CosySEL: Improving SAT Solving Using Local Symmetries

Sabrine Saouli, Souheib Baarir, Claude Dutheillet, Jo Devriendt

▶ To cite this version:

Sabrine Saouli, Souheib Baarir, Claude Dutheillet, Jo Devriendt. CosySEL: Improving SAT Solving Using Local Symmetries. Verification, Model Checking, and Abstract Interpretation, Jan 2023, Boston (MA), United States. pp.252 - 266, 10.1007/978-3-031-24950-1_12 . hal-04616520

HAL Id: hal-04616520
https://hal.science/hal-04616520v1

Submitted on 18 Jun 2024

CosySEL: Improving SAT Solving Using Local Symmetries

Sabrine Saouli[1(✉)], Souheib Baarir[2], Claude Dutheillet[1], and Jo Devriendt[3]

[1] Sorbonne Université, CNRS, LIP6, 75005 Paris, France
{Sabrine.Saouli, Claude.Dutheillet}@lip6.fr

[2] Université Paris Nanterre (now at EPITA, LRE), 92000 Nanterre, France
Souheib.Baarir@lip6.fr

[3] KU Leuven, Department of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium
jo.devriendt@kuleuven.be

Abstract. Many satisfiability problems exhibit symmetry properties. Thus, the development of symmetry exploitation techniques seems a natural way to try to improve the efficiency of solvers by preventing them from exploring isomorphic parts of the search space. These techniques can be classified into two categories: dynamic and static symmetry breaking. Static approaches have often appeared to be more effective than dynamic ones. But although these approaches can be considered as complementary, very few works have tried to combine them.

In this paper, we present a new tool, CosySEL, that implements a composition of the static Effective Symmetry Breaking Predicates (ESBP) technique with the dynamic Symmetric Explanation Learning (SEL). ESBP exploits symmetries to prune the search tree and SEL uses symmetries to speed up the tree traversal. These two accelerations are complementary and their combination was made possible by the introduction of *Local symmetries*.

We conduct our experiments on instances issued from the last ten SAT competitions and the results show that our tool outperforms the existing tools on highly symmetrical problems.

Keywords: Boolean satisfiability · Symmetry · Dynamic symmetry breaking · Static symmetry breaking · Local symmetries

1 Introduction

The Boolean satisfiability (SAT) problem is the problem of determining whether or not a solution that satisfies a Boolean formula exists, i.e., by assigning *true* or *false* values to the variables of a given Boolean formula, the latter can be evaluated as *true*. If such a solution exists, it is called a *model*.

Boolean satisfiability is a research area with application in fields such as cryptology [22], modal logic [15], decision planning [19], and hardware and software verification. Actually, SAT-based verification techniques have been widely explored [7,26–28,31].

Since SAT problems often exhibit symmetries, developing techniques to handle them prevents solving algorithms from needlessly exploring isomorphic parts of the search space. One common method to exploit symmetries is the *static symmetry breaking method* [1,10]. It consists in precomputing Symmetry Breaking Predicates (SBPs) and adding them to the original problem before starting the search process. These SBPs invalidate symmetrical solutions, so that the solver avoids exploring branches of the search tree symmetrical to the already explored ones. This method has been implemented in tools such as Shatter [2] and Breakid [12]. Even though these approaches are the most efficient on many symmetrical problems, highly symmetrical problems generate a large number of SBPs and this can affect the performance of the used solver.

Dynamic symmetry breaking techniques operate during the search process. Most of them are based on learning symmetric images of already learned clauses. The main such approaches are *Symmetric Learning Scheme (*SLS*)* [6], *Symmetry Propagation (*SP*)* [13] and *Symmetric Explanation Learning (*SEL*)* [11]. Even if these techniques are less effective than the static ones in general, they perform very well on some problems that static approaches fail to solve.

Hence the question of combining both approaches arises naturally, and has already been tackled in some studies: *Effective Symmetry Breaking Predicates method (*ESBP*)* [23], that uses the same principle as static methods, but operates dynamically, has been combined with SP in [24]. In [29], the authors generate SBPs in the preprocessing phase and apply the SEL method afterwards.

The tool we present in this paper combines ESBP with SEL. Our experiments show that it improves the capacity of the conflict-driven clause learning (CDCL) like algorithm to handle some classes of symmetrical SAT problems.

The paper is structured as follows: Sect. 2 gives the basic definitions relevant to this work. Section 3 recalls the notion of local symmetries and presents the combo algorithm. In Sect. 4, we discuss the implementation of the tool and the experimental results.

2 State of the Art and Some Definitions

We recall here some basic definitions and the main ideas of ESBP and SEL.

