Symmetry Propagation: Improved Dynamic Symmetry Breaking in SAT

Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker
Department of Computer Science
KU Leuven
Leuven, Belgium
e-mail: {jo.devriendt, bart.bogaerts, broes.decat, marc.denecker}@cs.kuleuven.be

Christopher Mears
Caulfield School of IT
Monash University
Melbourne, Australia
e-mail: chris.mears@monash.edu

***Abstract*—For constraint programming, many well performing dynamic symmetry breaking techniques have been devised. For propositional satisfiability solving, dynamic symmetry breaking is still either slower or less general than static symmetry breaking. This paper presents Symmetry Propagation, which is an improvement to Lightweight Dynamic Symmetry Breaking, a dynamic symmetry breaking approach from CP. Symmetry Propagation uses any given symmetry as a propagator, and as a result is a general symmetry breaking technique. Experiments with an implementation in the SAT solver Minisat show that on many benchmarks, Symmetry Propagation outperforms the state-of-the-art static symmetry breaking method Shatter.**

***Keywords*-Dynamic Symmetry Breaking, SAT, CP**

I. Introduction

Many difficult propositional satisfiability (SAT) and constraint programming problems exhibit symmetry properties. Exploiting these symmetry properties can significantly reduce the time needed to solve such problems. The process of exploiting symmetries is called *symmetry breaking*, which aims to speed up search by only considering parts of the search space which are not symmetrical to already considered parts. Symmetry breaking can be subdivided into two categories: *static* and *dynamic* symmetry breaking.

Static symmetry breaking adds to an original problem extra constraints which exclude a large part of the search space. The goal is to let the solver only explore the remaining small part containing solutions not symmetrical to each other. Static symmetry breaking can be used with any symmetry, performs well, and since the extra constraints are typically added in a preprocessing step, it does not require a modification of the particular solving mechanism.

A disadvantage of static symmetry breaking is that it pushes the solver to a particular part of the search space which might not be the most efficient part for the problem and solver at hand [1]. Another disadvantage is that the extra constraints can be too large for a solver to handle. Finally, static symmetry breaking is not always possible, for instance when symmetries are detected during search [2], or when a problem is symmetrical, but an optimization function over the solutions is not [3].

Dynamic symmetry breaking does not have these problems because it breaks symmetry by interacting with the solver during search, ensuring search paths symmetrical to an already inspected path are avoided. Both in constraint programming (CP) and SAT, many dynamic symmetry breaking methods have been proposed. Unlike in CP, dynamic symmetry breaking methods for SAT are either less performant than static symmetry breaking, or are limited to a specific class of symmetries.

In this paper, we present *Symmetry Propagation* (*SP*), a dynamic symmetry breaking approach that speeds up search by propagating literals symmetrical to already propagated literals.

This paper starts out by a brief overview of how modern SAT solvers work in Sect. II. Section III delves into theoretical properties of symmetry, providing tools such as the notion of *weak activity* to detect logical consequences of a logical theory. This section also contains a detailed description of how SP can be implemented in a SAT solver. An actual implementation in Minisat [4] yielded the experimental results given in Sect. IV. Section V relates SP to other symmetry breaking algorithms available in literature. We conclude this paper and discuss further research opportunities in Sect. VI.

II. Background

A. The SAT Problem

In this section, we briefly recall some terminology regarding SAT solving.

We assume a *theory* T is a conjunction of *clauses*, a clause is a disjunction of *literals*, and a literal is an expression of the form x or $\neg x$, with x a boolean variable. The set of variables occurring in T is denoted by $\Sigma(T)$ and the set of all possible literals of T is $\bar{\Sigma}(T)$. An *assignment* for a SAT problem is a set of literals $\alpha \subseteq \bar{\Sigma}(T)$ such that α contains at most one literal of every variable in $\Sigma(T)$. An assignment is *complete* if it contains as many literals as there are variables in $\Sigma(T)$; if not complete, it is *partial*. A literal l is *true* under assignment α if $l \in \alpha$, l is *false* if $\neg l \in \alpha$, and l is *undefined* otherwise. A clause c is a *conflict clause* under assignment α if it contains only false literals, c is *satisfied* under α if it contains at least one true literal, and c is a *unit clause* under α if all but one literal in c are false.

A complete assignment α is a *model* for T if all clauses of T are satisfied under α. A clause c is a logical consequence of T if c is satisfied in all models of T; we denote this by $T \models c$. A theory T' is a logical consequence of T if all of its clauses are, and a literal l is a logical consequence of T if the clause containing only l is. A theory T is *satisfiable* if there exists a model for T. The SAT problem consists of deciding whether or not a given theory is satisfiable.

Finally, we denote by $T \cup \alpha$ a theory that consists of the clauses of T and for each literal $l \in \alpha$ a clause containing only l.

B. A Conflict Driven Clause Learning Solver

We briefly recall some of the concepts of a Conflict Driven Clause Learning (CDCL) SAT solver [5].

At each search step, a SAT solver chooses a value for an unassigned variable of the given theory, and adds the corresponding literal to the current assignment. This literal is called a *choice literal*, and may result in some clauses becoming unit clauses under the new assignment. This prompts a *unit propagation* phase, where all undefined literals occurring in a unit clause are added to the current assignment. These literals are *propagated literals*, with the unit clause they belong to referred to as their *explanation clause*. If no more unit clauses remain under the resulting assignment, the unit propagation phase ends, and a new search step starts by choosing a new choice literal.

