SAT Preprocessors and Symmetry

Markus Anders
TU Darmstadt, Germany

Abstract

Exploitation of symmetries is an indispensable approach to solve certain classes of difficult SAT instances. Numerous techniques for the use of symmetry in SAT have evolved over the past few decades. But no matter how symmetries are used precisely, they have to be detected first. We investigate how to detect more symmetry, faster. The initial idea is to reap the benefits of SAT preprocessing for symmetry detection. As it turns out, applying an off-the-shelf preprocessor before handling symmetry runs into problems: the preprocessor can haphazardly remove symmetry from formulas, severely impeding symmetry exploitation.

Our main contribution is a theoretical framework that captures the relationship of SAT preprocessing techniques and symmetry. Based on this, we create a symmetry-aware preprocessor that can be applied safely before handling symmetry. We then demonstrate that applying the preprocessor does not only substantially decrease symmetry detection and breaking times, but also uncovers hidden symmetry not detectable in the original instances. Overall, we depart the conventional view of treating symmetry detection as a black-box, presenting a new application-specific approach to symmetry detection in SAT.

2012 ACM Subject Classification Theory of computation → Complexity theory and logic; Theory of computation → Graph algorithms analysis; Mathematics of computing → Graph algorithms

Keywords and phrases boolean satisfiability, symmetry exploitation, graph isomorphism

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.1

Related Version *Previous Version*: `https://arxiv.org/abs/2205.12799`

Funding The research leading to these results has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EngageS: grant agreement No. 820148).

Acknowledgements I want to thank Moritz Lichter, Pascal Schweitzer, Constantin Seebach and Damien Zufferey for the valuable discussions at different stages of the project. I also want to thank the anonymous reviewers at SAT 2022 for pointing out an error with the counter-examples used in an earlier version of the paper.

1 Introduction

Many difficult classes of SAT instances contain a large number of symmetries. Exploitation of symmetries is an indispensable tool to speed up solving these instances. Various techniques, such as for example the use of symmetry breaking predicates or symmetry-based DPLL branching rules, have evolved over the past few decades [20, 14]. In fact, there is still ongoing research on how to use symmetries best while solving SAT [10, 17, 23, 22].

In practice, state-of-the-art symmetry exploitation is based on syntactic symmetries of the formula [20, 10, 17, 23, 22]. Syntactic symmetries are permutations of variables (or literals) that map a formula F back to itself, i.e., a permutation φ is a syntactic symmetry whenever $\varphi(F) = F$ holds. The most common way to compute these symmetries is to first model the given input formula as a graph. Then, the automorphism group of this model graph is computed using a graph isomorphism solver (e.g., [7, 4, 16]). In fact, computing syntactic symmetries is polynomial-time equivalent to the graph isomorphism problem [6].

Table 1 Pigeonhole principle solved with CRYPTOMINISAT (C), BREAKID+CRYPTOMINISAT (B+C), and CRYPTOMINISAT (preprocessor)+BREAKID+CRYPTOMINISAT (P+B+C).

	C	B+C		P+B+C	
instance	T_{solve}	T_{solve}	#syms	T_{solve}	#syms
php(9,8)	6.74s	<1s	$1.46 \cdot 10^{10}$	<1s	$2.07 \cdot 10^{6}$
php(15,14)	>60s	<1s	$1.14 \cdot 10^{23}$	>60s	$8.72 \cdot 10^{10}$
php(21,20)	>60s	<1s	$1.24 \cdot 10^{38}$	>60s	$2.43 \cdot 10^{18}$

Generally speaking, this describes the two major components that make up the use of symmetry in SAT: first, there is symmetry detection, i.e., finding symmetries of the formula in the first place. Secondly, there is the exploitation of symmetries itself, i.e., using symmetries to cut away parts of the search space. In this paper, we investigate and improve the former: how to detect more symmetry, faster.

Due to the sheer size of SAT instances and number of symmetries, symmetry detection can indeed become expensive. So much so, that for the state-of-the-art symmetry breaking tool BREAKID, the version that limits the time used for symmetry detection outperforms the version that is not time-limited [10]. Often, the reason why handling symmetry is slow, is that the underlying instances are bloated with many easily reducible variables and clauses. Some symmetry detection tools try to rectify this by using intricate graph-level preprocessing techniques. In fact, these techniques have been meticulously engineered to deal with structures common in CNF formulas, i.e., low-degree vertices [7, 8, 16, 19]. While this marks the currently most successful approach to symmetry detection for CNF formulas, it does not yet exploit the underlying semantics of SAT.

In the SAT domain, the typical first step in tackling large formulas would be to first apply a SAT preprocessor to reduce the formulas [11]. This naturally leads to the question: could applying a SAT preprocessor *before* symmetry detection reduce computation time, and maybe even aid in finding more symmetry? Indeed, if applying a SAT preprocessor were to improve symmetry detection in any way, then this could be considered a win-win situation: SAT preprocessing techniques are usually applied anyway and hence the resulting improvement would be virtually free. Unfortunately, after conducting simple testing (see Table 1) we find that applying an off-the-shelf preprocessor (e.g., [21]) *before* dealing with symmetry (e.g., [10]) does not work for symmetry detection. The test suggests that even in the basic case of the pigeonhole principle, the preprocessor *removes* symmetries in a way that renders the subsequent symmetry breaking ineffective.

On the other hand, preprocessing can actually also lead to *more* symmetry: consider the CNF formula $(x) \wedge (\overline{x} \vee a \vee c) \wedge (b \vee c)$. Without any alteration, the formula has no non-trivial symmetries. However, when we apply, say, the unit rule on x, the formula becomes $(a \vee c) \wedge (b \vee c)$. This in turn makes a and b symmetrical. Indeed, in this example, simplifying the formula allows us to detect more symmetry.

Confusingly, we thus find that SAT preprocessing can both lead to less, as well as more symmetry in formulas. This raises several questions. What is the right order of operations? Can we choose and schedule SAT preprocessing techniques in such a manner, that we both increase interesting symmetry while making the formula easier for symmetry detection? Overall, it seems that a fundamental understanding of the effect of preprocessing techniques on symmetry is required.

The ultimate goal of this line of research is to develop symmetry detection for SAT that maximizes detected symmetries while minimizing computation time.

Contribution. We improve symmetry detection of CNF formulas through the use of *adapted* SAT preprocessing: we reduce the time needed for state-of-the-art symmetry detection and breaking, while also uncovering hidden symmetry not detectable in the unprocessed formulas.

In particular, our main contribution is to provide a theoretical framework that captures how symmetries of formulas before and after applying CNF transformations are related. Among other results, the most important property we consider is whether transformations are "symmetry-preserving": here, we demand that all applicable symmetries of the original formula are also symmetries of the reduced formula (see Section 3 for a formal definition). This means, when applying a symmetry-preserving transformation, we only have to compute symmetries of the reduced formula. This in turn allows us to categorize a selection of quintessential SAT preprocessing techniques (see first column of Table 2). For techniques that turn out to not be symmetry-preserving, we provide tailored restrictions to rectify this.

The novelty of our approach lies in the fact that we exploit *SAT techniques* to improve *symmetry detection*: we depart the conventional view of treating symmetry detection as a black-box, presenting a new application-specific approach to symmetry detection in SAT.

Theoretical Framework. On the theoretical side, the first challenge is to define formal notions describing the effect of CNF transformations on symmetry. We identify three main properties that seem particularly interesting. These properties describe how symmetries behave across the different directions of the transformations. In the following, we want to give a brief intuition for these properties. For the formal definitions see Section 3. Let F denote a CNF formula that is then transformed into F' (using the transformation in question, e.g., applying the unit rule). We demand that $\mathrm{Var}(F') \subseteq \mathrm{Var}(F)$, where Var denotes the set of variables of a formula.

1. If a transformation is *symmetry-preserving (SP)*, then any syntactic symmetry of F is also a syntactic symmetry of F', when restricted to the reduced set of variables $\mathrm{Var}(F')$. When applying the transformation, it therefore suffices to compute symmetries of F'.
   Regarding the particular example in Table 1, we want to remark that this property indeed guarantees that a transformation must either preserve *all* the symmetries of the pigeonhole principle instances, or reduce *all* variables at once, in which case deciding SAT becomes trivial.
2. If a transformation is *weakly symmetry-preserving (WSP)*, then it is possible to restrict syntactic symmetries of F using group-theoretic algorithms to *semantic* symmetries of transformed formulas of F'. This allows for a manual collection of symmetries on reduced formulas, while not guaranteeing an automatic preservation of all the symmetries.
3. If a transformation is *symmetry-lifting (SL)*, then semantic symmetries of F' are also semantic symmetries of the original formula F. This means that the transformation can be used to potentially find more symmetries of the original formula. This seems particularly useful for model counting and enumeration tasks, i.e., applications in which all solutions of the original formula are of interest. In other words, this enables us to potentially find more symmetries of F through the transformation, without actually having to apply the transformation to F.

