CDCLSym: Introducing Effective Symmetry Breaking in SAT Solving

Hakan Metin, Souheib Baarir, Maximilien Colange, Fabrice Kordon

CDCLSym: Introducing Effective Symmetry Breaking in SAT Solving

Hakan Metin[1(✉)], Souheib Baarir[1,2,3], Maximilien Colange[3], and Fabrice Kordon[1]

[1] Sorbonne Université, CNRS UMR 7606 LIP6, 75005 Paris, France
hakan.metin@lip6.fr
[2] Université Paris Nanterre, Nanterre, France
[3] LRDE, EPITA, Le Kremlin-Bicêtre, France

Abstract. SAT solvers are now widely used to solve a large variety of problems, including formal verification of systems. SAT problems derived from such applications often exhibit symmetry properties that could be exploited to speed up their solving. *Static symmetry breaking* is so far the most popular approach to take advantage of symmetries. It relies on a symmetry preprocessor which augments the initial problem with constraints that force the solver to consider only a few configurations among the many symmetric ones.

This paper presents a new way to handle symmetries, that avoid the main problem of the current static approaches: the prohibitive cost of the preprocessing phase. Our proposal has been implemented in `MiniSym`. Extensive experiments on the benchmarks of last six SAT competitions show that our approach is competitive with the best state-of-the-art static symmetry breaking solutions.

Keywords: Boolean satisfiability · Static symmetry breaking
Dynamic symmetry breaking · Symmetry based reduction

1 Introduction

Nowadays, Boolean satisfiability (SAT) is an active research area finding its applications in many contexts such as planning decision [14], hardware and software verification [3], cryptology [19], computational biology [17], etc. Hence, the development of approaches that could treat increasingly challenging SAT problems has become a focus.

State-of-the-art complete solvers of SAT problems are based on the well-known *Conflict Driven Clauses Learning (CDCL)* algorithm [18], itself inspired from the Davis–Putnam–Logemann–Loveland algorithm [6]. These are complete backtracking based search algorithms that welcome any heuristic/optimisation

The datasets generated during and/or analysed during the current study are available in the figshare repository: https://doi.org/10.6084/m9.figshare.5901025.v1.

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10805, pp. 99–114, 2018.
https://doi.org/10.1007/978-3-319-89960-2_6

pruning of parts of the explored search tree. In this paper, we are interested in exploiting the symmetry properties of SAT problems to perform such a pruning.

Symmetries in SAT Solving. SAT problems often exhibit symmetries[1], and not taking them into account forces solvers to needlessly explore isomorphic parts of the search space.

For example, the "pigeonhole problem" (where n pigeons are put into $n-1$ holes, with the constraint that each pigeon must be in a different hole) is a highly symmetric problem. Indeed, all the pigeons (resp. holes) are swappable without changing the initial problem. Trying to solve it with a standard SAT solver, like MiniSAT [10], turns out to be very time consuming (and even impossible, in reasonable time, for high values of n). Here, such a standard solver ignores the symmetry property of the problem, and then potentially tries all variables combinations; this eventually leads to a combinatorial explosion.

Symmetries of a SAT problem are classically obtained through a reduction to an equivalent graph automorphism problem. Technically, the SAT problem is converted to a colored graph, then it is passed to a tool, like `saucy3` [13] or `bliss` [12], to compute its automorphism group.

A common approach to exploit such symmetries is to pre-compute and enrich the original SAT problem with *symmetry breaking predicates (sbp)*. These added predicates will prevent the solver from visiting equivalent (isomorphic) parts that eventually yield the same results [1,5]. This technique, called *static symmetry breaking*, has been implemented first in the state-of-the-art tool `SHATTER` [2] and then improved in `BREAKID` [8]. However, while giving excellent results on numerous symmetric problems, these approaches still fail to solve some classes of symmetric problems.

Another class of approaches exists, known as *dynamic symmetry breaking* techniques. They intervene directly during the search exploration. It concerns, to mention but a few, the injection of symmetric versions of *learned clauses* [7,21], particular classes of symmetries [20], or speeding up the search by inferring symmetric facts [9]. These approaches succeeded in treating particular and hand crafted problems but, to the best of our knowledge, none of them is competitive face to the *static symmetry breaking* methods.

Drawbacks of the Static-Based Approaches. In the general case, the size of the *sbp* can be exponential in the number of variables of the problem so that they cannot be totally computed. Even in more favorable situations, the size of the generated *sbp* is often too large to be effectively handled by a SAT solver [15]. On the other hand, if only a subset of the symmetries is considered then the resulting search pruning will not be that interesting and its effectiveness depends heavily on the heuristically chosen symmetries [4]. Besides, these approaches are preprocessors, so their combination with other techniques, such as *symmetry*

[1] Roughly speaking, a SAT problem exhibits symmetries when it is possible to swap some variables while keeping the original problem unchanged.

propagation [9], can be very hard. Also, tuning their parameters during the solving turns out to be very difficult. For all these reasons, some classes of SAT problems cannot be solved yet despite exhibiting symmetries.

Proposed Solution. To handle these issues, we propose a new approach that reuses the principles of the static approaches, but operates dynamically: the symmetries are broken during the search process without any pre-generation of the *sbp*. To do so, we elaborate the notions of *symmetry status tracking* and *effective symmetric breaking predicates* (*esbp*).