If at some point a literal l would be propagated with $\neg l$ present in the current assignment, a *conflict* arises. At this moment, the CDCL solver will construct a *learned clause* by investigating the explanation clauses for the unit propagations leading to the conflict. This learned clause is a logical consequence of the theory, and adding it to the theory prevents the conflict from occurring again after a backjump. We refer to the collection of learned clauses of a CDCL SAT solver as the *learned clause store*.

Formally, we characterize the state of a CDCL solver solving a theory T by a triple $(\alpha, \delta, expl)$, where α is the current assignment, $\delta \subseteq \alpha$ the set of choice literals such that $T \cup \delta \models T \cup \alpha$, $\alpha \setminus \delta$ is the set of propagated literals, and $expl$ is a function from $\alpha \setminus \delta$ to T such that, for each propagated literal $l \in \alpha \setminus \delta$, $expl(l)$ is the explanation clause for l.

III. Symmetry Propagation

A. Theoretical Properties of Symmetries

Given a theory T, let σ be a permutation of $\bar{\Sigma}(T)$. As such, σ is a mapping of literals. We can extend σ in a natural way to be a mapping of clauses, assignments and theories, and by slight abuse of notation, we will identify σ with those extensions. σ is said to *commute with negation* if $\sigma(\neg l) = \neg\sigma(l)$ for every literal $l \in \bar{\Sigma}(T)$. σ is called a *symmetry* of T if it commutes with negation and if for every complete assignment α: α is a model of T if and only if $\sigma(\alpha)$ is. Equivalently, σ is a symmetry if it commutes with negation and if T and $\sigma(T)$ are logically equivalent. We will always denote our symmetries in disjoint cycle notation, i.e. $(ab)(c\neg de)$ denotes the symmetry that sends the literals a to b, b to a, c to $\neg d$, $\neg d$ to e and e to c. We assume that this symmetry also commutes with negation (i.e., it sends $\neg a$ to $\neg b$, d to $\neg e$,...) and sends all remaining literals to themselves. This definition of symmetry is based on the definition of semantic symmetry [6], but it also maintains boolean consistency by enforcing the commute with negation property.

Symmetries can be composed, so a set of symmetries $\mathcal{S}$ of a theory T generates a subgroup $G_{\mathcal{S}}$ of the group of all permutations of $\bar{\Sigma}(T)$. We call $G_{\mathcal{S}}$ a *symmetry group* of T. In general, the size of a symmetry group may be exponentially larger than a set of generators of this group. In practice, this means that if theories have symmetries, they often have very many of them. For a symmetry group $G_{\mathcal{S}}$, we define the *orbit* of a literal l as $\{\sigma(l) | \sigma \in G_{\mathcal{S}}\}$. Similarly, the orbit of a clause c is $\{\sigma(c) | \sigma \in G_{\mathcal{S}}\}$. The *order* of a symmetry σ is the smallest positive n such that σ^n is the identity.

A well-known property of symmetries of a SAT problem is the following:

Proposition 1. *Given a SAT problem with theory T, a symmetry σ of T, and a clause c. If $T \models c$, then also $T \models \sigma(c)$.*

Since learned clauses are always logical consequences of the initial theory, every time a SAT solver learns a clause c, we may apply Proposition 1 and add $\sigma(c)$ as a learned clause for every symmetry σ of some symmetry group of T. In fact, this approach can be used as a symmetry breaking tool for SAT: because every learned clause prevents the solver from encountering a certain conflict, the orbit of this clause under some symmetry group will prevent the encounter of all symmetrical conflicts, resulting in complete symmetry breaking. However, since there are possibly exponentially many symmetries, this approach will in most cases add too many clauses to the theory to be of practical use. Symmetry breaking methods based on Proposition 1 need to limit the amount of symmetrical learned clauses. For example, the Symmetrical Learning Scheme [6] only adds $\sigma(c)$ to the learned clause store for each element σ of the generator set $\mathcal{S}$, rather than for each element of the group $G_{\mathcal{S}}$.

Instead of adding symmetrical clauses, SP breaks symmetry by propagating symmetrical literals. The following corollary of Proposition 1 is the foundation for SP:

Corollary 2. *Let T be the theory of a SAT problem, α an assignment and l a literal. If σ is a symmetry of $T \cup \alpha$ and $T \cup \alpha \models l$, then also $T \cup \alpha \models \sigma(l)$.*

Corollary 2 means that if a SAT solver has state $(\alpha, \delta, expl)$ where l can be propagated, then for every symmetry σ of $T \cup \alpha$, $\sigma(l)$ can also be propagated.

To detect whether symmetries of T are symmetries of

$T \cup \alpha$, Mears et al. introduced the notion of *activity* in their Lightweight Dynamic Symmetry Breaking algorithm [7].

Definition 3. A symmetry σ is called *active* under assignment α if $\sigma(\alpha) = \alpha$.

Which leads to the following proposition:

Proposition 4. *Let T be a theory and α an assignment. If σ is a symmetry of T that is active under α, then σ is also a symmetry of $T \cup \alpha$.*

Proposition 4 states that the symmetries of T active under assignment α form a subset of the symmetries of $T \cup \alpha$. By Corollary 2, we can conclude that if a symmetry σ of T is active under assignment α, and a literal l is propagated, we are also allowed to propagate $\sigma(l)$. Since the composition of two symmetries of $T \cup \alpha$ is again a symmetry of $T \cup \alpha$, we can also propagate $\sigma^2(l), \sigma^3(l), \ldots$ After doing so, σ will again be active, so for other propagated literals l', $\sigma(l')$ can again be propagated, which results in dynamic symmetry breaking. Many dynamic symmetry breaking methods in CP use this property [8], [7].