2.1 Basics on Boolean Satisfiability

Boolean satisfiability aims at checking whether a Boolean formula φ is satisfiable or not, i.e., whether there exists an assignment α of the Boolean variables for which the formula is true. If so, φ is said to be *satisfiable* (SAT), otherwise φ is *unsatisfiable* (UNSAT).

A formula φ in Conjunctive Normal Form (CNF) is a finite conjunction of *clauses*, each clause being a disjunction of (possibly negated) variables. The set of variables of a formula φ is denoted by $\mathcal{V}_\varphi$.

An assignment α is a function $\alpha\colon \mathcal{V}_\varphi \to \{\top, \bot\}$ and can be represented by the subset of its true literals. We call a true literal x if $\alpha(x) = \top$ or $\neg x$ if $\alpha(x) = \bot$. An extension of α is any α' such that $\alpha \subset \alpha'$. Assignment α is said to be *complete* if it contains one literal over each variable in $\mathcal{V}_\varphi$; it is partial otherwise. The set of all (possibly partial) assignments to $\mathcal{V}_\varphi$ is denoted $Ass(\mathcal{V}_\varphi)$.

An assignment α *satisfies* a clause ω, denoted $\alpha \models \omega$, if α contains at least one true literal from ω. An assignment α satisfies a formula φ, denoted $\alpha \models \varphi$, if α satisfies all the clauses in φ. Such an assignment α is said to be a model of φ. The formula φ is *unsatisfiable* (UNSAT) otherwise. For more details, the interested reader can refer to the very complete handbook [8].

Example. Let $\varphi = \{\{x_1, x_2, x_3\}, \{x_1, x_2\}, \{\neg x_1, x_3\}\}$ be a formula. The partial assignments $\{\neg x_1, x_2\}$ and $\{x_1, x_3\}$ satisfy φ, so φ is satisfiable. Extending φ with the unit clauses $\{\neg x_2\}$ and $\{\neg x_3\}$ would make it unsatisfiable.

2.2 Symmetry Group of a Formula

Let φ be a formula and let $\mathfrak{S}(\mathcal{V}_\varphi)$ be the group of permutations of $\mathcal{V}_\varphi$ under composition. We say that $g \in \mathfrak{S}(\mathcal{V}_\varphi)$ is a symmetry of φ if and only if for every complete assignment α such that $\alpha \models \varphi$, $g.\alpha \models \varphi$, with $g.\alpha = \{g(x) \mid x \in \alpha\} \cup \{\neg g(x) \mid \neg x \in \alpha\}$. We denote $S(\varphi) \subseteq \mathfrak{S}(\mathcal{V}_\varphi)$ the symmetry group of φ and we call *generator* the elements of a generating set of $S(\varphi)$. A variable x is said to appear in a generator g if $g(x) \neq x$.

Let G be a subgroup of $\mathfrak{S}(\mathcal{V}_\varphi)$. The *orbit of* α *under* G is the set $[\alpha]_G = \{g.\alpha \mid g \in G\}$. The set of orbits $\{[\alpha]_G \mid \alpha \in Ass(\mathcal{V}_\varphi)\}$ partitions $Ass(\mathcal{V}_\varphi)$ into equivalence classes, called *symmetry classes* of φ when $G = S(\varphi)$. We introduce an ordering relation between assignments in order to identify a unique representative for each symmetry class.

Definition 1. [23] *We assume a total order $\prec$ on $\mathcal{V}_\varphi$. Given two assignments $\alpha, \beta \in Ass(\mathcal{V}_\varphi)$, $\alpha \prec \beta$, if there exists a variable $v \in \mathcal{V}_\varphi$ 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 $v \in \beta$.*

Moreover, $\prec$ is a total order on complete assignments. For a complete assignment α we define the lexicographic leader (*lex-leader*) of an orbit $[\alpha]_G$ as the minimum of $[\alpha]_G$ w.r.t. $\prec$.

2.3 (Effective) Symmetry Breaking

From the above presentation, it is clear that either all the assignments within the same symmetry class satisfy the formula, or none do. Adding a *Symmetry Breaking Predicate* (SBP) to a symmetric SAT formula aims at limiting the search tree exploration to only one assignment per symmetry class, e.g., the *lex-leader*. However, finding the *lex-leader* of a class is computationally hard [20] and best-effort approaches are commonly used [2,12].

SBPs were first introduced as pre-generated predicates (i.e., in a *static* approach) but they required auxiliary variables, making the size of the formulas often intractable in practice. *Effective* SBPs (ESBPs) were then proposed to tackle the problem with a *dynamic* approach [23], where the solver detects on-the-fly when the current assignment cannot be extended to a *lex-leader*. Actually, assignment ordering is monotonic, i.e., whenever $\alpha < \beta$, any extension α' of α (resp. β' of β) are such that $\alpha' < \beta'$. Hence, if $g.\alpha < \alpha$, any possible extension α' of α is such that $g.\alpha' < \alpha'$, because $g.\alpha'$ is an extension of $g.\alpha$. In this case, we can define a predicate contradicting α that still preserves the satisfiability of the formula. Such a predicate will be used to discard α and all its extensions from further exploration, thus pruning the search tree.