The approach is implemented using a couple of components: (1) a *Conflict Driven Clauses Learning (CDCL) search engine*; (2) *a symmetry controller*. Roughly speaking, the first component performs the classical search activity on the SAT problem, while the second observes the engine and maintains the status of the symmetries. When the controller detects a situation where the engine is starting to explore a redundant part[2], it orders the engine to operate a backjump. The detection is performed thanks to *symmetry status tracking* and the backjump order is given by a simple injection of an *esbp* computed on the fly.

The main advantage of such an approach is to cope with the heavy (and potentially blocking) pre-generation phase of the static-based approaches, but also offers opportunities to combine with other dynamic-based approaches, like the *symmetry propagation* technique [9]. It also gives more flexibility for adjusting some parameters on the fly. Moreover, the overhead for non symmetric formulas is reduced to the computation time of the graph automorphism.

The extensive evaluation of our approach on the symmetric formulas of the last six SAT contests shows that it outperforms the state-of-the-art techniques, in particular on unsatisfiable instances, which are the hardest class of the problem.

Content of the Paper. The remainder of the paper is organized as follows. Section 2 is dedicated to preliminaries and definitions. Section 3 discusses the details of our CDCLSym algorithm. Section 4 highlights our tooling support and evaluations. Section 5 concludes this work and gives directions for future work.

2 Preliminaries and Definitions

This section introduces some definitions. First, we define the problem of Boolean satisfiability. Then, we introduce the notions of ordering and monotonicity that provide a lexicographical order to assignments. These are central concepts to the definition of a representative assignment.

Finally, we introduce two core notions that are required to define our new algorithm: (i) *Reducer, inactive and active permutation*, and (ii) the *effective symmetry breaking predicates* (*esbp*).

[2] Isomorphic to a part that has been/will be explored.

2.1 Basics on Boolean Satisfiability

A *Boolean variable*, or *propositional variable*, is a variable that has two possible values: true or false (noted $\top$ or $\bot$, respectively). A *literal* l is a propositional variable or its negation. For a given variable x, the positive literal is represented by x and the negative one by $\neg x$. A *clause* ω is a finite disjunction of literals represented equivalently by $\omega = \bigvee_{i=1}^{k} l_i$ or the set of its literals $\omega = \{l_i\}_{i \in [\![1,k]\!]}$. A clause with a single literal is called *unit clause*. A *conjunctive normal form (CNF) formula* φ is a finite conjunction of clauses. A CNF can be either noted $\varphi = \bigwedge_{i=1}^{k} \omega_i$ or $\varphi = \{\omega_i\}_{i \in [\![1,k]\!]}$. We denote $\mathcal{V}_\varphi$ ($\mathcal{L}_\varphi$) the set of variables (literals) used in φ (the index in $\mathcal{V}_\varphi$ and $\mathcal{L}_\varphi$ is usually omitted when clear from context).

For a given formula φ, an *assignment* of the variables of φ is a function $\alpha : \mathcal{V} \mapsto \{\top, \bot\}$. As usual, α is *total*, or *complete*, when all elements of $\mathcal{V}$ have an image by α, otherwise it is *partial*. By abuse of notation, an assignment is often represented by the set of its true literals. The set of all (possibly partial) assignments of $\mathcal{V}$ is noted $Ass(\mathcal{V})$.

The assignment α *satisfies* the clause ω, denoted $\alpha \models \omega$, if $\alpha \cap \omega \neq \emptyset$. Similarly, the assignment α satisfies the propositional formula φ, denoted $\alpha \models \varphi$, if α satisfies all the clauses of φ. Note that a formula may be satisfied by a partial assignment. A formula is said to be *satisfiable* (SAT) if there is at least one assignment that satisfies it; otherwise the formula is *unsatisfiable* (UNSAT).

Example. Let $\varphi = \{\{x_1, x_2, x_3\}, \{x_1, \neg x_2\}, \{\neg x_1, \neg x_2\}\}$ be a formula. φ is satisfied under the assignment $\alpha = \{x_1, \neg x_2\}$ (meaning $\alpha(x_1) = \top$ and $\alpha(x_2) = \bot$) and is reported to be SAT. Note that the assignment α, making φ SAT, does not need to be complete because x_3 is a *don't care variable* with respect to α.

2.2 Ordering and Monotonicity

In order to exploit the symmetry properties of a SAT problem, we need to introduce an ordering relation between the assignments.

Definition 1 (Assignments ordering). *We assume a total order, $\prec$, on $\mathcal{V}$. Given two assignments $(\alpha, \beta) \in Ass(\mathcal{V})^2$, we say that α is strictly smaller than β, noted $\alpha < \beta$, if there exists a variable $v \in \mathcal{V}$ such that:*

- *for all $v' \prec v$, either $v' \in \alpha \cap \beta$ or $\neg v' \in \alpha \cap \beta$.*
- *$\neg v \in \alpha$ and $v \in \beta$.*[3]

Note that $<$ coincides with the lexicographical order on *complete* assignments. Furthermore, the $<$ relation is monotonic as expressed in the following proposition.

Proposition 1 (Monotonicity of assignments ordering). *Let $(\alpha, \alpha', \beta, \beta') \in Ass(\mathcal{V})^4$ be four assignments.*

$$If\ \alpha \subseteq \alpha'\ and\ \beta \subseteq \beta',\ then\ \alpha < \beta \implies \alpha' < \beta'$$

[3] We could have chosen as well $v \in \alpha$ and $\neg v \in \beta$ without loss of generality.

Proof. The proposition follows on directly from Definition 1.

It is worth noting that this last proposition is the key property for the efficient implementation of our algorithm.