The results of our theoretical analysis are summarized in Table 2. For techniques that turn out to not be symmetry-preserving, we give new, restricted variants that are.

Practical Evaluation. On the practical side, armed with these new insights, we analyze the effect of symmetry-preserving SAT preprocessing techniques on symmetry detection. We evaluate symmetry detection on instances from the main track of the SAT competition 2021 [2].

Table 2 The table shows whether symmetries are preserved under applying a given CNF transformation (SP and WSP) and whether symmetries of a transformed formula lift back to the original one (SL). [1] indicates that it is assumed that the transformation is applied exhaustively. [2] denotes new, symmetry-preserving variants defined in Section 3.

transformation	SP	WSP	SL
subsumption[1]	✓	✓	✓
self-subsumption	✗	✓	✓
simultaneous self-subsumption[2]	✓	✓	✓
adding learned clauses	✗	✓	✓
unit[1]	✓	✓	✓
pure[1]	✓	✓	✗
blocked clause elimination[1]	✓	✓	✗
bounded variable elimination	✗	✓	✗
symmetric variable elimination[2]	✓	✓	✗

We simplify the formulas using only the aforementioned symmetry-preserving transformations. It then turns out that even when only applying this subset of simplifications, computation times of state-of-the-art symmetry detection and breaking algorithms are decreased substantially. In terms of the type of symmetry detected, we observe that indeed, preprocessing is frequently able to uncover symmetry that is hidden in the unprocessed instance. Moreover, it turns out that unprocessed instances often contain a substantial number of symmetries that exclusively operate on variables that can be reduced away through preprocessing techniques.

2 Syntactic and Semantic Symmetry

We begin by introducing some general notation used in SAT solving. This allows us to discuss the notions of syntactic and semantic symmetries, which we will make use of in this paper.

SAT. Given a variable v, we define the corresponding *literals* v and $\overline{v}$, the latter denoting negation. A SAT instance F is commonly given in *conjunctive normal form* (CNF), i.e., a conjunction of disjunctions of literals, meaning that it is in the form $F = \bigwedge_{i\in\{1,\dots,m\}} \bigvee_{j\in\{1,\dots,k_i\}} l_{i,j}$ where each disjunction of literals is called a *clause*, k_i is the number of literals in clause i and $l_{i,j}$ is the j-th literal in the i-th clause. In this paper, we use the common representation where a formula in CNF is a set of clauses and clauses are sets of literals, without explicitly encoding the conjunctions and disjunctions. The instance above thus becomes $F = \{\{l_{1,1}, \dots, l_{1,k_1}\}, \dots, \{l_{m,1}, \dots, l_{m,k_m}\}\}$.

We denote with $\mathrm{Var}(F) = \{v_1, \dots, v_n\}$ the set of variables of F. We implicitly use the fact that $\overline{\overline{v}} = v$ whenever possible: given a literal $l = \overline{v}$ we may write $\overline{l} = \overline{\overline{v}} = v$. We never distinguish between $\overline{\overline{v}}$ and v. The set of literals of a formula F is denoted by $\mathrm{Lit}(F) := \mathrm{Var}(F) \cup \{\overline{v} \mid v \in \mathrm{Var}(F)\}$.

An *assignment* of F is a function $\sigma : L \to \{\bot, \top\}$ where $L \subseteq \mathrm{Lit}(F)$. An assignment must always act consistently on the literals of a variable, meaning that $\sigma(v) = \top$ if and only if $\sigma(\overline{v}) = \bot$ holds. We call an assignment *complete* whenever $L = \mathrm{Lit}(F)$ and *partial* otherwise. Let $v \in \mathrm{Var}(F)$. Abusing notation, we may write that $l \in \sigma$ whenever $\sigma(l) = \top$, and $l \notin \sigma$ otherwise. Note that if $l \notin \sigma$ and $\overline{l} \notin \sigma$, then $l \notin L$.

We can simplify a formula F with respect to σ: $F[\sigma] := \{C[\sigma] \mid C \in F \wedge \nexists l \in \mathrm{Lit}(F) : (l \in \sigma \wedge l \in C)\}$ with $C[\sigma] := \{l \mid l \in C \wedge \bar{l} \notin \sigma\}$. If $F[\sigma] = \{\}$ we call σ a *satisfying assignment*, whereas if $\{\} \in F[\sigma]$ we call σ a *conflicting assignment*. A complete assignment is either satisfying or conflicting.

Let C_1 and C_2 denote two clauses with $x \in C_1$ and $\overline{x} \in C_2$. We denote with $C_1 \circ_x C_2 = (C_1 \setminus \{x\}) \cup (C_2 \setminus \{\overline{x}\})$ the resolvent of C_1 and C_2 on variable x.

Syntactic Symmetry. We introduce the notion of *syntactic symmetries* of a CNF formula F. Consider bijections $\varphi : \mathrm{Lit}(F) \to \mathrm{Lit}(F)$ mapping literals to literals. A bijection φ naturally lifts to clauses, formulas and assignments, by applying φ element-wise to these objects. Syntactic symmetries have two defining properties: (1) The formula is mapped back to itself, meaning $\varphi(F) = F$ holds. (2) For all $l \in \mathrm{Lit}(F)$ it holds that $\overline{\varphi(l)} = \varphi(\bar{l})$, meaning φ also induces a permutation of the variables. If the above requirements are met, we call φ a syntactic symmetry or automorphism of F. These symmetries form a permutation group under composition. We denote this group of automorphisms of F by $\mathrm{Aut}_{\mathrm{syn}}(F)$.

By modelling a CNF formula as a graph (and vice versa), it can be shown that computing all syntactic symmetries is polynomial-time equivalent to computing the automorphism group of a graph and hence to the graph isomorphism problem [6], also known as GI-complete. Therefore, syntactic symmetries can indeed be computed using tools for graph isomorphism (for modern tools see, e.g., [16, 4, 7]).

Semantic Symmetry. Let us again consider bijections $\varphi : \mathrm{Lit}(F) \to \mathrm{Lit}(F)$. We define the notion of a *semantic symmetry* φ as follows: (1) For all complete variable assignments σ of F it holds that $F[\sigma] = \varphi(F)[\sigma]$. (2) For all $l \in \mathrm{Lit}(F)$ it holds that $\overline{\varphi(l)} = \varphi(\bar{l})$. Note that the second property is the same property as for syntactic symmetries.

We may apply a symmetry φ on a complete assignment σ: we let $\varphi(\sigma)$ denote the complete assignment with $\varphi(\sigma)(v) := \sigma(\varphi(v))$. We can interchange applying any semantic symmetry φ to a formula F or a corresponding complete assignment σ, i.e., $\varphi(F)[\sigma] = F[\varphi(\sigma)]$. The set of all semantic symmetries indeed also forms a group under composition as well:

▶ **Lemma 1.** *The set of all semantic symmetries forms a group under composition.*

Proof. Consider two semantic symmetries φ, φ' of a formula F. We show that $\varphi \circ \varphi'$ is a semantic symmetry of F as well. By definition, for any complete variable assignment σ of F, it holds that $F[\sigma] = \varphi(F)[\sigma] = \varphi'(F)[\sigma]$. But since the definition of semantic symmetry requires that this holds for any σ, it in particular also holds for any $\varphi(\sigma)$. Hence, $F[\sigma] = \varphi'(F)[\sigma] = F[\varphi'(\sigma)] = \varphi(F)[\varphi'(\sigma)] = \varphi'(\varphi(F))[\sigma] = \varphi \circ \varphi'(F)[\sigma]$. ◀

We denote the permutation group of all semantic symmetries of F as $\mathrm{Aut}_{\mathrm{sem}}(F)$.

In order to streamline notation, we introduce the notion of a "symmetric group" on a set of literals L. We want to be able to denote the symmetric group, but with the restriction $\overline{\varphi(l)} = \varphi(\bar{l})$, as is required for syntactic and semantic symmetries. We therefore denote $\mathrm{SymL}(L) := \{\varphi \in \mathrm{Sym}(L) \mid \forall l \in L : \overline{\varphi(l)} = \varphi(\bar{l})\}$ (where $\mathrm{Sym}(L)$ denotes the symmetric group on the domain L). We also give an alternative version for a set of literals V where there is no $l \in V$ with $\bar{l} \in V$, which disallows negation symmetries: $\mathrm{SymV}(V) := \{\varphi' \mid \varphi \in \mathrm{Sym}(V)\}$, where $\varphi'(l) := \varphi(l)$ if $l \in V$, and $\varphi'(l) := \overline{\varphi(\bar{l})}$ otherwise.

It is easy to see that all syntactic symmetries are also semantic symmetries, i.e., $\mathrm{Aut}_{\mathrm{syn}}(F) \subseteq \mathrm{Aut}_{\mathrm{sem}}(F)$, but the reverse does not hold in general [20]. As an example note that for any unsatisfiable formula F, the semantic symmetry group is actually $\mathrm{SymL}(\mathrm{Lit}(F))$. Unsurprisingly, computing semantic symmetries is NP-hard.