Definition 2. [23] *Let $\alpha \in Ass(\mathcal{V}_\varphi)$, and $g \in \mathfrak{S}(\mathcal{V}_\varphi)$. We say that the formula ψ is an* Effective Symmetry Breaking Predicate *(*ESBP*) for α under g if:*

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

.

The equi-satisfiability of φ *and* $\varphi \cup \psi$ is guaranteed by the fact that ψ will not prune the branch of the lex-leader. This approach avoids the pre-generation of a large SBP that could have a negative effect on the overall performance of the classical static symmetry breaking approaches. The extensive experiments conducted in [23] show that it outperforms other state-of-the-art symmetry breaking techniques, both dynamic and static, when considering the total number of solved instances. However, this technique fails to solve some problems that have been trivially solved by other dynamic symmetry breaking techniques such as SEL developed in [13]. We give an overview of SEL in the following section.

2.4 Symmetric (Explanation) Learning

An orthogonal approach to symmetry breaking is *symmetric learning*. The idea here is not to remove the symmetric assignments by posting extra constraints, but to add implied symmetric clauses to a SAT solver's internal *learned clause database*. Symmetric learning hinges on the following theorem:

Theorem 1. [6] *Let φ be a formula, $g \in S(\varphi)$ a symmetry for the formula, and ω a clause. Then, $\varphi \models \omega$ implies $\varphi \models g.\omega$.*

As a result, for any implied clause derived by a SAT solver, any of its symmetric images can safely be derived as well, which, through unit propagation, discourages the solver from visiting symmetric search branches [13].

The crucial question when implementing symmetric learning is how to avoid overloading the solver with exponentially many symmetric clauses, while retaining effective pruning of the search tree. One answer is *symmetric explanation* *learning* (SEL), which was shown to be competitive to (but not better than) state-of-the-art static symmetry breaking [11]. The idea behind SEL, given a small set[1] of symmetries G, is to keep track of all clauses $\{g.\omega \mid g \in G\}$ symmetric to the clauses ω that triggered a currently propagated literal. Only the symmetric clauses that propagate in turn will be added to the learned clause database. When they propagate, all their symmetric images will be tracked, in effect composing the symmetries in G. If the propagation explained by a clause is cancelled (when backtracking the search), SEL will quickly forget the symmetric images that did not propagate.

3 The Proposed Technique

Our first attempt to combine static and dynamic approaches was proposed in [24], where we combined ESBP with SP. However, as it appears that SEL is theoretically more effective than SP, we decided to investigate the integration of ESBP with SEL. This work can also be considered as a generalization of [29], where the combination was purely static and did not take advantage of the upcoming notion of local symmetries [24].

The idea of the SEL approach is to derive and efficiently use symmetrical clauses using a subset of $S(\varphi)$, the symmetry group of φ (the correctness is thus guaranteed by Theorem 1). If φ is extended by a set of clauses ψ, preserving equisatisfiability, then SEL can be applied, as long as $S(\varphi \cup \psi)$ is known. Therefore, the effectiveness of the composition of SEL and ESBP strongly depends on how hard it is to compute the elements of $S(\varphi \cup \psi)$.

The notion of *local symmetries* was introduced as the theoretical framework materializing the computation of the aforementioned elements in [24]. In this section, we recall the definition and properties of *local symmetries* and invite the interested reader to consult the original work [24] for more details.

3.1 Theoretical Foundations and Practical Considerations

Local symmetries of a clause of a formula are defined as follows.

Definition 3. *Let φ be a formula. We define $L_{\omega,\varphi}$, the set of local symmetries for a clause ω with respect to a formula φ, as follows:*

$$L_{\omega,\varphi} = \{g \in \mathfrak{S}(\mathcal{V}) \mid \varphi \models g.\omega\}$$

Through this definition, it is straightforward to derive the next proposition.

Proposition 1. *Let φ be a formula. Then, $\bigcap_{\omega \in \varphi} L_{\omega,\varphi} \subseteq S(\varphi)$.*

[1] Such a small set typically does not form a group, i.e., is not closed under composition, but closing it under composition *generates* a detected symmetry group for the formula.