2.3 Symmetry Group of a Formula

The group of permutations of $\mathcal{V}$ (i.e. bijections from $\mathcal{V}$ to $\mathcal{V}$) is noted $\mathfrak{S}(\mathcal{V})$. The group $\mathfrak{S}(\mathcal{V})$ naturally acts on the set of literals: for $g \in \mathfrak{S}(\mathcal{V})$ and a literal $\ell \in \mathcal{L}$, $g.\ell = g(\ell)$ if ℓ is a positive literal, $g.\ell = \neg g(\neg\ell)$ if ℓ is a negative literal. The group $\mathfrak{S}(\mathcal{V})$ also acts on (partial) assignments of $\mathcal{V}$ as follows: for $g \in \mathfrak{S}(\mathcal{V})$, $\alpha \in Ass(\mathcal{V})$, $g.\alpha = \{g.\ell \mid \ell \in \alpha\}$. Let φ be a formula, and $g \in \mathfrak{S}(\mathcal{V})$. We say that $g \in \mathfrak{S}(\mathcal{V})$ is a symmetry of φ if for every *complete* assignment α, $\alpha \models \varphi$ if and only if $g.\alpha \models \varphi$. The set of symmetries of φ is noted $S(\varphi) \subseteq \mathfrak{S}(\mathcal{V})$.

Let G be a subgroup of $\mathfrak{S}(\mathcal{V})$. The *orbit of* α *under* G (or simply the *orbit of* α when G is clear from the context) is the set $[\alpha]_G = \{g.\alpha \mid g \in G\}$. The lexicographic leader (*lex-leader* for short) of an orbit $[\alpha]_G$ is defined by $min_<([\alpha]_G)$. This *lex-leader* is unique because the lexicographic order is a total order.

The optimal approach to solve a symmetric SAT problem would be to explore only one assignment per orbit (for instance each *lex-leader*). However, finding the *lex-leader* of an orbit is computationally hard [16].

What we propose here is a best effort approach that tries to eliminate, *dynamically*, the *non lex-leading* assignments with a minimal computation effort. To do so, we first introduce the notions of *reducer*, *inactive* and *active* permutation with respect to an assignment α.

Definition 2 (Reducer, inactive and active permutation). *A permutation g is a* reducer *of an assignment α if $g.\alpha < \alpha$ (hence α cannot be the lex-leader of its orbit. g reduces it and all its extensions). g is* inactive *on α when $\alpha < g.\alpha$ (so, g cannot reduce α and all the extensions). A symmetry is said to be* active *with respect to α when it is neither inactive nor a reducer of α.*

Proposition 2 restates this definition in terms of variables and is the basis of an efficient algorithm to keep track of the status of a permutation during the solving. Let us, first, recall that the *support*, $\mathcal{V}_g$, of a permutation g is the set $\{v \in \mathcal{V} \mid g(v) \neq v\}$.

Proposition 2. *Let $\alpha \in Ass(\mathcal{V})$ be an assignment, $g \in \mathfrak{S}(\mathcal{V})$ a permutation and $\mathcal{V}_g \subseteq \mathcal{V}$ the support of g. We say that g is:*

1. a reducer of α *if there exists a variable $v \in \mathcal{V}_g$ such that:*
   - $\forall\ v' \in \mathcal{V}_g$, *s. t.* $v' \prec v$, *either* $\{v', g^{-1}(v')\} \subseteq \alpha$ *or* $\{\neg v', \neg g^{-1}(v')\} \subseteq \alpha$,
   - $\{v, \neg g^{-1}(v)\} \subseteq \alpha$;
2. inactive *on α if there exists a variable $v \in \mathcal{V}_g$ such that:*
   - $\forall\ v' \in \mathcal{V}_g$, *s. t.* $v' \prec v$, *either* $\{v', g^{-1}(v')\} \subseteq \alpha$ *or* $\{\neg v', \neg g^{-1}(v')\} \subseteq \alpha$,
   - $\{\neg v, g^{-1}(v)\} \subseteq \alpha$;
3. active *on α, otherwise.*

When g is a *reducer* of α we can define a predicate that contradicts α yet preserves the satisfiability of the formula. Such a predicate will be used to discard α, and all its extensions, from a further visit and hence pruning the search tree.

Definition 3 (Effective Symmetry Breaking Predicate). *Let $\alpha \in Ass(\mathcal{V})$, and $g \in \mathfrak{S}(\mathcal{V})$. We say that the formula ψ is an effective symmetry breaking predicate (esbp for short) for α under g if:*

$$\alpha \not\models \psi \text{ and for all } \beta \in Ass(\mathcal{V}), \beta \not\models \psi \Rightarrow g.\beta < \beta$$

The next definition gives a way to obtain such an effective symmetry-breaking predicate from an assignment and a reducer.

Definition 4 (A construction of an *esbp*). *Let φ be a formula. Let g be a symmetry of φ that reduces an assignment α. Let v be the variable whose existence is given by item 1. in Proposition 2. Let $U = \{v', \neg v' \mid v' \in \mathcal{V}_g \text{ and } v' \preceq v\}$. We define $\eta(\alpha, g)$ as $(U \cup g^{-1}.U) \setminus \alpha$.*

Example. Let us consider $\mathcal{V} = \{x_1, x_2, x_3, x_4, x_5\}$, $g = (x_1\, x_3)(x_2\, x_4)$, and a partial assignment $\alpha = \{x_1, x_2, x_3, \neg x_4\}$. Then, $g.\alpha = \{x_1, \neg x_2, x_3, x_4\}$ and $v = x_2$. So, $U = \{x_1, \neg x_1, x_2, \neg x_2\}$ and $g^{-1}.U = \{x_3, \neg x_3, x_4, \neg x_4\}$ and we can deduce than $\eta(\alpha, g) = (U \cup g^{-1}.U) \setminus \alpha = \{\neg x_1, \neg x_2, \neg x_3, x_4\}$.