We improve this approach by introducing the notion of *weakly active* symmetries, a notion that generalizes activity.

Definition 5. Given a theory T, let $(\alpha, \delta, expl)$ be the state of a solver. A symmetry σ of T is *weakly active* for assignment α and choice literals δ if $\sigma(\delta) \subseteq \alpha$.

We now show that a literal $\sigma(l)$ is a logical consequence of a theory $T \cup \alpha$ if l is a logical consequence of $T \cup \alpha$ and σ a weakly active symmetry of T under α.

Proposition 6. *Let T be a theory and α an assignment. If there exists a subset $\delta \subseteq \alpha$ and a symmetry σ of T such that $\sigma(\delta) \subseteq \alpha$ and $T \cup \delta \models T \cup \alpha$, then σ is also a symmetry of $T \cup \alpha$.*

Proof: Since σ is a symmetry of T, we know that it commutes with negation, thus all we need to prove is that $T \cup \alpha$ and $\sigma(T) \cup \sigma(\alpha)$ are logically equivalent.

Because σ is a symmetry of T, $\sigma(T)$ and T are logically equivalent. Combining this with the fact that $T \cup \delta \models T \cup \alpha$ allows us to derive $T \cup \sigma(\delta) \models T \cup \sigma(\alpha)$, and because $\sigma(\delta) \subseteq \alpha$, $T \cup \alpha \models T \cup \sigma(\alpha)$. If we let $\alpha' = \sigma(\alpha)$ and $\delta' = \sigma(\delta)$, then α' and δ' satisfy the same conditions as α and δ respectively in the beginning of the proof. With the same reasoning, we find $T \cup \sigma(\alpha) \models T \cup \sigma^2(\alpha)$. Continuing this way, we find the chain

$$T \cup \alpha \models T \cup \sigma(\alpha) \models T \cup \sigma^2(\alpha) \models \cdots \models T \cup \sigma^{n-1}(\alpha) \models T \cup \alpha,$$

with n the order of σ. As a result, $T \cup \alpha$ and $T \cup \sigma(\alpha)$ are logically equivalent. ■

Note that if $(\alpha, \delta, expl)$ is the state of a solver, and σ is weakly active, the conditions in Proposition 6 are satisfied. Combined with Corollary 2, this implies that for every propagated literal l, $\sigma(l)$ can also be propagated.

Weak activity has two advantages when compared to activity. The first is that weak activity is more general than activity, which allows for more propagations. The second is that keeping track of weakly active symmetries is easier than keeping track of active symmetries, since we only need to check for every choice literal l in δ whether $\sigma(l) \in \alpha$. We describe an efficient incremental approach to keep track of weakly active symmetries in Sect. III-B.

To conclude this theoretical section, we derive from Proposition 6 a corollary telling us what set of literals is a logical consequence from a theory given a set of weakly active symmetries. It also shows another link between activity and weak activity.

Corollary 7. *Let T be a theory, and $(\alpha, \delta, expl)$ the state of a solver for T. Suppose $\mathcal{S}$ is a set consisting of weakly active symmetries under α and δ. Furthermore, let $G_{\mathcal{S}}$ be the group generated by $\mathcal{S}$, and β the assignment consisting of the union of the orbits under $G_{\mathcal{S}}$ of all literals of α. Then $T \cup \alpha \models T \cup \beta$, and all symmetries in $G_{\mathcal{S}}$ are active under β.*

Proof: We have proven in Proposition 6 that all $\sigma \in \mathcal{S}$ are symmetries of $T \cup \alpha$. The first statement then follows from the fact that all symmetries of $G_{\mathcal{S}}$ are symmetries of $T \cup \alpha$. The second statement follows from the construction of β. ■

Corollary 7 shows that potentially many literals can be propagated by SP if during search the group generated by the weakly active symmetries is big.

B. The SP Algorithm

As mentioned before, we characterize the state of a CDCL solver by a current assignment α, a set of choice literals δ, and an explanation clause function $expl$. Symmetry is represented in the solver by a set of *input symmetries* $\mathcal{S}$, given to the solver at the beginning of the search.

At the start of the search, $\alpha = \emptyset$, so every input symmetry in $\mathcal{S}$ is weakly active. Every time a literal is added to α, all input symmetries containing that literal are notified using a watched literal scheme, so the symmetries can update their weak activity status accordingly. This updating is implemented by keeping a counter per symmetry σ, indicating the minimum number of literals to be added to α to make σ weakly active. Initially, the counter is 0, signifying the corresponding symmetry σ is weakly active. The counter is increased whenever a literal l with $l \in \delta$ and $\sigma(l) \notin \alpha$ is added to α, and decreased whenever a literal l is added to α such that $\sigma^{-1}(l) \in \delta$. This way, updating the weak activity status of a symmetry is a constant time operation.

For every input symmetry σ, SP also keeps track of the *first asymmetric literal* for σ under assignment α. If such a literal exists, the first asymmetric literal is the oldest literal $l \in \alpha$ for which $\sigma(l) \notin \alpha$. Whenever σ is weakly active, according to Proposition 6 and Corollary 2, $\sigma(l)$ can be propagated. We will refer to this type of propagation as *symmetry propagation*, as opposed to unit propagation by unit clauses.