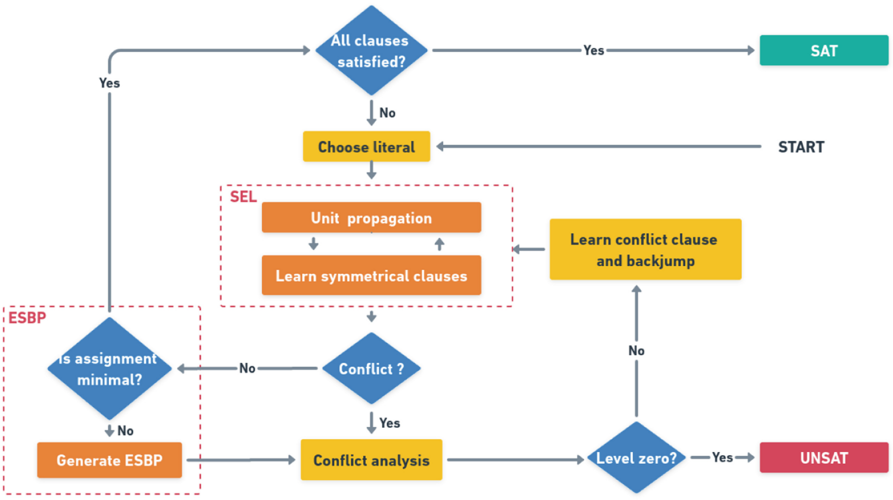

Fig. 1. Workflow of the ESBP_SEL algorithm.

A direct consequence is that the intersection of the sets of *local symmetries* of all the clauses of a formula φ are symmetries of $S(\varphi)$. Hence, when adding a symmetry breaking predicate ω to φ, a set of valid symmetries for $\varphi \cup \{\omega\}$ can be computed on-the-fly as the intersection of $L_{\omega,\varphi}$ and $\bigcap_{\omega' \in \varphi} L_{\omega',\varphi}$.

However, full $L_{\omega,\varphi}$ sets are hard to compute in general, hence our tool only computes subsets based on the following considerations. While solving a formula based on a symmetry breaking approach, three sets of clauses are manipulated: the original formula φ, the set of SBP clauses φ_e and the set φ_d of clauses derived from $\varphi \cup \varphi_e$. Our computation of the local symmetries of a clause ω takes into account the fact that symmetries $S(\varphi)$ of φ are already known, and depends on which of the three sets φ, φ_e, or φ_d, ω belongs to.

Let $\varphi' = \varphi \cup \varphi_e \cup \varphi_d$. There are three cases:

1. if $\omega \in \varphi$, then by definition $S(\varphi) \subseteq L_{\omega,\varphi'}$, so we take $S(\varphi)$ as a representative for $L_{\omega,\varphi'}$.
2. if $\omega \in \varphi_e$, this is an ESBP clause, and we choose the set of stabilizing symmetries: $Stab(\omega) = \{g \in \mathfrak{S}(\mathcal{V}) \mid \omega = g.\omega\} \subseteq L_{\omega,\varphi'}$.
3. if $\omega \in \varphi_d$, this is a derived clause, and we choose the set $(\bigcap_{\omega' \in \varphi_1} L_{\omega',\varphi'})$, where φ_1 is the set of clauses that derived ω.

3.2 Algorithm

In this section, we describe how we combined SEL and ESBP.

Figure 1 gives an overview of the integration of ESBP and SEL in the CDCL algorithm.

```
1  function ESBP_SEL($\varphi$: CNF formula, symCtrl: symmetry controller)
2
3      $dl \leftarrow 0$ ;                                   // Current decision level
4      while not all variables are assigned do
5          $isConflict \leftarrow$ unitPropagation() $\wedge$ selPropagation();
6          symCtrl.updateAssign(crtAssignment());
7          $isReduced \leftarrow$ symCtrl.isNotLexLeader(crtAssignment());
8          if $isConflict \vee isReduced$ then
9              if $dl = 0$ then
10                 return UNSAT;                                       // $\varphi$ is UNSAT
11             if $isConflict$ then
12                 $\langle \omega, L = \bigcap_{\omega' \in \varphi_1} L_{\omega', \varphi_1} \bigcup Stab(\omega) \rangle \leftarrow$ analyzeConflictEsbpSel();
13             else
14                 $\langle \omega, L = Stab(\omega) \rangle \leftarrow$ symCtrl.genEsbpSel(crtAssignment());
15             $dl \leftarrow$ backjumpOrRestart();
16             $\varphi \leftarrow \varphi \cup \{\omega\}$ ;
17             symCtrl.updateCancel(crtAssignment());
18         else
19             assignDecisionLiteral();
20             $dl \leftarrow dl + 1$;
21     return SAT;                                                     // $\varphi$ is SAT
```

Algorithm 1: The ESBP_SEL algorithm. Instructions derived from ESBP and SEL algorithms are reported in blue and red (respectively). Instructions derived from the combination are reported with a grey background.