Proposition 3. *$\eta(\alpha, g)$ is an effective symmetry-breaking predicate.*

Proof. It is immediate that $\alpha \not\models \eta(\alpha, g)$.

Let $\beta \in Ass(\mathcal{V})$ such that $\beta \wedge \eta(\alpha, g)$ is UNSAT. We denote a α' and β' as the restrictions of α and β to the variables in $\{v' \in \mathcal{V}_g \mid v' \preceq v\}$. Since $\beta \wedge \eta(\alpha, g)$ is UNSAT, $\alpha' = \beta'$. But $g.\alpha' < \alpha'$, and $g.\beta' < \beta'$. By monotonicity of $<$, we thus also have $g.\beta < \beta$.

It is important to observe that the notion of *ebsp* is a refinement of the classical concept of *sbp* defined in [2]. In particular, like *sbp*, *esbp* preserve satisfiability.

Theorem 1 (Satisfiability preservation). *Let φ be a formula and ψ an ebsp for some assignment α under $g \in S(\varphi)$. Then,*

$$\varphi \text{ and } \varphi \wedge \psi \text{ are equi-satisfiable.}$$

Proof. If $\varphi \wedge \psi$ is SAT then φ is trivially SAT. If φ is SAT, then there is some assignment β that satisfies φ. Without loss of generality, β can be chosen to be the lex-leader of its orbit under $S(\varphi)$. Thus, g does not reduce β, which implies that $\beta \models \psi$.

3 CDCLSym Algorithm

This section describes how to augment the state-of-the-art CDCL algorithm with the aforementioned concepts to develop an efficient symmetry-guided SAT solving algorithm. We first recall how the CDCL algorithm works. We then explain how to extend it with a *symmetry controller* component which guides the behavior of CDCL algorithm depending on the status of symmetries.

3.1 Classical CDCL

A Conflict-Driven Clause Learning (CDCL) algorithm is depicted in Algorithm 1. The parts in red (grey in B&W printings) should be ignored for the moment.

The algorithm walks a binary search tree. It first applies unit propagation to the formula φ for the current assignment α (line 4). A conflict at level 0 indicates that the formula is not satisfiable, and the algorithm reports it (lines 8–9). If a conflict is detected, it is analyzed, which provides a *conflict clause* explaining the reason for the conflict (line 11). This clause is learnt (line 14), as it does not change the satisfiability of φ, and avoids encountering a conflict with the same causes in the future. The analysis is completed by the computation of a backjump point to which the algorithm backtracks (line 15). Finally, if no conflict appears, the algorithm chooses a new decision literal (line 18–19). The above steps are repeated until the satisfiability status of the formula is determined.

It is out of the scope of this paper to detail the existing variations for the conflict analysis and for the decision heuristic.

```
1  function CDCLSym(φ: CNF formula, SymController: symmetry controller)
     returns ⊤ if φ is SAT and ⊥ otherwise
2    dl ← 0 ;                                          // Current decision level
3    while not all variables are assigned do
4      isConflict ← unitPropagation();
5      SymController.updateAssign(currentAssignment());
6      isReduced ← SymController.isNotLexLeader(currentAssignment());
7      if isConflict || isReduced then
8        if dl == 0 then
9          return ⊥;                                          // φ is UNSAT
10       if isConflict then
11         ω ← analyzeConflict();
12       else
13         ω ← SymController.generateEsbp(currentAssignment());
14       addLearntClause(ω);
15       dl ← backjumpAndRestartPolicies();
16       SymController.updateCancel(currentAssignment());
17     else
18       assignDecisionLiteral();
19       dl ← dl + 1;
20   return ⊤;                                             // φ is SAT
```

Algorithm 1. The CDCLSym SAT Solving Algorithm.

3.2 Symmetry-Guided Search

As explained earlier, the main problem of the static approaches is that they generate many *sbp* that are not effective in the solving (size of the generated formulas, overburden of the unit propagation procedure, etc.).

The idea we bring is to break symmetries *on the fly*: when the current partial assignment can not be a prefix of a *lex-leader* (of an orbit), an *esbp* (see Definition 3) that prunes this forbidden assignment and all its extensions is generated.

We implement this approach using two components that communicate with each other: the SAT-solving engine itself, and a *symmetry controller*. The symmetry controller is initially given a set of symmetries G[4]. It observes the behavior of the SAT engine and updates its internal data according to the current assignment, to keep track of the status of the symmetries. This observation is *incremental*: whenever a literal is assigned or cancelled, the symmetry controller updates the status of all the symmetries. This corresponds to lines 5 and 16 of Algorithm 1. When the controller detects that the current assignment can not be a *lex-leader* (line 6), it generates the corresponding *esbp* (line 13).

In the remainder of this section, we detail the functions composing the symmetry controller.

Symmetries Status Tracking. The `updateAssign`, `updateCancel` and `isNotLexLeader` functions (see Algorithm 2) track the status of symmetries based on Proposition 2; there, resides the core of our algorithm.

All these functions rely on the pt structure: a map of variables indexed by permutations. Initially, $pt[g] = \min(\mathcal{V}_g)$ for all $g \in G$ and all permutations are marked *active*.