Domain Change. Next, we define tools to compare groups on different sets of literals. This is needed whenever we are, for example, comparing the symmetries of a reduced version of a formula to the original formula. To enable this, we define a way to reduce or lift the (finite) domain of a given group. Let Γ be a permutation group on the domain Ω, and let Ω' be a set such that $\Omega \subseteq \Omega'$. We then define $\Gamma \uparrow^{\Omega'} := \{\varphi \uparrow^{\Omega'} \mid \varphi \in \Gamma\}$ where $\varphi \uparrow^{\Omega'} : \Omega' \to \Omega'$ with $\varphi \uparrow^{\Omega'} (x) := \varphi(x)$ if $x \in \Omega$ and $\varphi \uparrow^{\Omega'} (x) := x$ else. Intuitively, we just extend every permutation on Ω with the identity on $\Omega' \setminus \Omega$.

Analogously, to reduce the domain, let $\Omega' \subseteq \Omega$. We first define the *setwise stabilizer* $\Gamma_{\{\Omega'\}} := \{\varphi \mid \varphi \in \Gamma \wedge \varphi(\Omega') = \Omega'\}$. We further define $\Gamma \downarrow_{\Omega'} := \{\varphi|_{\Omega'} \mid \varphi \in \Gamma_{\{\Omega'\}}\}$, i.e., first taking the setwise stabilizer fixing Ω' and subsequently reducing the domain to Ω'. Lastly, we also recall the *pointwise stabilizer* $\Gamma_{(\Omega')} := \{\varphi \in \Gamma \mid \forall p \in \Omega' : \varphi(p) = p\}$, fixing all points of Ω' individually.

3 Transformations and Symmetry

We now focus on the main objective of this paper: we analyze the relationship of CNF transformations and symmetry. In order to do this systematically, we first describe a prototype for all the transformations that we consider. A *transformation* Π defines for a formula F a set of formulas $\Pi(F) = \{F_1, \ldots, F_m\}$, where for each formula F_i it holds that $\mathrm{Var}(F_i) \subseteq \mathrm{Var}(F)$ ($i \in \{1, \ldots, m\}$). We denote $F \xrightarrow{\Pi} F'$ whenever $F' \in \Pi(F)$, i.e., we *apply* Π to F. Naturally, a transformation can be non-deterministic: at any given point, potentially, we could transform F into any of the formulas of $\Pi(F)$. We write $F \xrightarrow{\Pi}{}^{*} F^*$ whenever Π is applied exhaustively, i.e., until $\Pi(F^*) = \emptyset$. We always assume that both the set $\Pi(F)$ as well as the sequence of rule applications is finite.

Next, we define the properties of interest. Firstly, we define when transformations are *symmetry-preserving*. This property formalizes the notion that all symmetries of the original formula that operate on remaining literals are also syntactic symmetries of the reduced formulas.

▶ **Definition 2** (Symmetry-preserving.). *Let Π denote a transformation. We call Π* symmetry-preserving, *if $\mathrm{Aut}_{\mathrm{syn}}(F)_{\{\mathrm{Lit}(F')\}} = \mathrm{Aut}_{\mathrm{syn}}(F)$ and $\mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F')} \leq \mathrm{Aut}_{\mathrm{syn}}(F')$ hold for all CNF formulas F, F' with $F \xrightarrow{\Pi} F'$.*

The symmetry-preserving property indeed comprehensively guarantees preservation of syntactic symmetry on variables of F'. When we apply the simplification, we only need to compute symmetries of F'. Despite the name, on the removed variables $\mathrm{Var}(F) \setminus \mathrm{Var}(F')$, naturally, F' does not contain the symmetries of F: it can be the case that $|\mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F')}| < |\mathrm{Aut}_{\mathrm{syn}}(F)|$. We refer to these symmetries of F as *reducible symmetries*.

Reducible symmetries *solely* permute literals removed by the transformation. All symmetries that operate on remaining literals present in F – even the ones that simultaneously permute remaining literals and removed literals – are guaranteed to be preserved in F'. In fact, there even are no symmetries permuting remaining literals with removed literals to begin with (since $\mathrm{Aut}_{\mathrm{syn}}(F)_{\{\mathrm{Lit}(F')\}} = \mathrm{Aut}_{\mathrm{syn}}(F)$).

We want to mention that whether applying the transformation is "desirable" in the first place is clearly not captured by the symmetry-preserving property. For example, applying blocked clause elimination (which is symmetry-preserving, see Section 3.2) can indeed make formulas more difficult to solve [15]. This in turn obscures the question whether blocked clause elimination should be applied before handling symmetry, since it is not clear whether it should be applied at all. This discussion is however unrelated to our goal: in any case, the property guarantees that if we apply the transformation, we only need to compute symmetries on the remaining formula.

To the contrary, $\mathrm{Aut}_{\mathrm{syn}}(F')$ can also contain additional symmetries, which do not appear in $\mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F')}$. We refer to these symmetries as *hidden symmetries*. Next, we define a weaker notion of the symmetry-preserving property:

▶ **Definition 3** (Weakly Symmetry-preserving.). *Let Π denote a transformation. We call Π* weakly symmetry-preserving, *if $\mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F')} \leq \mathrm{Aut}_{\mathrm{sem}}(F')$ holds for all CNF formulas F, F' with $F \xrightarrow{\Pi} F'$.*

Here, symmetries do not have to be preserved syntactically, only semantically. Also, the setwise stabilizer may remove symmetries beyond reducible symmetries.

Lastly, we also want to reconcile symmetries in the opposite direction of transformations, i.e., how symmetries of reduced formulas relate to symmetries of the original formula. We define the *symmetry-lifting* property, which formalizes the notion that symmetries of the reduced formula are symmetries of the original formula:

▶ **Definition 4** (Symmetry-lifting.). *Let Π denote a transformation. We call Π* symmetry-lifting, *if $\mathrm{Aut}_{\mathrm{sem}}(F') \uparrow^{\mathrm{Lit}(F)} \leq \mathrm{Aut}_{\mathrm{sem}}(F)$ holds for all CNF formulas F, F' with $F \xrightarrow{\Pi} F'$.*

If the transformed formula contains syntactic symmetries that the original one does not, we therefore indeed uncover semantic symmetries of the original formula. In particular, concerning the discussion above, this also makes the additional symmetries found independent of whether we actually want to apply the transformation or not: we can use symmetries in the original formula, without having to apply the transformation

We now show a technical lemma that will aid us in proving that transformations are symmetry-preserving. But in order to state the lemma, we first need to recall and adapt two well-known properties: isomorphism-invariance and confluence.

We say a transformation Π is *isomorphism-invariant*, whenever for all formulas F, for all finite sets V, for all $\varphi \in \mathrm{SymL}(\mathrm{Lit}(F) \cup V \cup \{\overline{v} \mid v \in V\})$ and for all $F' \in \Pi(F)$ it holds that $\varphi(F') \in \Pi(\varphi(F))$. It follows that if Π is isomorphism-invariant and $\varphi \in \mathrm{Aut}_{\mathrm{syn}}(F)$, then $F' \in \Pi(F)$ implies $\varphi(F') \in \Pi(F)$. Intuitively, the property states that transformations are invariant under renaming of variables. The set V represents variable names not present in F.

Furthermore, a transformation Π is *confluent* whenever the following holds for any F: let F_1^* and F_2^* denote formulas where Π was applied exhaustively to F, i.e., $F \xrightarrow{\Pi}{}^* F_1^*$ and $F \xrightarrow{\Pi}{}^* F_2^*$. It then must follow that $F_1^* = F_2^*$. Using these two properties, we can prove our lemma:

▶ **Lemma 5.** *Let Π be a transformation. If Π is confluent and isomorphism-invariant, it follows that the transformation that applies Π exhaustively is symmetry-preserving.*

Proof. Let F^* denote the formula where Π was applied exhaustively to F, i.e., $F \xrightarrow{\Pi}{}^* F^*$. Since Π is confluent, F^* is indeed unique. We need to show that the transformation Π' that defines $F \xrightarrow{\Pi'} F^*$ is symmetry-preserving, i.e., that both $\mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F^*)} \leq \mathrm{Aut}_{\mathrm{syn}}(F^*)$ and $\mathrm{Aut}_{\mathrm{syn}}(F)_{\{\mathrm{Lit}(F^*)\}} = \mathrm{Aut}_{\mathrm{syn}}(F)$ hold.