The integration of SEL in the CDCL algorithm operates the same way as a basic CDCL, except for the unit propagation function (SEL in Fig. 1). With SEL, the algorithm keeps symmetrical versions of propagated literals' explanation clauses in a different database and in addition to regular unit propagation over the regular clauses, when a symmetrical clause is asserting, SEL adds it to the learnt clauses and the asserting symmetrical literal is propagated.

The integration of ESBP then consists in controlling the behaviour of the previous algorithm by introducing a *symmetry controller component* that operates all symmetry-based actions. It inspects all partial assignments and detects non-minimal ones as soon as possible. In this case, it generates an ESBP clause and injects it into the original problem (see ESBP in Fig. 1).

The details of the aforementioned approach are given in Algorithm 1. The algorithm first executes the *unitPropagation()* and *selPropagation* functions (line 5). In propagation phase, regular and symmetrical unit clauses are propagated until a conflict is detected or fixed point is reached. Next, the symmetry controller updates the current assignment and checks if it can still be extended to a lex-leader (lines 6 – 7). When a conflict is detected, function *analyzeCon-*

flictEsbpSel() (line 12) analyses the conflict and generates a *learnt clause* ω. With respect to a classical analysis function of a basic CDCL algorithm, *analyzeConflictEsbpSel()* will generate the set of local symmetries associated with ω. This is done by computing the intersection of the sets of symmetries of all the clauses used to derive ω (as explained in Sect. 3.1), augmented with the stabilizers set. If the current assignment is conflict free but can not be extended to a lex-leader, function *genEsbpSel()* (line14) is called. It generates the ESBP clause to inject, along with its set of stabilizers. Function *updateCancel()* (line17) is the counterpart of function *backjumpOrRestart* but for *symCtrl*, the symmetry controller.

4 Tooling and Evaluation

In this section, we first present the tooling support of our combined approach ESBP_SEL combining ESBP and SEL. Then, we compare it to the vanilla SAT solver Glucose[2] [5] and to the implementations of ESBP, SEL on top of Glucose and discuss the results.

4.1 Tool Usage

Cosy[3] is a C++ library offering all the functionalities necessary for the implementation of the ESBP method. This library can easily be integrated to any CDCL-like solver.

The implementation of ESBP approach on top of Glucose is available at https://github.com/lip6/cosy/tree/master/solvers/glucose-3.0. In the remaining of this paper, this implementation is simply called Cosy. The implementation of SEL approach on top of Glucose is available at https://bitbucket.org/krr/glucose-sel. Our implementation, CosySEL, of the combined approach ESBP_SEL is available at https://github.com/sabrinesaouli/CosySEL. It integrates Cosy in the already mentioned implementation of SEL according to Algorithm 1.

CosySEL can be used with two symmetry generator tools: Bliss [17] or Saucy [18]. These are two of the best graph automorphism tools, that compute a set of symmetries for a given graph.

In our tool-chain, a given CNF formula is first encoded as a colored graph into a file that is given to Bliss or Saucy to obtain the set of symmetry generators as a file in the corresponding format (`.bliss` or `.sym` respectively). The obtained file is then given along with the `.cnf` file to the CosySEL solver (Fig. 2).

This workflow is encapsulated in a script `cosysel.sh` that we can execute with **bliss** or **saucy** options.

```
$./cosysel.sh bliss <\cnf>
$./cosysel.sh saucy <\cnf>
```

[2] https://www.labri.fr/perso/lsimon/downloads/softwares/glucose-syrup.tgz.
[3] Cosy library is released under GPL v3 license at https://github.com/lip6/cosy.

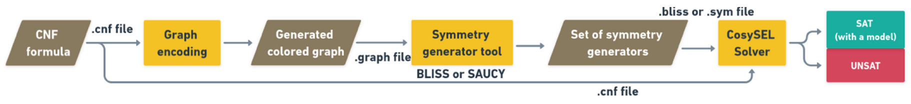

Fig. 2. Workflow of the CosySEL tool.

4.2 Evaluation

Among all the instances from the last ten SAT competitions (from 2012 to 2021) [16], we selected the 1362 for which Bliss detects at least one symmetry generator in at most 1000 s of CPU time.

All experiments were conducted using the following settings: each solver ran once on each problem, with a memory limit of 15 GB and a time-out of 7200 s seconds (this time limit includes symmetry detection time for all the solvers except for Glucose which does not compute symmetries). Experiments were executed on a computer with an Intel(R) Xeon(R) Gold 6148 CPU @ 2.40 GHz and 1500 GB of memory, running Linux 5.0.16.

All the approaches dealing with symmetries are built on top of a CDCL-like solver. Therefore, for a fair comparison, the solver must not introduce side effects. As our tool combines SEL and ESBP, we must at least prove that it outperforms both. SEL is built on top of Glucose, and there is also a version of ESBP on Glucose, so we chose Glucose [5] to avoid solver-induced side effects.