For each permutation, g, the symmetry controller keeps track of the smallest variable $pt[g]$ in the support of g such that $pt[g]$ and $g^{-1}(pt[g])$ do not have the same value in the current assignment. If one of the two variables is not assigned, they are considered not to have the same value.

When new literals are assigned, only active symmetries need to have their $pt[g]$ updated (line 2). This update is done thanks to a while loop (lines 4–5).

When literals are cancelled, we need to update the status of symmetries for which some variable v before $pt[g]$, or $g^{-1}(v)$, becomes unassigned (lines 9–10). Symmetries that were inactive may be reactivated (line 11).

The current assignment is not a *lex-leader* if some symmetry g is a reducer. This is detected by comparing the value of $pt[g]$ with the value of $g^{-1}(pt[g])$ (line 16). The function `isNotLexLeader` also marks symmetries as *inactive* when appropriate (lines 18–19).

Generation of the *esbp*. When the current assignment cannot be a *lex-leader*, some symmetry g is a reducer. The function `generateEsbp` computes the

[4] The generators of the group of symmetries.

```
1  function updateAssign(α: assignment)
2      foreach active g ∈ G do
3          v ← pt[g];
4          while {v, g^{-1}(v)} ⊆ α or {¬v, ¬g^{-1}(v)} ⊆ α do
5              v ← next variable in V_g;
6          pt[g] ← v
7  function updateCancel(α: assignment)
8      foreach g ∈ G do
9          u ← min{v ∈ V_g | {v, ¬v} ∩ α = ∅ or {g^{-1}(v), ¬g^{-1}(v)} ∩ α = ∅};
10         if u ⪯ pt[g] then
11             mark g as active;
12             pt[g] ← u;
13 function isNotLexLeader(α: assignment)
14     foreach active g ∈ G do
15         v ← pt[g];
16         if {v, ¬g^{-1}(v)} ⊆ α then
17             return ⊤;                                   // g is a reducer
18         if {¬v, g^{-1}(v)} ⊆ α then
19             mark g as inactive ;      // g can't reduce α or its extentions
20     return ⊥
21 function generateEsbp(α: assignment) returns ω: generated esbp
22     ω ← {};
23     g ← the reducer of α detected in isNotLexLeader;
24     v ← min(V_g);
25     u ← pt[g];
26     while u ≠ v do
27         if v ∈ α then ω ← ω ∪ {¬v} else ω ← ω ∪ {v};
28         if g^{-1}(v) ∈ α then ω ← ω ∪ {¬g^{-1}(v)} else ω ← ω ∪ {g^{-1}(v)};
29         v ← next variable in V_g
30     ω ← ω ∪ {¬v, g^{-1}(v)};
31     return ω
```

Algorithm 2. The functions keeping track of the status of the symmetries and generating the *esbp*.

$\eta(\alpha, g)$ defined in Definition 4, which is an effective symmetry-breaking predicate by Proposition 3. This will prevent the SAT engine to explore further the current partial assignment.

3.3 *Lex-leader* Forcing

Our algorithm prevents as much as possible the solver from visiting *non lex-leaders* assignments. To do so, we propose an additional heuristic that delays the visit of *non lex-leaders* partial assignments.

Let us consider a permutation g and an assignment α. Assume there exists a variable $v \in \mathcal{V}_g$, with, for all $v' \in \mathcal{V}_g$, such that $v' \prec v$, either $\{v', g^{-1}(v')\} \subseteq \alpha$ or $\{\neg v', \neg g^{-1}(v')\} \subseteq \alpha$ and $v \in \alpha$. Let $\alpha' = \alpha \cup \{\neg g^{-1}(v)\}$. Then g is a reducer of α', which would generate $\eta(\alpha', g)$ (Proposition 2 and Definition 4).

A way to prevent α from becoming a *non lex-leader* is to force the literal $g^{-1}(v)$ into α. This can be easily done by learning $\eta(\alpha', g)$ when the current assignment is α. The same reasoning holds when $\neg g^{-1}(v) \in \alpha$ and $v \notin \alpha$.

3.4 Illustrative Example

Let us illustrate the previous concepts and algorithms on a simple example. Let $\mathcal{V} = \{v_1 \prec v_2 \prec v_3 \prec v_4 \prec v_5 \prec v_6\}$, and a set of symmetries $G = \{g_1 = (v_1 v_5 v_3)(v_2 v_4), g_2 = (v_1 v_6)(v_4 v_5)\}$ (written in cycle notation). Their respective supports are, $\mathcal{V}_{g_1} = \{v_1, v_2, v_3, v_4, v_5\}$ and $\mathcal{V}_{g_2} = \{v_1, v_4, v_5, v_6\}$.

On the assignment $\alpha = \emptyset$, both permutations are active and $pt[g_1] = pt[g_2] = v_1$. When the solver updates the assignment to $\alpha = \{v_6\}$, both permutations remain active and $pt[g_1] = pt[g_2] = v_1$. On the assignment $\alpha = \{v_6, v_1\}$, the symmetry controller updates $pt[g_2]$ to v_5, while $pt[g_1]$ remains unchanged. On the assignment $\alpha = \{v_6, v_1, \neg v_3\}$, $g_1.\alpha = \{v_6, v_5, \neg v_1\}$, which is smaller than α (because $v_1 \in \alpha$ and $\neg v_1 \in g.\alpha$): g_1 is a reducer of α. The symmetry controller then generates the corresponding *esbp* $\omega = \{\neg v_1, v_3\}$. Alternatively, when *lex-leader* forcing is active, from the assignment $\alpha = \{v_6, v_1\}$, the symmetry controller could force the value of the variable v_3, by learning the same *esbp* $\omega = \{\neg v_1, v_3\}$.