Let $F \xrightarrow{\Pi} F'$ and $\varphi \in \mathrm{Aut}_{\mathrm{syn}}(F)$. Since φ is a symmetry of F, it holds that $\varphi(F) = F$. Let $F \xrightarrow{\Pi} F_1 \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} F^*$ denote a derivation of F^*. It follows due to isomorphism-invariance that we may also derive $F = \varphi(F) \xrightarrow{\Pi} \varphi(F_1) \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} \varphi(F^*)$. Due to confluence, we know that $F^* = \varphi(F^*)$. But this proves precisely that $\varphi|_{\mathrm{Lit}(F^*)}$ is a syntactic symmetry of F^*, i.e., $\varphi|_{\mathrm{Lit}(F^*)} \in \mathrm{Aut}_{\mathrm{syn}}(F)_{\{\mathrm{Lit}(F^*)\}}$. Since this is true for all $\varphi \in \mathrm{Aut}_{\mathrm{syn}}(F)$, it also follows that $\mathrm{Aut}_{\mathrm{syn}}(F) = \mathrm{Aut}_{\mathrm{syn}}(F)_{\{\mathrm{Lit}(F^*)\}}$, proving the claim. ◀

3.1 Equivalence-Preserving Transformations

Let us first consider transformations that do not manipulate the set of solutions, i.e., with the property that for all $F \xrightarrow{\Pi} F'$ and complete assignments σ of F, $F[\sigma] = F'[\sigma]$ holds. We call such transformations *equivalence-preserving*. We show the following useful property.

▶ **Lemma 6.** *Let* Π *denote an equivalence-preserving transformation. Let* F, F' *be CNF formulas with* $F \xrightarrow{\Pi} F'$. *Then, it holds both that* $\mathrm{Aut}_{\mathrm{sem}}(F) \downarrow_{\mathrm{Lit}(F')} \subseteq \mathrm{Aut}_{\mathrm{sem}}(F')$ *and* $\mathrm{Aut}_{\mathrm{sem}}(F') \uparrow^{\mathrm{Lit}(F)} \subseteq \mathrm{Aut}_{\mathrm{sem}}(F)$.

Proof. Let $F \xrightarrow{\Pi} F'$. The equivalence-preserving property of Π guarantees that for all assignments σ of F, it holds that $F'[\sigma] = F[\sigma]$. Consider any semantic symmetry φ of F' extended by the identity, i.e., $\varphi \in \mathrm{Aut}_{\mathrm{sem}}(F') \uparrow^{\mathrm{Lit}(F)}$. It follows that both $F'[\sigma] = F[\sigma]$ as well as $F'[\varphi(\sigma)] = F[\varphi(\sigma)]$ for all complete assignments σ of F. We can conclude $F'[\varphi(\sigma)] = F'[\sigma] = F[\sigma] = F[\varphi(\sigma)]$, meaning $\varphi \in \mathrm{Aut}_{\mathrm{sem}}(F)$. On the other hand, using the same argument, it also follows that any semantic symmetry $\varphi \in \mathrm{Aut}_{\mathrm{sem}}(F) \downarrow_{\mathrm{Lit}(F')}$ is a semantic symmetry of F'. ◀

Considering our terminology, this proves that equivalence-preserving transformations are always symmetry-lifting and weakly symmetry-preserving.

Subsumption. Let us now consider the *subsumption* rule as our first concrete instance of a transformation rule. We formally define the subsumption rule:

$$\frac{C \in F \quad D \in F \quad C \subset D}{F \setminus \{D\}}.$$

It is easy to see that subsumption is equivalence-preserving, thus, Lemma 6 applies. The question remains whether subsumption is symmetry-preserving. While it is not difficult to show that subsumption is not symmetry-preserving in general, it is so when applied exhaustively and can be concluded using Lemma 5:

▶ **Lemma 7.** *Exhaustive subsumption is symmetry-preserving.*

Proof. We prove the claim by applying Lemma 5. First of all, note that subsumption is indeed confluent [12]. It thus suffices to show that subsumption is isomorphism-invariant, which follows readily: whenever a clause C is subsumed by $D \in F$, then indeed, $\varphi(C)$ is subsumed by $\varphi(D) \in \varphi(F)$ for all $\varphi \in \mathrm{SymL}(\mathrm{Lit}(F))$. ◀

Throughout the paper, we often show that applying a rule exhaustively is symmetry-preserving. We want to remark however that this is usually not a requirement and there are indeed also other ways to apply rules in a symmetry-preserving manner (e.g., in a round-based manner).

Self-Subsumption. Next, we analyze self-subsumption [11]. We define self-subsumption based on *self-subsuming resolution* [11]:

$$\frac{C_1 \cup \{x\} \quad C_2 \cup \{\overline{x}\} \quad C_1 \subset C_2}{C_2}$$

Whenever we can apply self-subsuming resolution to a formula F, we can transform F by removing $C_2 \cup \{\overline{x}\}$ and adding C_2 (essentially removing the literal $\overline{x}$ from C_2). Since this simply constitutes one application of resolution and subsumption, self-subsumption is naturally equivalence-preserving. Thus, Lemma 6 is applicable.

Self-subsumption (also exhaustively, or in conjunction with subsumption) is neither confluent nor symmetry-preserving. Consider the formula

$$(a_1 \vee b_1 \vee \overline{x_1} \vee \overline{y_1}) \wedge (a_1 \vee x_1) \wedge (b_1 \vee \overline{x_1} \vee y_1) \wedge (a_2 \vee b_2 \vee \overline{x_2} \vee \overline{y_2}) \wedge (a_2 \vee x_2) \wedge (b_2 \vee \overline{x_2} \vee y_2).$$

Clearly, a_1 and a_2 are symmetrical. Now, one way to exhaustively apply self-subsumption is

$$(a_1 \vee b_1 \vee \overline{y_1}) \wedge (a_1 \vee x_1) \wedge (b_1 \vee \overline{x_1} \vee y_1) \wedge (a_2 \vee b_2 \vee \overline{x_2}) \wedge (a_2 \vee x_2) \wedge (b_2 \vee \overline{x_2} \vee y_2).$$

In this reduced formula however, a_1 and a_2 are not symmetrical anymore.

Simultaneous Self-Subsumption. We present a way to make self-subsumption symmetry-preserving. We change two aspects of self-subsumption. Firstly, we only apply self-subsumption whenever there is only one unique literal that can be removed from a clause. Secondly, we apply all applicable self-subsumptions simultaneously, in a round-based scheme.

Let $f_F(C_1, C_2, x)$ denote whenever self-subsuming resolution with respect to literal x is applicable to C_1, C_2 in F (i.e., removing the literal x from C_2). We restrict self-subsuming resolution to *unique self-subsuming resolution*:

$$\frac{f_F(C_1, C_2, x) \quad \forall x' \in \mathrm{Lit}(F), \forall C_1' \in F : f_F(C_1', C_2, x') \implies x = x'}{C_2 \setminus \{x\}}.$$

If unique self-subsuming resolution is applicable, then there is only one unique literal that can be removed from C_2 in F using self-subsumption. Hence, we can define a deterministic function $r_F(C) := C'$, that applies unique self-subsuming resolution for C in F, if it is applicable, and defines $C' = C$ otherwise. Based on r, we now apply the rule simultaneously for all clauses in F: we define $R(F) := \{r_F(C) \mid C \in F\}$. This prevents rules initially applicable from deactivating one another. By definition, R is now a deterministic (and thus confluent) transformation.

We record that R is still equivalence-preserving: in every application of R, we first perform all the resolution steps for the individual self-subsuming resolutions, followed by the subsumption steps. We show that simultaneous self-subsumption is symmetry-preserving:

▶ **Lemma 8.** *Simultaneous self-subsumption is symmetry-preserving.*

Proof. Let $\varphi \in \mathrm{Aut}_{\mathrm{syn}}(F)$ and $F \xrightarrow{R} F'$ (slightly abusing notation). We now prove that $\mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F')} \subseteq \mathrm{Aut}_{\mathrm{syn}}(F')$ holds. Let C denote a clause where a literal x can be removed through unique self-subsuming resolution. It immediately follows from syntactic symmetry, that the same holds true for $\varphi(C)$ and $\varphi(x)$.

It remains to be shown that $\mathrm{Aut}_{\mathrm{syn}}(F) = \mathrm{Aut}_{\mathrm{syn}}(F)_{\{\mathrm{Lit}(F')\}}$. Again, note that if a literal x is removed exhaustively from F in F', then by symmetry, the same holds true for $\varphi(x)$. ◀

Learning clauses. Next, we briefly consider learning clauses in a CDCL solver. The transformation adds clauses to F that can be derived from F using resolution. It is easy to see this transformation is indeed equivalence-preserving, and thus Lemma 6 applies.

However, learning clauses is not symmetry-preserving. Consider $(x \vee a) \wedge (\overline{x} \vee a) \wedge (x \vee b) \wedge (\overline{x} \vee b)$. In the formula, a and b are syntactically symmetrical. But if we derive and add, e.g., the clause (a), a and b are not symmetrical anymore.

We could ensure a symmetry-preserving transformation by, e.g., always either adding all symmetrical learned clauses, or only adding clauses on asymmetrical variables (analogous to the technique that we will apply to variable elimination later on in Section 3.2). Also, although impractical, adding learned clauses exhaustively is symmetry-preserving as well.