Modern CDCL solvers require that for every propagated literal l an explanation clause exists, which must contain l, and must be a unit clause when l is propagated. Fortunately, creating an explanation clause for a symmetry propagation is straightforward. A literal $\sigma(l)$ will only be propagated as a symmetry propagation if l is the first asymmetric literal for some weakly active symmetry σ under a solver state $(\alpha, \delta, expl)$. Hence, for all false literals $l' \in expl(l)$, $\sigma(\neg l') \in \alpha$. Taking into account that σ commutes with negation, $\neg\sigma(l') \in \alpha$, so $\sigma(expl(l))$ is a unit clause under α. Since $\sigma(expl(l))$ contains $\sigma(l)$, we can use it as an explanation clause for the propagation of $\sigma(l)$. By Proposition 1, we know this clause is a logical consequence of the original theory, so we can add it to the learned clause store to use it for future propagations.

During the propagation phase of the SAT solver, unit propagation and symmetry propagation are executed alternately. More precisely: when no more unit propagations are possible, the SP algorithm loops over the set of input symmetries in search of a weakly active symmetry σ which still has a first asymmetric literal l under the current assignment. If such a symmetry exists, symmetry propagation of $\sigma(l)$ occurs, after which unit propagation is immediately reactivated. This cycle continues until no more unit and symmetry propagations can be made, or a conflict occurs.

By switching back to unit propagation after every single symmetry propagation, SP makes sure all literals propagated by symmetry propagation could not have been propagated by unit propagation. As a consequence, all explanation clauses for symmetry propagations did not occur in the original theory or the learned clauses store, so no time is wasted constructing existing clauses. Also, the total amount of learned clauses is minimized, which is beneficial since the performance of CDCL solvers strongly correlates with the number of clauses they have to keep track of.

Algorithm 1 is a pseudocode representation of the propagation phase of a typical SAT solver using SP:

Algorithm 1. Given a theory T, an assignment α over variables of T and a set of symmetries $\mathcal{S}$ of T, a SAT solver using SP propagates literals as follows:

```
repeat
  execute unit propagation
  for each weakly active symmetry $\sigma$ in $\mathcal{S}$ do
    if a first asymmetric literal $l$ for $\sigma$ under $\alpha$ exists then
      $push\_back(\alpha, \sigma(l))$
      $expl(\sigma(l)) := \sigma(expl(l))$
      $push\_back(T, expl(\sigma(l)))$
      $break$ {go back to unit propagation}
    end if
  end for
until no new literals have been propagated or a conflict has occurred
```

Example 8 presents a typical run of the SP algorithm:

Example 8. Consider theory $T = \{\neg f \vee a, \neg f \vee b, \neg a \vee d, \neg b \vee e \vee c, \neg c \vee \neg g, \neg c \vee g\}$ and its symmetry $\sigma = (ab)(de)$. Suppose the SAT algorithm chooses a, so $\alpha = \delta = \{a\}$. During the unit propagation phase, we can propagate d, so $\alpha = \{a, d\}$, $\delta = \{a\}$ and $expl(d) = \neg a \vee d$. Since no more unit propagation is possible, a check for symmetry propagation is made. However, since $\sigma(\delta) = \{b\} \nsubseteq \alpha$, σ is not weakly active and no symmetry propagation occurs. Note that at this time, a is the first asymmetric literal for σ, since a is the first literal added to α and $\sigma(a) \notin \alpha$.

Since no more unit or symmetry propagation is possible, the algorithm chooses, say f, and propagates b during unit propagation. After this, $\alpha = \{a, d, f, b\}$, $\delta = \{a, f\}$, $expl(b) = \neg f \vee b$, and symmetry propagation starts. σ is inactive since $\sigma(d) = e \notin \alpha$, but σ is weakly active since $\sigma(\delta) = \{b, f\} \subseteq \alpha$. Also, the first asymmetric literal for σ now is d, since $\sigma(a) \in \alpha$ and $\sigma(d) \notin \alpha$. As a result, $\sigma(d) = e$ can be propagated during symmetry propagation, so that $\alpha = \{a, d, f, b, e\}$, and $expl(e) = \sigma(expl(d)) = \neg b \vee e$.

Now, unit propagation is immediately reactivated. Since symmetry propagation did not induce new unit clauses, no further unit propagation happens, and symmetry propagation is continued. Even though σ now is (weakly) active, it has no first asymmetric literal, so it can no longer propagate, ending the unit and symmetry propagation. The search now continues by choosing a new choice literal, and so on.

SP offers some interesting properties. Firstly, the calculations needed to perform symmetry propagation can not send the underlying solver in a loop, so SP preserves the completeness of the underlying solver.

Secondly, since all the literals propagated by symmetry propagation are a logical consequence of the original theory T and the partial assignment α, and since all the corresponding explanation clauses added to T are a logical consequence of T, SP also preserves the soundness of the underlying solver. This also implies that SP does not prohibit the underlying solver from finding any model. SP merely avoids visiting branches of the search tree symmetrical to failed branches, and as such it leaves the underlying solver heuristic free to choose the most optimal search branch.

Thirdly, during unit and symmetry propagation, the assignment α only grows, while the set of decision literals does not change. Hence, the set of weakly active symmetries $\mathcal{S}$ can only increase in this phase. Also, for all weakly active symmetries $\sigma, \sigma' \in \mathcal{S}$, after $\sigma(l)$ is propagated, $\sigma'(\sigma(l))$ will also be propagated – either by unit propagation or by symmetry propagation. As a consequence, all propagations allowed by Corollary 7 will take place.

C. Optimizations to SP

With the main ideas of SP being clear, we will discuss two optimizations to SP. In Sect. IV, we show that they improve performance. The first optimization is based on the interaction of SP with *inverting* symmetries.

Definition 9. A literal l is *inverting* for a symmetry σ if $\sigma(l) = \neg l$. A symmetry σ is *inverting* if at least one literal is inverting for σ.