4 Implementation and Evaluation

In this section, we first highlight some details on our implementation of the symmetry controller. Then, we experimentally assess the performance of our algorithm against three other state-of-the-art tools.

4.1 cosy: An Efficient Implementation of the Symmetry Controller

We have implemented our method in a C++ library called cosy (1630 LoC). It implements a symmetry controller as described in the previous section, and can be interfaced with virtually any CDCL SAT solver. cosy is released under GPL v3 licence and is available at https://github.com/lip6/cosy.

Heuristics and Options. Let us recall that finding the optimal ordering of variables (with respect to the exploitation of symmetries) is NP-hard [15], so the choice for this ordering is heuristic. cosy offers several possibilities to define this ordering:

- a naive ordering, where variables are ordered by the lexicographic order of their names;

- an ordering based on occurrences, where variables are sorted according to the number of times they occur in the input formula. The lexicographic order of variables names is used for those having the same number of occurrences;
- an ordering based on symmetries, where variables belonging to the same orbit (under the given set of symmetries) are grouped together. Orbit are ordered by their numbers of occurrences.

The ordering of assignments we use in this paper orders negative literals before positive ones (thus, $\{\neg v\} < \{v\}$), but using the converse ordering does not change the overall method. However, it can impact the performance of the solver on some instances, so that it is an option of the library.

All the symmetries we used for the presentation of our approach are permutations of variables. Our method straightforwardly extends to permutations of literals, also known as *value permutations* [4]. Another option allows to activate the *lex-leader* forcing described in Sect. 3.3.

Integration in `MiniSAT`. We show how to integrate `cosy` to an existing solver, through example of `MiniSAT` [10].

First, we need an adapter that allows the communication between the solver and `cosy` (30 LoC). Then, we adapt Algorithm 1 to the different methods and functions of `MiniSAT`. In particular, the function `updateAssign` is moved into the `uncheckEnqueue` function of `MiniSAT` (2 LoC). The `updateCancel` function is moved to the `cancelUntil` function of `MiniSAT` that performs the backjumps (2 LoC). The `isNotLexLeader` and `generateEsbp` functions are integrated in the `propagate` function of `MiniSAT` (30 LoC). This is to keep track of the assignments as soon as they occur, then the *esbp* is produced as soon as an assignment is identified as not being *lex-leader*. Initialization issues are located in the main function of `MiniSAT` (15 LoC).

The integration of `cosy` increases `MiniSAT` code by 3%.

4.2 Evaluation

This section presents the evaluation of our approach. All experiments have been performed with our modified `MiniSAT` called `MiniSym`. The symmetries of the SAT problem instances have been computed by two different state-of-the-art tools `saucy3` [13] and `bliss` [12]. For a given group of symmetries, the first tool generates less permutations to represent the group than the second one, but it is slower than the other one.

We selected from the last six editions of the SAT contests [11], the CNF instances for which `bliss` finds at least 2% of the variables are involved in some symmetries that could be computed in at most 1000 s of CPU time. We obtained a total of 1350 symmetric instances (discarding repetitions) out of 3700 instances in total.

All experiments have been conducted using the following conditions: each solver has been run once on each instance, with a time-out of 5000 s (including

the execution time of the symmetries generation except for `MiniSAT`) and limited to 8 GB of memory. Experiments were executed on a computer with an Intel Xeon X7460 2.66 GHz featuring 24 cores and 128 GB of memory, running a Linux 4.4.13, along with g++ compiler version 6.3.

We compare `MiniSym` using the occurrence order, value symmetries, and without *lex-leader* forcing, against:

- `MiniSAT`, as the reference solver without symmetry handling [10];
- `Shatter`, a symmetry breaking preprocessor described in [2], coupled with the `MiniSAT` SAT engine;
- `breakID`, another symmetry breaking preprocessor, described in [8], also coupled with the `MiniSAT` SAT engine.

Each SAT solution was successfully checked against the initial CNF. For UNSAT situations, there is no way to provide an UNSAT certificate in presence of symmetries. Nevertheless, we checked our results were also computed by the other measured tools. Unfortunately, out of the 1350 benchmarked formulas, we have no proof or evidence for the 15 UNSAT formulas computed by `MiniSym` only.

Results are presented in Tables 1, 2, and 3. We report the number of instances solved within the time and memory limits for each solver and category. We separate the UNSAT instances (Table 1) from the SAT ones (Table 2). Besides the reference with no symmetry (column `MiniSAT`), we have compared the performance of the three tools when using symmetries computed by `saucy3` (see Tables 1a and 2b), and `bliss` (see Tables 1a and 2b). Rows correspond to groups of instances: from each edition of the SAT contest, and when possible, we separated applicative instances (app$\langle x \rangle$ where $\langle x \rangle$ indicates the year) from hard combinatorial ones (hard$\langle x \rangle$). This separation was not possible for the editions 2015 and 2017 (all2015 and all2017). The total number of instances for each bench is indicated between parentheses. For each row, the cells corresponding to the tools solving the most instances (within time and memory limits) are typeset in bold and greyed out. Table 3 shows the cumulative and average PAR-2 times of the evaluated tools.

Table 1. Comparison of different approaches on the UNSAT instances of the benchmarks of the six last editions of the SAT competition.