3.2 Equisatisfiability-Preserving Transformations

We now turn our attention to rules that may alter the set of solutions, i.e., transformations that are only *equisatisfiability-preserving*. Since the reduced formulas change the set of satisfying and conflicting assignments, we need to adjust our expectations for the resulting symmetries. Indeed, Lemma 6 becomes incompatible. This means a transformation might now indeed not be symmetry-lifting, and not weakly symmetry-preserving.

Unit. We analyze the unit literal rule. The unit literal rule requires that there is a unit clause $\{l\} \in F$ and consequently reduces the formula to $F[l \mapsto \top]$.

Since unit can lead to conflicts, we add a *conflict rule*, such that if $\{\} \in F$, we can reduce F to $F' = \{\{\}\}$. In case of a conflict, we thus reduce the formula to a unique conflicting formula. We record that with the conflict rule, the rules are confluent [12]. Indeed, we can now prove the rules to be both symmetry-preserving and symmetry-lifting.

▶ **Lemma 9.** *Exhaustive unit and conflict is symmetry-preserving and symmetry-lifting.*

Proof. *(Symmetry-preserving.)* Since unit and conflict are confluent, it suffices to show that unit and conflict are isomorphism-invariant to apply Lemma 5: whenever a literal l is unit in F and we reduce F to $F[l \mapsto \top]$, then indeed, $\varphi(l)$ is unit in $\varphi(F)$ and we may reduce it to $\varphi(F)[\varphi(l) \mapsto \top] = \varphi(F[l \mapsto \top])$ for all $\varphi \in \mathrm{SymL}(\mathrm{Lit}(F))$.

(Symmetry-lifting.) Let $F \xrightarrow{\Pi} F'$ (where Π denotes unit and conflict). We prove $\mathrm{Aut}_{\mathrm{sem}}(F') \subseteq \mathrm{Aut}_{\mathrm{sem}}(F)$. Let $\varphi \in \mathrm{Aut}_{\mathrm{sem}}(F')$ and σ be a complete assignment of F. Furthermore, let $\mathrm{unit}'(F)$ denote the unit literals assigned in F'. If σ does not assign all literals in $\mathrm{unit}'(F)$ positively, $F[\sigma]$ is unsatisfiable and the permutations of φ clearly have no effect on this. Hence, we may assume that all unit literals are assigned correctly in σ. Since this reduces the formula to F', symmetries of F' now apply. ◀

Pure. The pure literal rule requires that there is a literal $l \in \mathrm{Lit}(F)$ such that for every clause $C \in F$ it holds that $\bar{l} \notin C$. We may then assign $F[l \mapsto \top]$. Formally, we define

$$\frac{l \in \mathrm{Lit}(F) \quad \forall C \in F : \bar{l} \notin C}{F[l \mapsto \top]}.$$

We again consider the transformation that applies pure exhaustively:

▶ **Lemma 10.** *Exhaustive pure is symmetry-preserving.*

Proof. We apply Lemma 5 to show the claim. Note that pure is confluent [12]. It thus suffices to show that pure is isomorphism-invariant, which follows readily: whenever a literal l is pure in F and we reduce F to $F[l \mapsto \top]$, then indeed, $\varphi(l)$ pure in $\varphi(F)$ and we may reduce it to $\varphi(F)[\varphi(l) \mapsto \top] = \varphi(F[l \mapsto \top])$ for all $\varphi \in \mathrm{SymL}(\mathrm{Lit}(F))$. ◀

The question remains whether pure is symmetry-lifting. The technique used to prove unit to be symmetry-lifting strongly depended on the fact that wrongly assigning a literal immediately leads to a conflict. For pure literals, this is simply not true: "wrongly" assigning a pure literal is of course not helpful towards proving a formula satisfiable, however, a formula might still be satisfiable. Moreover, say we have two pure literals l_1 and l_2. It might even be the case that assigning $\overline{l_1}$ makes the formula unsatisfiable, while assigning $\overline{l_2}$ still leaves the formula satisfiable. This means pure literals can indeed be distinct on a semantic level. Indeed, pure is not symmetry-lifting:

▶ **Corollary 11.** *Exhaustive pure is not symmetry-lifting.*

Proof. Consider $F = (x \lor z) \land (\overline{y} \lor z) \land (\overline{y} \lor \overline{z}) \land (y \lor b) \land (y \lor a) \land (y \lor b) \land (\overline{y} \lor \overline{a} \lor \overline{b})$. If we assign $F[x \mapsto \top] = (\overline{y} \lor z) \land (\overline{y} \lor \overline{z}) \land (y \lor a) \land (y \lor b) \land (\overline{y} \lor \overline{a} \lor \overline{b})$ (x is pure), we get the syntactic symmetry $\varphi = (z\overline{z})$. However, if we want φ to be a semantic symmetry of F, the symmetry must also be valid in the case where we assign x differently, i.e., where we have $F' := F[x \mapsto \bot] = (z) \land (y \lor a) \land (\overline{y} \lor z) \land (\overline{y} \lor \overline{z}) \land (y \lor a) \land (y \lor b) \land (\overline{y} \lor \overline{a} \lor \overline{b})$. Clearly F' is still satisfiable, but φ is no semantic symmetry: $F'[z \mapsto \top]$ is satisfiable but $F'[z \mapsto \bot]$ is unsatisfiable. ◀

Blocked Clause Elimination (BCE). Let us now consider blocked clause elimination [13]. A literal $l \in C$ blocks C, if for every clause $C' \in F$ with $\overline{l} \in C'$ it holds that $C \circ_l C'$ is a tautology. A clause is called blocked whenever there is a literal that blocks it. Blocked clause elimination exhaustively removes blocked literals from a formula. Since blocked clause elimination is confluent [13], we can again consider the deterministic transformation that exhaustively applies blocked clause elimination.

We show that blocked clause elimination is not symmetry-lifting.

▶ **Corollary 12.** *Exhaustive blocked clause elimination is not symmetry-lifting.*

Proof. We show the result in a similar fashion to Corollary 11. Consider $F = (x \lor z) \land (\overline{y} \lor z) \land (\overline{y} \lor \overline{z}) \land (y \lor a) \land (y \lor b) \land (\overline{a} \lor \overline{b} \lor \overline{c}) \land (c \lor \overline{y})$. Again, we assign $F[x \mapsto \top] = (\overline{y} \lor z) \land (\overline{y} \lor \overline{z}) \land (y \lor a) \land (y \lor b) \land (\overline{a} \lor \overline{b} \lor \overline{c}) \land (c \lor \overline{y})$ (x is pure), yielding the syntactic symmetry $\varphi = (z\overline{z})$. Also, note that BCE can not be applied further to $F[x \mapsto \top]$. We consider $F[x \mapsto \bot] = (z) \land (\overline{y} \lor z) \land (\overline{y} \lor \overline{z}) \land (y \lor a) \land (y \lor b) \land (\overline{a} \lor \overline{b} \lor \overline{c}) \land (c \lor \overline{y})$, where φ is no semantic symmetry: $F'[z \mapsto \top]$ is satisfiable but $F'[z \mapsto \bot]$ is unsatisfiable. ◀

However, it is indeed symmetry-preserving:

▶ **Lemma 13.** *Exhaustive blocked clause elimination is symmetry-preserving.*

Proof. We again apply Lemma 5 to show the claim. Note that BCE is confluent [13]. It thus suffices to show that BCE is isomorphism-invariant, which follows readily: whenever a literal $l \in C$ blocks C in F, then $\varphi(l) \in \varphi(C)$ blocks $\varphi(C)$ in $\varphi(F)$ for all $\varphi \in \mathrm{SymL}(\mathrm{Lit}(F))$. ◀

Bounded Variable Elimination (BVE). Next, we consider variable elimination [9], which is a crucial component of SAT preprocessors [11]. Let $C_x(F)$ and $C_{\overline{x}}(F)$ denote the clauses containing x (or $\overline{x}$) of F. Variable elimination of a variable x removes all clauses containing x, while adding $R_x(F) := \{C_1 \circ_x C_2 \mid C_1 \in C_x(F), C_2 \in C_{\overline{x}}(F)\}$. Overall, elimination of x produces $F' = F \cup R_x(F) \setminus (C_x(F) \cup C_{\overline{x}}(F))$. Usually, variable elimination is only applied in a *bounded* form (BVE). This means that the rule is only applied whenever F' is "smaller" than F. Different metrics for "smaller" are used. One example is the number of clauses, i.e., simply checking whether $|F'| < |F|$ (see [11]).

Bounded variable elimination is neither symmetry-preserving nor symmetry-lifting, even when applied exhaustively. The corresponding counter-examples can be found in Appendix A. However, we can still show the following.