Whenever an inverting symmetry σ is weakly active for assignment α and choices δ, and one of its inverting literals l is propagated, SP will propagate $\neg l$, resulting in a conflict and thus, a backjump in the search. However, if an inverting literal l for σ would become a choice literal, σ would become weakly inactive and remain so until the solver backtracks over l. By choosing choice literals in such a way that they are inverting for as few symmetries as possible, we can keep as many inverting symmetries weakly active as long as possible. In our implementation, we simply ordered the variables by the number of symmetries the corresponding literals were inverting for, and used this as the initial variable ordering by which literals were selected to become choice literals. Note that this *inverting symmetry optimization* has no effect if no inverting symmetries are present in the theory.

A second optimization concerns symmetry propagation when a symmetry σ is not weakly active. In this case, it still is possible that for some propagated literal l, the clause $\sigma(expl(l))$ is a unit clause. Since by Proposition 1 this clause is always a logical consequence of the original theory, the non-false literal $l' \in \sigma(expl(l))$ can be propagated using $\sigma(expl(l))$ as the explanation clause. We implemented this idea by a simple check after unit propagation and (weakly active) symmetry propagation could propagate no more. More precisely, we loop over all inactive symmetries σ and all propagated literals $l \in \alpha \setminus \delta$ to check whether $\sigma(expl(l))$ is a unit clause. If so, the appropriate propagation is made, after which unit propagation is immediately reactivated. Despite the incurred overhead of this *inactive propagation optimization*, it still gave good results in our experiments.

The inactive propagation optimization is a special case of dynamic symmetry breaking by Proposition 1, and since it does not depend on the activity status of a symmetry, it generalizes SP. This does not make the notion of weak activity useless: if a symmetry σ is weakly active, by Corollary 7, we can propagate $\sigma(l)$ for each propagated literal l and skip the check of whether $\sigma(expl(l))$ is a unit clause. Also, the inactive propagation optimization requires a solver to incorporate an explanation clause mechanism, which plain SP does not require.

IV. Experiments

To test the algorithms outlined in the previous section, we implemented SP in the CDCL solver Minisat [4] released on GitHub on March 27th 2011 [9]. The implementation follows the algorithm described in Sect. III-B, with the option to use the inverting symmetry optimization or inactive propagation optimization described in Sect. III-C. We tested the SP-implementation with different combinations of these options, of which we will present two here. The first version has both optimizations deactivated. We refer to this regular version by *Minisat+SP^{reg}*. The second version has both optimizations activated. We refer to this optimized version by *Minisat+SP^{opt}*. We refer to both versions by *Minisat+SP*. The source code of Minisat+SP is available on GitHub as a branch of Niklas Sörensson's Minisat solver [9].

We compare the performance of our solvers with Shatter [10] as reference. Shatter is a static symmetry breaking tool which in a preprocessing step adds symmetry breaking clauses to a theory, on which we can then run Minisat. We will refer to this as the *Minisat+Shatter* approach. We use Shatter as reference because it is easy to use, freely available, and because Shatter currently is the most effective approach for exploiting symmetry in SAT [11].

We also include measurements with plain Minisat in our statistics, to give an idea how important symmetry breaking is in each selected benchmark.

As benchmarks, we used SAT theories modeled in the standard DIMACS .cnf format. Since this format does not contain information about symmetries, we detect symmetry by using Shatter's builtin symmetry detection algorithm. For this, Shatter first converts a theory T to a graph, and then uses Saucy 2.0 [12] to detect the graph's automorphism group. The generators of the graph automorphism group are then converted to $\mathcal{S}$, a set of generators of the detected symmetry group of T. We used these generators as input symmetries for all symmetry breaking algorithms. An improved version of Saucy has been developed [13], but is not freely available.

We constructed two benchmark sets to test the algorithms. The first benchmark set was constructed by running Shatter on the problems of the SAT 2011 competition benchmark set. If we could detect within a time limit of 1000 seconds that a certain problem exhibited symmetry, we included this problem in the first benchmark set. In the end, the first benchmark set contained 96 problems, of which the results are summarized in Fig. 1.

The second benchmark set consists of classical SAT symmetry breaking problems: problems of wire routing in the channels of field-programmable integrated circuits (**chnl** and **fpga**) [14], pigeonhole problems (**hole**) and Urquhart's problems (**Urq**) [15]. We use a shuffled version of the pigeonhole problem, because we experienced that using the initial variable ordering resulted in very fast solving times for both Minisat+SP and Minisat+Shatter, which might not be representative for the expected performance of both algorithms. The results of the second benchmark set are described in Table I.

Two problem families from the first benchmark set (**x** and **battleship**) gave particularly interesting results, so we also included their information in Table I. Note that all problems in Table I contain no inverting symmetries, except for **Urq** and **x**, which contain only inverting symmetries.

For both benchmark sets, full experimental results are available online, as well as .cnf files for all problems in

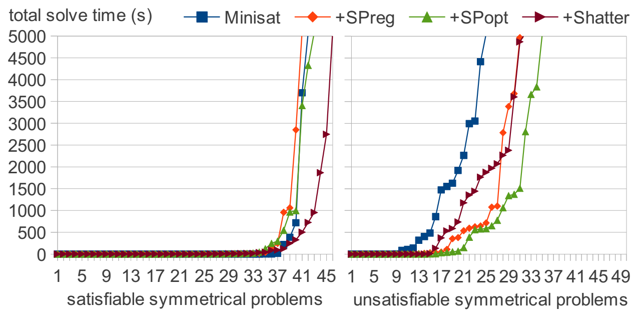

Figure 1. The performance of Minisat, Minisat+SP^{reg}, Minisat+SP^{opt} and Minisat+Shatter on 96 symmetry exhibiting problems of the SAT 2011 competition. For each algorithm, the problems are ordered on solve time.