Benchmark	`MiniSAT`	`Shatter`	`BreakID`	`MiniSym`
app2016 (134)	18	19	**20**	17
app2014 (161)	23	23	22	**24**
app2013 (145)	6	8	8	**10**
app2012 (367)	115	115	**120**	**120**
hard2016 (128)	8	17	**50**	42
hard2014 (107)	9	24	**30**	29
hard2013 (121)	12	24	**48**	29
hard2012 (289)	86	84	88	**93**
all2017 (124)	8	14	**15**	14
all2015 (65)	9	8	8	**10**
TOTAL (no dup)	261	302	**371**	345

(a) With `saucy3`

Benchmark	`MiniSAT`	`Shatter`	`BreakID`	`MiniSym`
app2016 (134)	18	**21**	18	19
app2014 (161)	23	21	20	**24**
app2013 (145)	6	7	10	**11**
app2012 (367)	115	106	114	**123**
hard2016 (128)	8	11	**79**	77
hard2014 (107)	9	45	40	**53**
hard2013 (121)	12	51	**56**	54
hard2012 (289)	86	69	90	**93**
all2017 (124)	8	14	**15**	**15**
all2015 (65)	**9**	7	8	8
TOTAL (no dup)	261	324	415	**439**

(b) With `bliss`

Table 2. Comparison of different approaches on the SAT instances of the benchmarks of the six last editions of the SAT competition.

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
app2016 (134)	20	**22**	21	20
app2014 (161)	**24**	**24**	**24**	22
app2013 (145)	34	35	35	**43**
app2012 (367)	121	112	119	**126**
hard2016 (128)	**0**	**0**	**0**	**0**
hard2014 (107)	14	**17**	**17**	14
hard2013 (121)	23	23	**24**	22
hard2012 (289)	135	141	**143**	138
all2017 (124)	23	20	26	**27**
all2015 (65)	**7**	5	**7**	6
TOTAL (no dup)	325	323	**337**	335

(a) With saucy3

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
app2016 (134)	20	20	**22**	20
app2014 (161)	**24**	**24**	23	22
app2013 (145)	**34**	32	30	33
app2012 (367)	**121**	112	120	118
hard2016 (128)	**0**	**0**	**0**	**0**
hard2014 (107)	14	14	17	**18**
hard2013 (121)	23	24	**26**	25
hard2012 (289)	135	134	141	**142**
all2017 (124)	23	25	26	**29**
all2015 (65)	**7**	5	6	6
TOTAL (no dup)	325	316	334	**336**

(b) With bliss

Table 3. Comparison of PAR-2 times (in seconds) of the benchmarks on the six last editions of the SAT competition.

Solver	PAR-2 sum	PAR-2 avg
MiniSAT	8 074 348	5 981
Shatter	7 770 434	5 756
BreakID	6 909 999	5 119
MiniSym	7 229 700	5 355

(a) With saucy3

Solver	PAR-2 sum	PAR-2 avg
MiniSAT	8 074 348	5 981
Shatter	7 517 556	5 569
BreakID	6 444 954	4 774
MiniSym	6 245 448	4 626

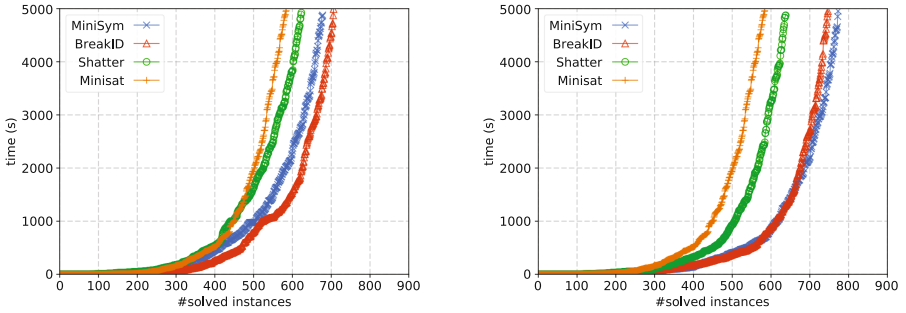
(b) With bliss

We observe that MiniSym with saucy3 solves the most instances in only half of the UNSAT categories. However, with bliss, MiniSym solves the most instances in all but four of the UNSAT categories; it then also solves the highest number of instances among its competitors. This shows the interest of our approach for UNSAT instances. Since symmetries are used to reduce the search space, we were expecting that it will bring the most performance gain for UNSAT instances.

The situation for SAT instances is more mitigated (Table 2), especially when using saucy3. Again, this is not very surprising: our method may cut the exploration of a satisfying assignment because it is not a *lex-leader*. This delays the discovery of a satisfying assignment. The other tools suffer less from such a delay, because they rely on symmetry breaking predicates generated in a pre-processing step. Also, when seeing the global results of MiniSAT, we can globally state that the use of symmetries in the case of satisfiable instances only offers a marginal improvement.

We observe that performances our tool are better with bliss than with saucy3 (see Fig. 1). We explain it as follows: saucy3 is known to compute fewer generators for the group of symmetries than bliss. Since, the larger the symmetries set is, the earlier the detection of an *evidence* that an assignment is not a *lex-leader* will be, we generate less symmetry-breaking predicates (only the effective ones). This is shown in Table 4; MiniSym generates an order of magnitude fewer predicates than breakID.

We also conducted experiments on highly symmetrical instances (all variables are involved in symmetries), whose results are presented in Table 5.