▶ **Lemma 14.** *Let $F \xrightarrow{VE} F'$. It holds that $\mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F')} \leq \mathrm{Aut}_{\mathrm{syn}}(F')$.*

Proof. Let F' denote the formula where the literals of L were eliminated from F. Let $\varphi \in \mathrm{Aut}_{\mathrm{syn}}(F) \downarrow_{\mathrm{Lit}(F')}$. We prove that $\varphi \in \mathrm{Aut}_{\mathrm{syn}}(F')$.

Consider a clause $C \in F$ where $C \cap L = \emptyset$, i.e., that was not involved in variable elimination. It follows immediately that $\varphi(C) \cap L = \emptyset$. Hence, $C \in F'$ and $\varphi(C) \in F'$.

Now consider $C \in F$ with $C \cap L = L' \neq \emptyset$. It follows that $\varphi(C) \in F$ with $\varphi(C) \cap L = \varphi(L')$. We use the fact that if multiple variables $v_1, \dots, v_m$ are eliminated, the order in which they are eliminated does not change the set of clauses produced. Let $\{C'_1, \dots, C'_k\} \subseteq F'$ denote the clauses that were resolved using C when eliminating L'. We prove that $\{\varphi(C'_1), \dots, \varphi(C'_k)\}$ are the corresponding clauses created from $\varphi(C)$ when eliminating $\varphi(L') \subseteq L$. If $k = 0$, then by symmetry, the same follows for $\varphi(C)$.

We can see that in the first step, C is resolved with $C_1, \dots, C_{k_1}$ each containing v_1 respectively. We can mimic this step on $\varphi(C)$, resolving with $\varphi(C_1), \dots, \varphi(C_{k_1})$ each containing $\varphi(v_1)$ – which all exist by symmetry. In fact, we can mimic every step for $v_1, \dots, v_m$, using $\varphi(v_1), \dots, \varphi(v_m)$. While this does not match the order used by variable elimination, we do know that $\{\varphi(v_1), \dots, \varphi(v_m)\} \subseteq L$, i.e., eventually we eliminate all involved variables.

The clauses involved in the resolution can then again introduce more literals involved in variable elimination, i.e., $(C'_1 \cup \dots \cup C'_k) \cap L = L'' \neq \emptyset$ (but of course $L'' \cap L' = \emptyset$). Due to the setwise stabilizer, this however also happens in a symmetrical fashion for clauses produced from C and $\varphi(C)$, i.e., the corresponding set for $\varphi(C)$ is $\varphi(L'')$. Hence, we can repeat the argument recursively until no further literals of L are introduced, and the claim follows. ◀

Symmetric Variable Elimination. Lemma 14 opens up the opportunity for *symmetric variable elimination*. If we can somehow ensure $\mathrm{Aut}_{\mathrm{syn}}(F)_{\{\mathrm{Lit}(F')\}} = \mathrm{Aut}_{\mathrm{syn}}(F)$, then the technique becomes symmetry-preserving. The idea is that we can do so using any over-approximation of the orbit partition of $\mathrm{Aut}_{\mathrm{syn}}(F)$.

Let σ denote the orbits of $\mathrm{Aut}_{\mathrm{syn}}(F)$, i.e., let $l_1, l_2 \in \mathrm{Lit}(F)$, then $\sigma(l_1) = \sigma(l_2)$ if and only if there exists a $\varphi \in \mathrm{Aut}_{\mathrm{syn}}(F)$ with $\varphi(l_1) = l_2$. We can naturally lift this to variables, i.e., two variables v_1, v_2 are in the same orbit if either $\sigma(v_1) = \sigma(v_2)$ or $\sigma(v_1) = \sigma(\overline{v_2})$.

A step of symmetric variable elimination eliminates a set of variables V, i.e., it performs multiple steps of variable elimination. The crucial requirement is that for each $v \in V$ it additionally holds that $\sigma^{-1}(\sigma(v)) \subseteq V$, i.e., the entire orbit of v must be contained in V. This means that symmetric variable elimination is only allowed to eliminate unions of orbits simultaneously. We can immediately conclude the following from Lemma 14:

▶ **Corollary 15.** *Symmetric variable elimination is symmetry-preserving.*

Following our practical arguments, we should obviously not truly use the orbit partition, since this would defeat the purpose of using preprocessing to speed-up symmetry detection. However, practical graph isomorphism solving is entirely based around fast over-approximations of the orbit partition. The main subroutine of practical graph isomorphism solvers is *color refinement* (or 1-dimensional Weisfeiler-Leman), which precisely delivers a partition with the desired properties [16].

4 A Symmetry-preserving Preprocessor

We implement and test a proof-of-concept, symmetry-preserving preprocessor.

4.1 Implementation and Refined Graph Encoding

The main strategy is to apply the symmetry-preserving CNF simplifications discussed in the previous section. Firstly, the preprocessor performs exhaustive unit, pure, subsumption, simultaneous self-subsumption and blocked clause elimination. We implemented subsumption and simultaneous self-subsumption based on the techniques described for SatELite [11]. Then, we perform bounded variable elimination, again following the heuristics of SatELite. However, we only do this step for variables guaranteed to be asymmetrical. Since we only do this step for asymmetrical variables, it becomes trivial to simultaneously guarantee symmetric variable elimination and bounded variable elimination. We ensure variables to be asymmetrical by performing color refinement on the model graph of the reduced formula. We employ the color refinement implementation of dejavu [3].

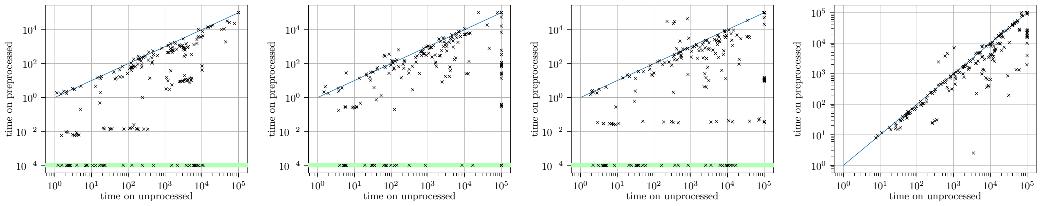
There is one more technique we use to improve the graph encoding. Before computing symmetries, we reduce the graph using the coloring computed by color refinement. We simply remove all asymmetrical parts of the graph, i.e., all nodes which have a discrete color. For readers familiar with graph isomorphism solvers, this strategy might be counter-intuitive. Essentially, the strategy mimics precisely the first step of graph isomorphism solvers. However, in the concrete implementations of solvers, many components are simply tied directly to the number of initial vertices (e.g., incidence-lists or the dense Schreier-Sims algorithm in Traces and dejavu).

4.2 Benchmark Setup

We perform an evaluation on the 400 instances of the main track of the SAT competition 2021 [2], comparing unprocessed versus preprocessed instances. Our evaluation comprises two parts, covering the two main aspects of detecting symmetry: measuring computation time of symmetry detection and breaking algorithms, and measuring the symmetry detected in the instances. The goal of our benchmarks is to provide a proof-of-concept that there can be a tangible benefit in synergizing SAT techniques with symmetry detection.

The code for the preprocessor, solvers and benchmarks are available at [1].

Computation Time. We test symmetry detection and breaking algorithms on unprocessed versus preprocessed instances. The goal is to find out whether symmetry-preserving formula transformations provide a benefit to algorithms concerning symmetry. We test three different graph isomorphism solvers, i.e., saucy [7, 8], Traces [16, 19] and dejavu [3, 4] (4 threads, $< 1\%$ probability of missing a generator). While all state-of-the-art solvers are based on the individualization-refinement framework, the tested solvers indeed represent distinct approaches to symmetry detection: first of all, saucy was specifically designed to be used on graphs stemming from CNF formulas [7]. Traces is well-known to be a highly-competitive general purpose GI solver. It uses a breadth-first approach and excels on hard combinatorial instances [16], while still featuring low-degree techniques typically deemed useful for CNF formulas [7, 8] (e.g., unit literals always have degree 1). dejavu is a recent general-purpose GI solver. It employs novel randomized search strategies [3] and features parallelization [4]. It has no special low-degree vertex or CNF techniques. Furthermore, we measure running time of the symmetry breaker BreakID on the CNF formulas (here, we do not use the special graph encoding of Section 4.1). Internally, BreakID uses saucy.



(a) SAUCY. **(b)** DEJAVU. **(c)** TRACES. **(d)** BREAKID.

Figure 1 Benchmarks for symmetry detection and breaking algorithms comparing unprocessed versus preprocessed SAT competition 2021 instances. Times are given in milliseconds. Instances that were processed to empty graphs (green bar) are shown separately.

Note that even for the "unprocessed" instances, we first removed all redundant clauses and literals from the formula. The timeout is 100 seconds, analogous to the evaluation of [10]. A solver running out of memory also counts as a timeout.

