. We merged the proof of F', produced by `glucose` 3.0, with the partial DRAT proof, produced by `sym2drat`. This proof can be checked by `drat-trim` in 1.9 seconds. We validated the proof using our ACL2-based, mechanically-verified RAT checker [23] as well. Notice that these tools allow to obtain a mechanically-verified proof in the theorem prover ACL2 that Ramsey number four is 18. We envision that this tool chain is a useful template to obtain trustworthy results of hard combinatorial problems.

Erdős Discrepancy Conjecture. Let $S = \langle s_1, s_2, s_3, \dots \rangle$ be an infinite sequence of 1's or -1's. Erdős Discrepancy Conjecture states that for any C there exists an d and k such that

$$\left| \sum_{i=1}^{k} s_{i \cdot d} \right| > C$$

Recently, the case $C = 2$ was proved using SAT solvers, resulting in a DRUP proof of 13Gb [5]. The problem contains a symmetry (swapping 1's and -1's), but it was not broken in the original approach. We proved the conjecture using our tool chain with `glucose` 3.0 and validated the DRAT proof using `drat-trim`. The size of our proof is slightly more than 2Gb in syntactically the same format as the original proof. Symmetry breaking allowed us to pick a variable which can be added to the formula, similar to the unit ($\bar{x}_{a,b}$) in Example 4. We choose unit ($\bar{s}_{60}$) as it occurs frequently in the original CNF formula. The combination of symmetry-breaking and selecting a good unit resulted in a proof a sixth of the size of the original one. The tool `drat-trim` can reduce the new proof to 850 Mb by removing redundant lemmas and discarding the deletion information.

Two Pigeons per Hole. One family of hard problems of the SAT Competitions of 2013 and 2014 are a variation of the *pigeon hole principle*. The Two-Pigeons-per-Hole (TPH) family consists of problems encoding that $2k + 1$ pigeons can

[2] available at `http://www.cs.utexas.edu/~marijn/sbp/`

be placed into k holes such that each hole has at most two pigeons. Most SAT solvers can refute the problem for $k = 6$, but they cannot solve problems of size $k > 6$ within an hour, unless symmetry breaking or cardinality resolution is applied. It is not known yet how to express cardinality resolution in the existing SAT proof formats. Thus, there existed no approach to produce DRAT proofs for the difficult instances of this family ($k > 6$). Using our method, we were able to produce and check DRAT proofs for these problems for $k \leq 12$ within an hour.

A TPH problem of size k contains $2k + 1$ symmetries of length k expressing that the pigeons are interchangeable. After breaking these symmetries, TPH problems become very easy and can be solved instantly. These symmetries are overlapping, but grouped. Therefore, "only" $(2k + 1)\log(2k + 1)$ swaps are required. However, the number of involved clauses per symmetry is large and the formula contains many clauses. As a consequence, expressing a single swap results in many DRAT steps. For $k = 12$, our method results in a 4Gb proof. The size of the DRAT proof sharply increases with k and so does the time to validate the proof.

9 Conclusions

Validating proofs of unsatisfiability helps to gain confidence in the correctness of results by SAT solvers which have been shown to contain errors on the implementation [24] and conceptual level [17]. We presented a method to express symmetry breaking in DRAT, the most widely-supported proof format for SAT solvers. Our method allows, for the first time, validation of SAT solver results obtained via symmetry breaking, thereby validating the results of symmetry extraction tools as well.

Symmetry breaking is often crucial to solve hard combinatorial problems. Our method provides a missing link to establish trust that results on these problems are correct. We demonstrated our method on hard combinatorial problems such as Ramsey number four and the Erdős Discrepancy Conjecture. We also constructed DRAT proofs of two-pigeons-per-hole (TPH) problems. TPH problems are special as they were among the few formulas used in recent SAT competitions that can be solved, but only without a proof. Hence, this work brings us closer to validation of all SAT solver results.

References

1. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: Proc. KR96, 5th Int. Conf. on Knowledge Representation and Reasoning. Morgan Kaufmann (1996) 148–159
2. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult sat instances in the presence of symmetry. In: Design Automation Conference, 2002. Proceedings. 39th. (2002) 731–736

3. Gent, I.P., Smith, B.M.: Symmetry breaking in constraint programming. In Horn, W., ed.: ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000, IOS Press (2000) 599–603
4. Wetzler, N., Heule, M.J.H., Hunt, Warren A., J.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Sinz, C., Egly, U., eds.: SAT 2014. Volume 8561 of LNCS. Springer (2014) 422–429
5. Konev, B., Lisitsa, A.: A SAT attack on the Erdős Discrepancy Conjecture. In Sinz, C., Egly, U., eds.: SAT 2014. Volume 8561 of LNCS. Springer (2014) 219–226
6. Kouril, M., Paul, J.L.: The van der Waerden number W(2, 6) is 1132. Experimental Mathematics **17**(1) (2008) 53–61
7. Codish, M., Cruz-Filipe, L., Frank, M., Schneider-Kamp, P.: Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In: ICTAI 2014, IEEE Computer Society (2014) 186–193
8. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for boolean satisfiability. IEEE Trans. Computers **55**(5) (2006) 549–558
9. Schaafsma, B., Heule, M.J.H., van Maaren, H.: Dynamic symmetry breaking by simulating zykov contraction. In Kullmann, O., ed.: SAT 2009. Volume 5584 of LNCS. Springer (2009) 223–236
10. Radziszowski, S.P.: Small Ramsey numbers. The Electronic Journal of Combinatorics **#DS1** (2014)
11. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics **96-97** (1999) 149–176
12. Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE. (2003) 10880–10885
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518
14. Biere, A.: Picosat essentials. JSAT **4**(2-4) (2008) 75–97
15. Van Gelder, A.: Verifying rup proofs of propositional unsatisfiability. In: ISAIM. (2008)
16. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.: Verifying refutations with extended resolution. In: International Conference on Automated Deduction (CADE). Volume 7898 of LNAI., Springer (2013) 345–359
17. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In Gramlich, B., Miller, D., Sattler, U., eds.: IJCAR. Volume 7364 of LNCS., Springer (2012) 355–370
18. Järvisalo, M., Biere, A., Heule, M.J.H.: Blocked clause elimination. In Esparza, J., Majumdar, R., eds.: TACAS. Volume 6015 of LNCS., Springer (2010) 129–144
19. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of Spring Joint Computer Conference. AFIPS '68, ACM (1968) 307–314
20. Ajtai, M., Komlós, J., Szemerédi, E.: An $\mathcal{O}(n \log n)$ sorting network. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing. STOC '83, New York, NY, USA, ACM (1983) 1–9
21. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for cnf. In: DAC. DAC '04, ACM (2004) 530–534
22. Wetzler, N., Heule, M.J.H., Hunt, W.A.: Mechanical verification of sat refutations with extended resolution. In: ITP 2013. ITP'13, Springer (2013) 229–244
23. Wetzler, N.: Mechanically-Verified Validation of Satisfiability Solvers. (To Appear.)
24. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: SAT 2010. SAT'10, Springer (2010) 44–57