(a) with `saucy3` (b) with `bliss`

Fig. 1. Cactus plot total number of instances

Table 4. Comparison of the number of generated SBPs each time `breakID` and `MiniSym` both compute a verdict (number of verdicts between parentheses).

Number of SBPs	`BreakID`	`MiniSym`
UNSAT (316)	12 088 433	1 579 623
SAT (312)	13 839 689	359 352

(a) With `saucy3`

Number of SBPs	`BreakID`	`MiniSym`
UNSAT (399)	2 576 349	913 339
SAT (320)	12 179 513	457 452

(b) With `bliss`

Table 5. Comparison of the tools on 99 highly symmetric UNSAT problems.

Benchmark	`MiniSAT`	`Shatter`	`breakID`	`MiniSym`
battleship(6)	**5**	**5**	**5**	**5**
chnl(6)	4	**6**	**6**	**6**
clqcolor(10)	3	4	5	**6**
fpga(10)	6	**10**	**10**	**10**
hole(24)	10	12	**23**	11
hole shuffle(12)	1	2	**12**	3
urq(6)	1	2	**6**	2
xorchain(2)	1	1	**2**	**2**
TOTAL	31	42	**69**	45

(a) With `saucy3`

Benchmark	`MiniSAT`	`Shatter`	`breakID`	`MiniSym`
battleship(6)	5	5	5	**6**
chnl(6)	4	**6**	**6**	**6**
clqcolor(10)	3	5	8	**10**
fpga(10)	6	**10**	**10**	**10**
hole(24)	10	**24**	**24**	23
hole shuffle(12)	1	3	**7**	4
urq(6)	1	2	**6**	5
xorchain(2)	1	1	**2**	**2**
TOTAL	31	56	**68**	66

(b) With `bliss`

The performance of `breakID` on this benchmark is explained by a specific optimization for the *total symmetry groups* that are found in these examples, that is neither implemented in `Shatter` nor in `MiniSym`. However, the difference between `breakID` and `MiniSym` is rather thin when using `bliss`. Our tool still outperforms `Shatter` on this benchmark.

5 Conclusion

This paper presented an approach dealing with the symmetries when they appear in SAT problems. It borrows from the state-of-the-art static-based approaches their basic principle, i.e., the adding of *symmetry breaking predicates* to the

original problem, but performed in an incremental and dynamic way. This is possible thanks to the *dynamic tracking of symmetries status* and *on-the-fly generation of effective symmetry breaking predicates*.

Our approach outperforms other state-of-the-art static methods, as shown by an extensive evaluation on the symmetric problems gathered from the last six SAT competitions.

This approach is implemented in the C++ library called cosy. It is an off-the-shelf component that can be interfaced with virtually any CDCL SAT solver. cosy is released under GPL licence and is available at https://github.com/lip6/cosy.

We now plan to focus on combining our approach with *symmetry propagation* [9]. It seems that such a combination could be implemented thanks to minor changes on our algorithm. This would allow to integrate the acceleration mechanisms provided by the *symmetry propagation*, therefore obtaining a better pruning of the search three.

Another track for future work, is to evaluate the possibility of changing the order of variables dynamically: for example, following the order used by the solver when it chooses its decision variables.

References

1. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Solving difficult instances of Boolean satisfiability in the presence of symmetry. IEEE Trans. CAD Integr. Circuits Syst. **22**(9), 1117–1137 (2003)
2. Aloul, F., Sakallah, K., Markov, I.: Efficient symmetry breaking for Boolean satisfiability. IEEE Trans. Comput. **55**(5), 549–558 (2006)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
4. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS Press, Amsterdam (2009)
5. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-Breaking Predicates for Search Problems, pp. 148–159. Morgan Kaufmann, San Francisco (1996)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
7. Devriendt, J., Bogaerts, B., Bruynooghe, M.: Symmetric explanation learning: effective dynamic symmetry handling for SAT. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 83–100. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_6
8. Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for SAT. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 104–122. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_8
9. Devriendt, J., Bogaerts, B., de Cat, B., Denecker, M., Mears, C.: Symmetry propagation: improved dynamic symmetry breaking in SAT. In: IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, 7–9 November 2012, pp. 49–56 (2012)

10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
11. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international SAT solver competitions. AI Mag. **33**(1), 89–92 (2012)
12. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Applegate, D., Brodal, G.S., Panario, D., Sedgewick, R. (eds.) Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, pp. 135–149. SIAM (2007)
13. Katebi, H., Sakallah, K.A., Markov, I.L.: Symmetry and satisfiability: an update. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 113–127. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_11
14. Kautz, H.A., Selman, B., et al.: Planning as satisfiability. In: ECAI, vol. 92, pp. 359–363 (1992)
15. Luks, E., Roy, A.: The complexity of symmetry-breaking formulas. Ann. Math. Artif. Intell. **41**(1), 19–45 (2004)
16. Luks, E.M., Roy, A.: The complexity of symmetry-breaking formulas. Ann. Math. Artif. Intell. **41**(1), 19–45 (2004). https://doi.org/10.1023/B:AMAI.0000018578.92398.10
17. Lynce, I., Marques-Silva, J.: SAT in bioinformatics: making the case with haplotype inference. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 136–141. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_16
18. Marques-Silva, J.P., Sakallah, K., et al.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
19. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. J. Autom. Reason. **24**(1), 165–203 (2000)
20. Sabharwal, A.: SymChaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints **14**(4), 478–505 (2009)
21. Schaafsma, B., Heule, M.J.H., van Maaren, H.: Dynamic symmetry breaking by simulating Zykov contraction. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 223–236. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_22