Our working assumption is that most of the proposed preprocessing (or inprocessing) techniques are applied anyway in competitive SAT solvers (see e.g., [5, 18, 21]). Solvers such as CRYPTOMINISAT [21] even enable the user to re-schedule preprocessing and symmetry techniques. Our proposed setup can essentially be realized by carefully scheduling a selection of techniques in an existing preprocessor before symmetry detection, instead of implementing from scratch – with the exception of our adapted symmetry-preserving transformations. But here, the only technique we apply that is unconventional in terms of SAT preprocessors is color refinement.

We thus record and account for the computation time of color refinement separately. In other areas, our preprocessor is indeed poorly optimized compared to state-of-the-art implementations, and we will disregard its computation time.

Detected Symmetry. We analyze how much symmetry is detected both before and after applying preprocessing. To be more precise, we compute the following metrics. Let F be the original formula and F^* the preprocessed one. First of all, we check for *reducible symmetry* that acts exclusively on literals removable through preprocessing, i.e., which we measure in terms of

$$\frac{|\operatorname{Aut}_{\mathrm{syn}}(F)|}{|\operatorname{Aut}_{\mathrm{syn}}(F)_{(\operatorname{Lit}(F^*))}|}.$$

The expression describes the number of symmetries of F which pointwise-stabilize all literals of F^*, i.e., which solely act on the literals $\operatorname{Lit}(F) \setminus \operatorname{Lit}(F^*)$.

Furthermore, we quantify the amount of *hidden symmetry* detectable in the preprocessed but not the unprocessed instance, i.e.,

$$\frac{|\operatorname{Aut}_{\mathrm{syn}}(F^*)| \cdot |\operatorname{Aut}_{\mathrm{syn}}(F)_{(\operatorname{Lit}(F^*))}|}{|\operatorname{Aut}_{\mathrm{syn}}(F)|}.$$

4.3 Benchmark Discussion

For the sake of clarity, in the entire evaluation, we exclude instances that were asymmetrical in *both* the unprocessed and preprocessed instance. This was the case for 192 instances. Note that asymmetrical instances in the set are solved quickly by the graph isomorphism solvers and there are only negligible differences between solvers (e.g. SAUCY took 26*s* for all

Table 3 Benchmarks for symmetry detection and breaking algorithms comparing unprocessed versus preprocessed SAT competition 2021 instances (199 instances). Time taken for preprocessing is not included. The average time given is only for instances that finished within the 100s timeout.

	unprocessed		preprocessed	
Solver	#finished	avg. (finished)	#finished	avg. (finished)
SAUCY	**189**	3.67s	**189**	**1.28s**
DEJAVU	151	3.38s	**188**	**1.25s**
TRACES	168	5.03s	**185**	**2.04s**
BREAKID	170	10.61s	**185**	**8.46s**

asymmetrical instances). Furthermore, our preprocessor failed to process 9 instances, either because it ran out of memory, or took more than 1000 seconds. Thus, in the following, all results are stated for the remaining 199 instances.

Computation Time. Firstly, the preprocessor itself took a total time of 8946 seconds (45s average). The color refinement algorithm for all the instances took 255 seconds (1.28s average). The results for the different symmetry detection and breaking algorithms are summarized in Table 3.

We observe a speed-up for all tested tools across the benchmark suite (see Figure 1): we observe the same number or fewer timeouts for all tools. The overall average symmetry detection times in (on instances that finished) are between 2.47 and 3.38 times faster (1.25 for BREAKID). When the cost for color refinement is accounted for, instances are still solved on average between 1.36 and 1.51 times faster (1.09 for BREAKID) – but especially when additionally considering the reduced number of timeouts, the overhead cost for color refinement is easily amortized.

For the general-purpose graph isomorphism solvers TRACES and DEJAVU, the difference between unprocessed and preprocessed instances seems dramatic (a factor of 2.2 and 4.3 fewer timeouts, respectively). In particular, DEJAVU manages to close most of the performance gap to SAUCY. This is quite surprising, since DEJAVU features no CNF-oriented techniques at all. We record that on the preprocessed instances, DEJAVU even manages to uniquely solve one instance that SAUCY can not within the timeout – albeit while SAUCY does so for two other instances. Overall, on the preprocessed instances the performance of solvers is more comparable and thus seems less dependent on CNF-oriented optimizations.

For the symmetry-breaking tool BREAKID, which internally uses SAUCY, the difference in the number of timeouts is also substantial – in particular much more so than for SAUCY individually (2.07 times fewer timeouts for BREAKID versus the same number of timeouts for SAUCY). Looking more closely at the results reveals that the removal of reducible symmetry speeds up BREAKID: while SAUCY is often able to handle a lot of symmetry more efficiently through the CNF-oriented techniques, the computation time of BREAKID depends directly on the number of generators detected.

We record that in our benchmarks, the computation time taken up by our rudimentary preprocessor is not amortized by the time saved during symmetry detection. For a discussion as to why this is not a major practical concern see Section 4.2.

Detected Symmetry. There are substantial differences in the detected symmetry of the unprocessed and preprocessed instances. Using our preprocessor, we uncovered hidden symmetry in 43 instances (see Figure 2a). We found reducible symmetry in 73 instances. Indeed, these were often very large groups (see Figure 2b) that in many cases even included all symmetries of the respective instance.

(a) Hidden symmetry. **(b)** Reducible symmetry.

Figure 2 The diagrams show instances where non-trivial symmetries of the respective type exist (a value of 1 would mean no non-trivial symmetries existed). The left diagram shows the number of hidden symmetries detected in the instance. The right diagram shows instances where the initial group size was larger than the preprocessed group size, i.e., the number of reducible symmetries in the instance. Both diagrams are cut off at 10^5 for clarity (values go beyond 10^{2000} in both diagrams).

5 Conclusion and Future Work

Unit, pure, subsumption and blocked clause elimination are symmetry-preserving. If these simplifications are to be applied to a formula anyway, then symmetries should be detected and exploited after simplifying the formula. Other techniques, such us adding clauses to the formula derived using resolution, self-subsumption and variable elimination, turn out to not be symmetry-preserving, and can potentially remove symmetry from formulas in an undesirable manner. Going beyond the analysis, for variable elimination and self-subsumption we defined restricted variants, which enable the rules to be applied in a symmetry-preserving manner.

In practice, instances simplified using a symmetry-preserving preprocessor are substantially easier to handle for symmetry detection tools. In fact, the structure of instances changes considerably and techniques previously designed for CNF formulas seem to become less impactful. Most importantly, this opens up the opportunity to tune symmetry detection tools to solve *preprocessed* CNF formulas (e.g., to address the shortcomings we raised with the graph encoding in Section 4.1).

Regarding symmetries, preprocessed instances, due to the symmetry-preserving nature of transformations, are guaranteed to contain at least all applicable symmetries of the unprocessed instances. But indeed, in 21% of the symmetrical instances tested they even contained *more* symmetry. We also found reducible symmetry, that exclusively interacts with literals removable through preprocessing, in 40% of the benchmark instances. Overall, we believe that this motivates an even deeper analysis into how much "exploitable" symmetry is in the instances, and how it can be systematically uncovered. This could also involve tuning and testing with state-of-the-art preprocessing, SAT solvers and symmetry exploitation in the loop: not only to gain a better understanding of the potential effects of the different types of symmetry, but also the interaction between algorithms.

There are even more avenues to expand upon or apply the work in this paper. For example, there are many other preprocessing techniques in the literature (e.g., bounded variable addition) and dynamic techniques (e.g., inprocessing using learned clauses) that could be analyzed.

References

1 EngageS. `https://www.mathematik.tu-darmstadt.de/EngageS`.

2 SAT Competition 2021. `https://satcompetition.github.io/2021/`.

3 Markus Anders and Pascal Schweitzer. Engineering a fast probabilistic isomorphism test. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 73–84. SIAM, 2021. `doi:10.1137/1.9781611976472.6`.

4 Markus Anders and Pascal Schweitzer. Parallel computation of combinatorial symmetries. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ESA.2021.6`.

5 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin 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.

6 James M. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In *AAAI Workshop on Tractable Reasoning*, pages 17–22, 1992.

7 Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 530–534. ACM, 2004. `doi:10.1145/996566.996712`.

8 Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In Limor Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 149–154. ACM, 2008. `doi:10.1145/1391469.1391509`.

9 Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. `doi:10.1145/321033.321034`.

10 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In Nadia Creignou and Daniel 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 104–122. Springer, 2016. `doi:10.1007/978-3-319-40970-2_8`.

11 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby 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. `doi:10.1007/11499107_5`.

12 Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53:127–168, 2015. `doi:10.1613/jair.4694`.

13 Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak 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. `doi:10.1007/978-3-642-12002-2_10`.

14 Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In Ofer Strichman and Stefan 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 113–127. Springer, 2010. `doi:10.1007/978-3-642-14186-7_11`.

15 Oliver Kullmann. On a generalization of extended resolution. *Discret. Appl. Math.*, 96-97:149–176, 1999. `doi:10.1016/S0166-218X(99)00037-2`.