Table I [9]. For all problems of the SAT 2011 competition, .cnf files are available on the corresponding site [16].

The total solve time given to each algorithm on each problem was 5000 seconds, including the time needed to detect symmetry for all algorithms except Minisat. The symmetry detection time was typically very low: less than 4 seconds for all problems in Table I, and less than 100 seconds for all but 12 problems of the first benchmark set.

When no answer was given in the desired time limit, a "-" is shown in Table I. The problems were solved using an Intel Core i7-2600 @ 3.4GHz processor and 16 GiB of memory, with Ubuntu 10.04 64 bit as operating system.

The running times in Fig. 1 on problems of the first benchmark set show some interesting patterns. Firstly, on unsatisfiable symmetrical problems, adding symmetry information and exploiting it significantly improves performance compared to Minisat. We also learn that for unsatisfiable symmetrical problems, Minisat+Shatter and Minisat+SP^{reg} solve about the same number of problems, with Minisat+SP^{reg} being a bit faster on average. The best performing algorithm on the unsatisfiable instances clearly is Minisat+SP^{opt}.

The left part of Fig. 1 sketches another picture. Firstly, all algorithms are able to solve almost all satisfiable symmetric problems. This can be explained by the fact that when a problem is both satisfiable and symmetric, it often contains many (symmetric) solutions, which as a result are less hard to find. Secondly, Minisat+SP^{reg} and Minisat+SP^{opt} both perform similarly to Minisat, while Minisat+Shatter on the other hand is able to deliver improved performance. An explanation can be found in the difference between SP and Shatter: the symmetry breaking constraints generated by Shatter will always exclude some part of the search space, while SP can not guarantee that symmetry propagation (and corresponding search space reduction) will always happen. When symmetries are inactive or have no first asymmetric literal, they will not propagate. In this sense, Shatter is a more complete symmetry breaking tool, still breaking significant symmetry when solving relatively easy satisfiable symmetrical problems.

The results presented in Table I concerning the classical SAT symmetry breaking problems confirm the above observations: the satisfiable instances of **fpga** are easily solved, Minisat+SP^{reg} has performance similar to Minisat+Shatter, and Minisat+SP^{opt} performs best. Note that Minisat is able to solve the satisfiable instances, but has great trouble with the unsatisfiable ones.

Since Minisat+SP^{opt} uses two different optimizations at the same time, the question remains whether both optimizations have complementary strengths. Since the inverting symmetry optimization has no effect when no inverting symmetries are present in the problem, the performance difference between Minisat+SP^{reg} and Minisat+SP^{opt} on the unsatisfiable **fpga**, **hole** and **battleship** problems (which contain no inverting symmetries) is due to the inactive propagation optimization. Further testing with only the inactive propagation optimization activated, showed no significant performance gain on **Urq** and **x** (which contain only inverting symmetries) compared to Minisat+SP^{reg}. As a result, we can conclude that the performance gain of Minisat+SP^{opt} on **Urq** and **x** is due to the inverting symmetry optimization.

V. Related Work

Benhamou et al. proposed a general dynamic symmetry breaking approach for SAT [2], based on Corollary 2: every time a SAT solver with theory T and assignment α backtracks from a certain choice literal l, a local symmetry group G of $T \cup \alpha$ is computed using Saucy, and the orbit of $\neg l$ under G is propagated. The drawback of this algorithm is that repeated computation of symmetries of $T \cup \alpha$ can be very expensive. SP avoids this overhead by only considering the symmetry group of $T \cup \alpha$ generated by weakly active input symmetries, which as shown in Sect. III-B can be implemented efficiently. Also, SP checks for symmetry propagations during every propagation phase, not only when the solver backtracks.

Other general symmetry breaking approaches for SAT are based on Proposition 1, such as the already mentioned Symmetrical Learning Scheme (SLS) [6]. A similar method has been proposed by Keur et al. [17], and is used to break symmetry in the context of boolean optimization [3].

Even though good results have been achieved using SLS, a disadvantage is that not all learned clauses are guaranteed to contribute to the search by propagating a literal. This might result in lots of useless clauses being added to the solver, without breaking a significant amount of symmetry. In contrast, every learned clause added to the solver by SP will always have propagated a literal at least once. SLS has another inefficiency: when learning clauses symmetrical to learned clauses, it is possible that an already known clause is added to the set of learned clauses. We know from Sect. III-B that the explanation clauses created by SP were not yet present in the learned clause store or the original theory.

A completely different dynamic symmetry breaking approach in SAT is SymChaff, a structure-aware SAT solver

Table I

PERFORMANCE OF MINISAT, MINISAT+SP^{REG}, MINISAT+SP^{OPT} AND MINISAT+SHATTER ON THE PROBLEM FAMILIES **FPGA**, **CHNL**, **HOLE**, **URQ**, **X**, **BATTLESHIP**. THE BEST RESULTS ARE GIVEN IN BOLD. THE NAME OF EACH PROBLEM REVEALS WHETHER OR NOT THE PROBLEM IS SATISFIABLE. AN "*" INDICATES INVERTING SYMMETRIES. THE LAST TWO COLUMNS SHOW THE AMOUNT OF SYMMETRY PROPAGATIONS AS A PERCENTAGE OF THE TOTAL AMOUNT OF UNIT AND SYMMETRY PROPAGATIONS.