To be complete in our study, it seems natural to compare our new approach with the combination of ESBP and SP presented in [24] and implemented on top of MiniSAT (in a tool referred here as CosySP). However, although it was shown in [11] that SEL approach is theoretically more efficient than SP, no available implementation supports this claim. Nevertheless, as SP is built on top of MiniSAT, we decided to implement SEL on top of MiniSAT and compare the approaches on the whole benchmark. The results are given in Table 1 and confirm that SEL is by far better than SP. Therefore, considering that implementing a combination of ESBP and SP on top of Glucose is not simple and would require a great effort, we relied on these results to consider it is not relevant.

We computed the symmetries of each instance with Bliss and Saucy. Bliss is known to compute a larger number of generators for the symmetry group compared to Saucy. As shown in [23], in an SBP-like approach, this influences the results since it allows to cut branches of the search tree early.

Table 2 compares the use of Saucy and Bliss for computing symmetry generators. The values represent the number of SAT, UNSAT, and the total number

Table 1. Comparison of the number of instances solved by MiniSAT-SEL and MiniSAT-SP using Bliss.

	MiniSAT-SEL	MiniSAT-SP
SAT	304	271
UNSAT	402	378
TOTAL(1362)	706	649

Table 2. Comparison of different approaches when using Saucy and Bliss.

	None	Saucy			Bliss		
	Glucose	Cosy	sel	CosySEL	Cosy	sel	CosySEL
sat	238	227	255	229	235	253	240
unsat	473	452	505	474	497	503	553
TOTAL(1362)	711	679	760	703	732	756	793

Table 3. Comparison of instances solved by each approach according to the percentage of the variables in the symmetries, using Bliss to detect symmetry (the table is restricted to instances solved by at least one solver).

(a) sat instances

% sym vars	Glucose	Cosy	sel	CosySEL
0% - 25% (195)	152	149	174	165
25% - 50% (28)	19	23	19	16
50% - 75% (14)	14	14	13	12
75% - 100% (62)	53	49	47	47
Total (299)	238	235	253	240

(b) unsat instances

% sym vars	Glucose	Cosy	sel	CosySEL
0% - 25% (250)	242	211	233	218
25% - 50% (21)	20	19	21	20
50% - 75% (11)	8	7	9	9
75% - 100% (330)	203	260	240	306
Total (612)	473	497	503	553

of instances solved. It shows that our tool performs poorly with Saucy. Actually it computes too few symmetries to allow CosySEL to detect early non-lex-leader assignments, hence the overhead of keeping track of local symmetries is not counterbalanced. The effectiveness of Cosy, and thus the combined tool, is largely relying on the number of observed (tracked) literals while solving the problem and because of the reduced number of generators given by Saucy, this can be an issue for these approaches.

The results in Table 2 confirm that CosySEL is more effective using Bliss than Saucy. Globally, we notice that CosySEL, when used with Bliss, is the most effective, especially when considering unsat instances. It solves 50 more unsat and 13 fewer sat instances than the second-best method (sel).

After establishing that our tool works better with Bliss, we compared its effectiveness against the three others in each class of problems. In Table 3, results are split according to the percentage of variables appearing in at least one generator computed by Bliss, with the first column giving the intervals of percentages. Table 3a (respectively 3b) shows the number of sat (respectively unsat) instances solved by each approach in each interval.

unsat problems are exceptionally difficult to solve as they require traversing the entire search space, but our tool is particularly effective for highly symmetrical unsat problems. As far as sat problems are concerned, the loss of performance can be explained by the fact that Cosy can stop the exploration of a satisfying branch of the search space because it is not a lex-leader, even though this branch could still contain a non-lex-leader solution.

It is essential to mention that (as shown in Table 4a), CosySEL increased the VBS(Virtual Best Solver)—which represents the best performances combined, i.e. the number of instances that at least one solver can solve—with **35** problems

Table 4. Virtual Best Solver (VBS) results with and without CosySEL.

(a) Comparing the VBS when using Glucose, Cosy and sel only and when adding CosySEL to the set of solvers.

	without CosySEL	with CosySEL
SAT	297	299 (+**2**)
UNSAT	579	612 (+**33**)
TOTAL(1362)	876	911 (+**35**)

(b) Comparing the VBS when using Kissat-MAB only and when adding CosySEL.