16 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60(0):94–112, 2014. `doi:10.1016/j.jsc.2013.09.003`.

17 Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. Cdclsym: Introducing effective symmetry breaking in SAT solving. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2018. `doi:10.1007/978-3-319-89960-2_6`.

18 Muhammad Osama, Anton Wijs, and Armin Biere. SAT solving with GPU accelerated inprocessing. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2021. `doi:10.1007/978-3-030-72016-2_8`.

19 Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, abs/0804.4881, 2008. arXiv. `arXiv:0804.4881`.

20 Karem A. Sakallah. Symmetry and satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, 2009. `doi:10.3233/978-1-58603-929-5-289`.

21 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. `doi:10.1007/978-3-642-02777-2_24`.

22 Rodrigue Konan Tchinda and Clémentin Tayou Djamégni. Enhancing static symmetry breaking with dynamic symmetry handling in CDCL SAT solvers. *Int. J. Artif. Intell. Tools*, 28(3):1950011:1–1950011:32, 2019. `doi:10.1142/S0218213019500118`.

23 Tevich Treethanyaphong and Athasit Surarerks. Dynamic symmetry breaking in SAT using augmented clauses with a polynomial-time lexicographic pruning. In *2nd European Conference on Electrical Engineering and Computer Science, EECS 2018, Bern, Switzerland, December 20-22, 2018*, pages 242–247. IEEE, 2018. `doi:10.1109/EECS.2018.00053`.

A Counterexamples for Properties of Bounded Variable Elimination

In this section, we show that bounded variable elimination is neither symmetry-preserving nor symmetry-lifting.

Let us first consider whether (exhaustive) BVE is symmetry-preserving. Using the following example, we can show that this is not the case:

$$
\begin{aligned}
&(x \vee a_1) \wedge (x \vee b_1) \wedge (\overline{x} \vee c_1) \wedge (\overline{x} \vee d_1) \wedge \\
&(y \vee a_2) \wedge (y \vee b_2) \wedge (\overline{y} \vee c_2) \wedge (\overline{y} \vee d_2) \wedge \\
&(x \vee A) \wedge (y \vee A) \wedge (\overline{x} \vee A) \wedge (\overline{y} \vee A) \wedge \\
&(a_1 \vee A) \wedge (a_2 \vee A) \wedge (b_1 \vee A) \wedge (b_2 \vee A) \wedge \\
&(c_1 \vee A) \wedge (c_2 \vee A) \wedge (d_1 \vee A) \wedge (d_2 \vee A) \wedge (A) \wedge \\
&(a_1 \vee z_1) \wedge (b_1 \vee z_1) \wedge (c_1 \vee \overline{z_1}) \wedge (d_1 \vee \overline{z_1}) \wedge (a_2 \vee z_2) \wedge (b_2 \vee z_2) \wedge (c_2 \vee \overline{z_2}) \wedge (d_2 \vee \overline{z_2}) \wedge \\
&(\overline{a_1} \vee \overline{b_1} \vee \overline{c_1}) \wedge (\overline{b_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge (\overline{a_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge \\
&(\overline{a_2} \vee \overline{b_2} \vee \overline{c_2}) \wedge (\overline{b_2} \vee \overline{c_2} \vee \overline{d_2}) \wedge (\overline{a_2} \vee \overline{c_2} \vee \overline{d_2})
\end{aligned}
$$

Note that initially x and y are symmetrical. We now eliminate x. For exhaustive BVE to be symmetry-preserving, in any exhaustive application of the rule y must in turn also be removed.

$$
\begin{aligned}
&(y \vee a_2) \wedge (y \vee b_2) \wedge (\overline{y} \vee c_2) \wedge (\overline{y} \vee d_2) \wedge \\
&(a_1 \vee A) \wedge (a_2 \vee A) \wedge (b_1 \vee A) \wedge (b_2 \vee A) \wedge \\
&(c_1 \vee A) \wedge (c_2 \vee A) \wedge (d_1 \vee A) \wedge (d_2 \vee A) \wedge (A) \wedge \\
&(y \vee A) \wedge (\overline{y} \vee A) \wedge \\
&(a_1 \vee z_1) \wedge (b_1 \vee z_1) \wedge (c_1 \vee \overline{z_1}) \wedge (d_1 \vee \overline{z_1}) \wedge (a_2 \vee z_2) \wedge (b_2 \vee z_2) \wedge (c_2 \vee \overline{z_2}) \wedge (d_2 \vee \overline{z_2}) \wedge \\
&(\overline{a_1} \vee \overline{b_1} \vee \overline{c_1}) \wedge (\overline{b_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge (\overline{a_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge \\
&(\overline{a_2} \vee \overline{b_2} \vee \overline{c_2}) \wedge (\overline{b_2} \vee \overline{c_2} \vee \overline{d_2}) \wedge (\overline{a_2} \vee \overline{c_2} \vee \overline{d_2}) \wedge \\
&(a_1 \vee c_1) \wedge (a_1 \vee d_1) \wedge (b_1 \vee c_1) \wedge (b_1 \vee d_1)
\end{aligned}
$$

The resulting equation does indeed have fewer clauses than before. Note that the above already, unsurprisingly, implies that non-exhaustive BVE is not guaranteed to be symmetry-preserving. Next, we choose to eliminate A. Since A is pure, this again reduces the number of clauses and leads to the following formula.

$$
\begin{aligned}
&(y \vee a_2) \wedge (y \vee b_2) \wedge (\overline{y} \vee c_2) \wedge (\overline{y} \vee d_2) \wedge \\
&(a_1 \vee z_1) \wedge (b_1 \vee z_1) \wedge (c_1 \vee \overline{z_1}) \wedge (d_1 \vee \overline{z_1}) \wedge (a_2 \vee z_2) \wedge (b_2 \vee z_2) \wedge (c_2 \vee \overline{z_2}) \wedge (d_2 \vee \overline{z_2}) \wedge \\
&(\overline{a_1} \vee \overline{b_1} \vee \overline{c_1}) \wedge (\overline{b_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge (\overline{a_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge \\
&(\overline{a_2} \vee \overline{b_2} \vee \overline{c_2}) \wedge (\overline{b_2} \vee \overline{c_2} \vee \overline{d_2}) \wedge (\overline{a_2} \vee \overline{c_2} \vee \overline{d_2}) \wedge \\
&(a_1 \vee c_1) \wedge (a_1 \vee d_1) \wedge (b_1 \vee c_1) \wedge (b_1 \vee d_1)
\end{aligned}
$$

Finally, we reduce z_1, which produces no new clauses.

$$
\begin{aligned}
&(y \vee a_2) \wedge (y \vee b_2) \wedge (\overline{y} \vee c_2) \wedge (\overline{y} \vee d_2) \wedge \\
&(a_2 \vee z_2) \wedge (b_2 \vee z_2) \wedge (c_2 \vee \overline{z_2}) \wedge (d_2 \vee \overline{z_2}) \wedge \\
&(\overline{a_1} \vee \overline{b_1} \vee \overline{c_1}) \wedge (\overline{b_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge (\overline{a_1} \vee \overline{c_1} \vee \overline{d_1}) \wedge \\
&(\overline{a_2} \vee \overline{b_2} \vee \overline{c_2}) \wedge (\overline{b_2} \vee \overline{c_2} \vee \overline{d_2}) \wedge (\overline{a_2} \vee \overline{c_2} \vee \overline{d_2}) \wedge \\
&(a_1 \vee c_1) \wedge (a_1 \vee d_1) \wedge (b_1 \vee c_1) \wedge (b_1 \vee d_1)
\end{aligned}
$$

Depending on whether tautologies can be discarded or not, variables of $\{a_1, b_1, c_1, d_1\}$ can be reduced further. Note that this cluster of variables is independent from the rest of the variables. However, in any case, using this variable order, we are now not able to eliminate y (or any of the variables in $\{a_2, b_2, c_2, d_2, z_2\}$), which would be required to satisfy the symmetry-preserving property.

Using a similar example we can show that (exhaustive) BVE is also not symmetry-lifting:

$(y \vee a) \wedge (y \vee b) \wedge (\overline{y} \vee c) \wedge (\overline{y} \vee d) \wedge$
$(a \vee z) \wedge (b \vee z) \wedge (c \vee \overline{z}) \wedge (d \vee \overline{z}) \wedge$
$(\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{b} \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee \overline{c} \vee \overline{d}) \wedge$
$(A \vee a)$

Initially, a and b are not symmetrical due to the clause $(A \vee a)$. Eliminating A only removes the clause $(A \vee a)$. Now, a and b are symmetrical, in particular there is the symmetry that only interchanges a and b. However, considering the case where $[A \mapsto \bot]$, we can see that this is not a semantic symmetry of the original formula: a is a unit literal and hence assigning $[a \mapsto \bot]$ makes the formula unsatisfiable, while assigning $[b \mapsto \bot]$ still leaves the formula satisfiable.