Problem name	Input symmetries	Solve Time (s)				Decisions				Sym. prop. ratio	
		Minisat	+SP^{reg}	+SP^{opt}	+Shatter	Minisat	+SP^{reg}	+SP^{opt}	+Shatter	+SP^{reg}	+SP^{opt}
fpga10_8_sat	22	**0.0**	**0.0**	**0.0**	**0.0**	405	352	**320**	503	6.7%	**12.0%**
fpga10_9_sat	23	**0.0**	**0.0**	**0.0**	**0.0**	408	348	**316**	607	6.2%	**9.4%**
fpga12_11_sat	29	**0.0**	**0.0**	**0.0**	**0.0**	467	**363**	524	474	4.8%	**8.1%**
fpga12_8_sat	24	**0.0**	**0.0**	**0.0**	**0.0**	**296**	324	472	602	7.0%	**8.8%**
fpga12_9_sat	25	**0.0**	**0.0**	**0.0**	**0.0**	451	**378**	492	713	5.9%	**7.8%**
fpga13_10_sat	28	**0.0**	**0.0**	**0.0**	**0.0**	**278**	477	396	591	5.0%	**6.1%**
fpga13_12_sat	32	**0.0**	**0.0**	**0.0**	**0.0**	**316**	334	365	1112	2.5%	**6.1%**
fpga13_9_sat	26	**0.0**	**0.0**	**0.0**	**0.0**	**330**	858	696	601	2.4%	**4.9%**
fpga10_11_uns_rcr	38	71.5	**0.1**	**0.1**	0.2	5985130	17093	**6888**	15152	2.1%	**4.3%**
fpga10_12_uns_rcr	41	209.8	**0.1**	**0.1**	0.2	12881772	12154	**5530**	14446	2.5%	**3.5%**
fpga10_13_uns_rcr	42	955.2	0.2	**0.1**	0.2	42096627	25671	**4938**	16921	1.6%	**3.6%**
fpga10_15_uns_rcr	46	1841.0	**0.2**	**0.2**	**0.2**	60680481	19039	**9183**	13249	2.3%	**3.2%**
fpga10_20_uns_rcr	57	1271.8	0.4	**0.2**	0.4	29871337	21404	**4498**	16306	1.8%	**4.6%**
fpga11_13_uns_rcr	44	-	**0.3**	**0.3**	0.9	-	25790	**19236**	56623	2.8%	**4.2%**
fpga11_14_uns_rcr	46	-	0.6	**0.2**	0.8	-	51389	**10843**	48327	1.7%	**3.3%**
fpga11_15_uns_rcr	49	-	0.3	**0.2**	0.6	-	29983	**9836**	32194	1.9%	**3.4%**
fpga11_20_uns_rcr	59	-	0.8	**0.3**	1.3	-	45832	**9746**	54054	1.8%	**3.9%**
chnl10_11-uns	39	167.2	**0.0**	**0.0**	**0.0**	12437023	**46**	**46**	1473	**28.5%**	**28.5%**
chnl10_12-uns	41	85.0	**0.0**	**0.0**	**0.0**	6050506	**46**	**46**	1682	**25.9%**	**25.9%**
chnl10_13-uns	43	94.8	**0.0**	**0.0**	**0.0**	6006811	**46**	**46**	1766	**23.7%**	**23.7%**
chnl11_12-uns	43	3353.1	**0.0**	**0.0**	**0.0**	127407652	**56**	**56**	2017	**29.0%**	**29.0%**
chnl11_13-uns	45	3868.6	**0.0**	**0.0**	**0.0**	126818865	**56**	**56**	2264	**26.4%**	**26.4%**
chnl11_20-uns	59	3405.1	**0.1**	**0.1**	**0.1**	58302347	**56**	**56**	3954	**16.5%**	**16.5%**
hole010_shuffled-uns	19	114.3	0.3	**0.1**	**0.1**	12675295	33740	**10949**	17342	1.7%	**4.2%**
hole011_shuffled-uns	21	3320.3	0.5	**0.3**	1.4	193488347	55047	**18662**	142711	1.8%	**3.3%**
hole012_shuffled-uns	23	-	4.1	**0.3**	10.5	-	313497	**30992**	729083	2.0%	**3.3%**
hole013_shuffled-uns	25	-	31.0	**0.8**	105.2	-	1532124	**67184**	4531023	1.8%	**3.3%**
hole014_shuffled-uns	27	-	311.2	**13.3**	2821.2	-	9836194	**765810**	67375809	1.6%	**3.5%**
hole015_shuffled-uns	29	-	715.7	**201.5**	-	-	17048418	**7299563**	-	1.4%	**2.7%**
hole016_shuffled-uns	31	-	-	**122.7**	-	-	-	**3884224**	-	-	**2.6%**
hole017_shuffled-uns	33	-	-	**2863.2**	-	-	-	**54961134**	-	-	**2.1%**
hole018_shuffled-uns	35	-	-	**994.5**	-	-	-	**18031344**	-	-	**2.1%**
Urq3_5-uns	29*	139.7	**0.0**	**0.0**	0.1	73481538	6124	**33**	91985	7.3%	**35.1%**
Urq4_5-uns	43*	-	**0.0**	**0.0**	39.0	-	1200	**43**	20323094	4.4%	**40.4%**
Urq5_5-uns	72*	-	7.0	**0.2**	3810.3	-	2963855	**72**	1428323031	2.4%	**42.0%**
Urq6_5-uns	109*	-	-	**0.6**	-	-	-	**139**	-	-	**27.3%**
Urq7_5-uns	143*	-	-	**1.3**	-	-	-	**145**	-	-	**37.5%**
Urq8_5-uns	200*	-	-	**3.3**	-	-	-	**205**	-	-	**39.9%**
x1_40.shuffled-uns	40*	141.4	**0.0**	**0.0**	3.1	72930235	29191	**100**	1686967	0.9%	**7.6%**
x1_80.shuffled-uns	80*	-	29.8	**0.1**	1972.4	-	11256887	**84**	680826737	0.6%	**17.4%**
battleship-07-13-sat	12	**0.0**	0.4	0.4	0.4	414	**237**	517	606	0.4%	**1.0%**
battleship-08-15-sat	14	**0.0**	**0.0**	**0.0**	**0.0**	277	**268**	429	1078	1.3%	**2.1%**
battleship-09-17-sat	15	**0.0**	0.1	0.1	0.1	**1395**	1773	5148	4203	1.1%	**1.6%**
battleship-10-17-sat	12	3.9	**1.4**	2.2	5.4	368958	**142969**	143459	211317	1.1%	**1.2%**
battleship-10-18-sat	13	**0.0**	**0.0**	0.1	0.1	**622**	659	6360	6348	0.6%	**1.4%**
battleship-10-19-sat	15	**0.0**	0.1	0.1	0.1	648	759	**462**	3293	**0.3%**	**0.3%**
battleship-12-23-sat	18	**0.0**	0.1	0.1	0.1	1252	3049	**1034**	4429	**0.8%**	0.8%
battleship-14-26-sat	17	718.2	1060.2	546.1	**14.3**	29589334	37732810	17825596	**735680**	1.0%	**1.1%**
battleship-15-29-sat	22	386.0	**16.5**	296.6	88.1	21961391	**744001**	10058234	3373200	**1.1%**	**1.1%**
battleship-24-57-sat	41	16.5	**2.8**	21.9	34.3	1706712	**223013**	779295	3102966	1.0%	**1.2%**
battleship-05-08-uns	8	**0.0**	**0.0**	**0.0**	**0.0**	9281	1406	463	**447**	1.9%	**2.1%**
battleship-06-09-uns	7	0.1	**0.0**	**0.0**	**0.0**	45893	7947	2358	**1105**	1.0%	**1.6%**
battleship-07-12-uns	10	485.1	17.3	2.0	**1.4**	61922751	3040267	295311	**141734**	1.4%	**1.5%**
battleship-10-10-uns	8	1.6	1.1	0.2	**0.0**	199396	107931	13673	**5017**	0.0%	**0.1%**
battleship-12-12-uns	10	402.3	45.6	**0.7**	1.3	29000352	2828621	**45112**	119304	0.1%	**0.3%**
battleship-14-14-uns	8	-	-	1372.2	**736.6**	-	-	61447105	**18278571**	-	**0.0%**
battleship-15-15-uns	10	-	-	**149.0**	-	-	-	**7252565**	-	-	**0.0%**
battleship-16-16-uns	17	-	-	**32.9**	-	-	-	**1476236**	-	-	**0.2%**