	Kissat-MAB	Kissat-MAB+CosySEL
SAT	436	443 (+**7**)
UNSAT	570	680 (+**110**)
TOTAL(1362)	1006	1123 (+**117**)

that the other methods failed to solve. We also compared it to the best solver of the SAT contest 2021 Kissat-MAB [9] (see Table 4b). We noted that even though our CosySEL tool is overall less effective, it managed to solve **117** problems (mainly UNSAT) that Kissat-MAB could not handle. Upon taking a closer look into the classes of problems exclusively solved by CosySEL, we further confirm that our tool is more effective at handling highly symmetrical UNSAT problems:

– UNSAT and fully symmetrical: 40/117 are several variations of the *pigeon hole problem*, 30/117 are *Tseitin formulas* [30], and 2/117 belong to the class of *n-queens problem.*
– UNSAT with more than 90% of the variables being in the symmetry generators: 17/117 from the *clique colouring* class.
– The remaining 25/117 instances are of diverse classes of problems including 6 from the SAT-*based Bitcoin mining problems (Satcoin*[4] [21] which are SAT with $\sim 0.15\%$ of variables being in the symmetry generators.

Out of those 117 instances and according to the publicly available SAT competition results, our proposal is the only tool having succeeded at solving some problems that were previously unsolved: 29 *Tseitin formulas* (28 from SAT 2016 and 1 from SAT 2019), 4 *relativised pigeonhole problems* [3,4,14] from SAT competition 2016 and 2 classic *pigeonhole problems* from SAT competition 2021.

To push our experiments even further, we compared our tool to the previously mentioned solvers (including CosySP)[5] on fully symmetrical problems (all the variables are involved in symmetries). We collected 282 problems (Table 5) from the last SAT competitions, from [11] and from [25]. The instances represent different classes of problems like the *pigeon hole, the clique colouring and the channel routing problems*. We plotted the results in Figs. 3a and 3b for SAT and UNSAT instances respectively.

Figure 3a shows that Kissat-MAB is more effective on SAT problems compared to the other tools, which solve more or less the same number of instances. However, Fig. 3b shows that CosySEL stands out when it comes to UNSAT instances and even Kissat-MAB cannot compete with the tools exploiting symmetries except for CosySP that solves two fewer instances.

[4] github.com/jheusser/satcoin.
[5] We recall that CosySP is based of MiniSAT, and the comparison with the other tools is not totally fair!.

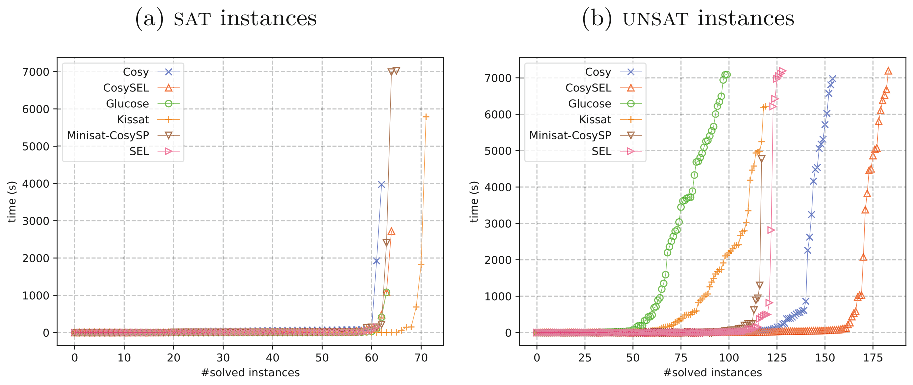

Fig. 3. Comparison of different approaches on fully symmetrical instances.

Table 5. Comparison of approaches on fully symmetrical instances using Bliss.

	Glucose	Cosy	SEL	CosySEL	CosySP	Kissat-MAB
SAT	64	63	62	65	66	72
UNSAT	100	155	129	184	118	120
`TOTAL(282)`	164	218	191	249	184	192

In Table 5, we observe that CosySP is slightly more effective than the other solvers exploiting symmetries on SAT instances. Nevertheless, on UNSAT problems, it solves fewer instances than almost all the other solvers. It is hard to identify whether this relatively low performance is due to the fact that SP is overloading the solver by keeping track of the status of all symmetries, or because it is embedded in an older CDCL solver (MiniSAT). Either ways, it is clear that our implementation of CosySEL outperforms the publicly available version of CosySP by solving 65 more instances. This is consistent with previously made observation regarding SEL and SP (see Table 1).

5 Conclusion

We presented in this paper CosySEL: a tool that combines ESBP, which dynamically adds symmetry breaking predicates to the formula, with SEL, which is based on learning symmetrical clauses only when they are useful.

This combination relies on the definition of *local symmetries* for a clause, which makes it possible to efficiently compute a subset of the symmetries of the formula each time symmetry breaking predicates are added to it. Our experiments investigate the effectiveness of the combined approach.

They show that CosySEL can solve a significant number of highly symmetrical problems that state-of-the-art solvers fail to handle. We believe that CosySEL works better with Bliss than Saucy because Bliss detects a more significant number of symmetries, which helps ESBP cut symmetrical branches earlier. It is also more efficient when the problem is UNSAT and highly symmetrical because ESBP allows the solver to visit only few (partial) assignments per symmetrical class. Moreover, the more symmetries, the more branches ESBP cuts, and the more symmetrical clauses SEL learns. However, CosySEL seems to be less effective on SAT problems. This may be due to ESBP stopping the exploration of a sat branch of the search tree just because it is not a lex-leader.

As a future work, we plan to implement ESBP_SEL in a more recent and effective SAT solver than Glucose, such as the winner of the 2021 SAT competition Kissat-MAB [9], or the best scoring Maple-like solver [32], which derives most of its code-base from Glucose and hence may be easier to port CosySEL to.

References

1. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult sat instances in the presence of symmetry. In: Proceedings 2002 Design Automation Conference (IEEE Cat. No. 02CH37324) pp. 731–736. IEEE (2002)
2. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for Boolean satisfiability. IEEE Trans. Comput. **55**(5), 549–558 (2006)
3. Atserias, A., Lauria, M., Nordström, J.: Narrow proofs may be maximally long. ACM Trans. Comput. Logic (TOCL) **17**(3), 1–30 (2016)
4. Atserias, A., Müller, M., Oliva, S.: Lower bounds for DNF-refutations of a relativized weak pigeonhole principle. J. Symbolic Logic **80**(2), 450–476 (2015)
5. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, pp. 399–404 (2009)
6. Benhamou, B., Nabhani, T., Ostrowski, R., Saïdi, M.R.: Enhancing clause learning by symmetry in sat solvers. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, vol. 1, pp. 329–335. IEEE (2010)
7. 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
8. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS press, Amsterdam (2009)
9. Cherif, M.S., Habet, D., Terrioux, C.: Kissat MAB: combining VSIDS and CHB through multi-armed bandit. SAT Competition **2021**, 15 (2021)
10. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. KR **96**(1996), 148–159 (1996)
11. 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
12. 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
13. Devriendt, J., Bogaerts, B., De Cat, B., Denecker, M., Mears, C.: Symmetry propagation: Improved dynamic symmetry breaking in SAT. In: 2012 IEEE 24th International Conference on Tools with Artificial Intelligence, vol. 1, pp. 49–56. IEEE (2012)

14. Elffers, J., Nordström, J.: Documentation of some combinatorial benchmarks. Proc. SAT Competition **2016**, 67–69 (2016)
15. Giunchiglia, F., Sebastiani, R.: Building decision procedures for modal logics from propositional decision procedures—the case study of modal K. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 583–597. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61511-3_115
16. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international sat solver competitions. AI Mag. **33**(1), 89–92 (2012)
17. 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)
18. 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
19. Kautz, H.A., Selman, B., et al.: Planning as satisfiability. In: ECAI, vol. 92, pp. 359–363 (1992)
20. Luks, E.M., Roy, A.: The complexity of symmetry-breaking formulas. Ann. Math. Artif. Intell. **41**(1), 19–45 (2004)
21. Manthey, N., Heusser, J.: Satcoin-bitcoin mining via SAT. SAT Competition **2018**, 67 (2018)
22. Massacci, F., Marraro, L.: Logical cryptanalysis as a sat problem. J. Autom. Reasoning **24**(1), 165–203 (2000)
23. Metin, H., Baarir, S., Colange, M., Kordon, F.: CDCLSym: introducing effective symmetry breaking in SAT solving. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 99–114. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_6
24. Metin, H., Baarir, S., Kordon, F.: Composing symmetry propagation and effective symmetry breaking for SAT solving. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 316–332. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_21
25. Sabharwal, A.: Symchaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints **14**(4), 478–505 (2009)
26. Shtrichman, O.: Tuning SAT checkers for bounded model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 480–494. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_36
27. Shtrichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 58–70. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_4
28. Tang, D., Malik, S., Gupta, A., Ip, C.N.: Symmetry reduction in SAT-based model checking. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 125–138. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_12
29. Tchinda, R.K., Tayou Djamegni, C.: Enhancing static symmetry breaking with dynamic symmetry handling in CDCL SAT solvers. Int. J. Artif. Intell. Tools **28**(03), 1950011 (2019)
30. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of reasoning, pp. 466–483. Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28

31. Wang, C., Jin, H., Hachtel, G.D., Somenzi, F.: Refining the SAT decision ordering for bounded model checking. In: Proceedings of the 41st Annual Design Automation Conference, pp. 535–538 (2004)
32. Zhang, X., Cai, S., Chen, Z.: Improving CDCL via local search. SAT Competition **2021**, 42 (2021)