[18]. SymChaff keeps track of interchangeable sets of variables for which it only matters how many variables are true or false, and not which exact variables are true or false. It breaks symmetry by branching over the the number of interchangeable variables that are true instead of branching on the truth value of one of the variables. This is of course very efficient, but can only be used with special symmetries which make sets of variables interchangeable. For instance, SymChaff can not break the inverting symmetries of **Urq** and **x**, which makes it a less general symmetry breaking approach than SP. A similar argument can be made for dynamic symmetry breaking for graph coloring from Schaafsma et al. [19].

VI. Conclusion and Future Work

In this paper, we presented a novel approach to dynamically break symmetries in SAT. Thanks to the notion of weak activity, it possesses promising theoretical properties. An implementation in Minisat was able to outperform the current state-of-the-art Shatter on unsatisfiable symmetrical benchmarks, while the satisfiable symmetrical benchmarks appeared relatively easy to solve. Compared to other dynamic symmetry breaking techniques for SAT, we believe that SP is either more general or incurs less overhead. Furthermore, SP is not limited to SAT, but applicable to other problem solving techniques as well.

Besides achieving good performance, SP also opens up a number of research opportunities. A first one is to adjust the internal heuristics of a SAT solver to maximize the number of weakly active symmetries during search. For instance, when the solver decides upon the next choice literal, taking into account the amount of symmetries made (in)active by each candidate literal can improve the average number of weakly active symmetries. The variable reordering optimization from Sect. III-C is a simple step in this direction, already resulting in significant speedups.

Since SP at its core is not a clause generator, but a literal propagator in CP style, it should be straightforward to construct a symmetry breaking approach SP(CP) based on the ideas presented in this paper. The explanation clause generation mechanism for SP also is useful for lazy clause generation CP solvers. Evaluating the performance of such a SP(CP) implementation might yield interesting results.

Acknowledgment

Broes De Cat is funded by the Institute for Science and Technology Flanders (IWT).

References

[1] N. Narodytska and T. Walsh, "Dynamic versus static value symmetry breaking," in *11th International Workshop on Symmetry in Constraint Satisfaction Problems, SymCon'11 at CP'11*, 2011.

[2] B. Benhamou, T. Nabhani, R. Ostrowski, and M. R. Saïdi, "Dynamic symmetry breaking in the satisfiability problem," in *Proceedings of the 16th international conference on Logic for Programming, Artificial intelligence, and Reasoning*, ser. LPAR-16, Dakar, Senegal, April 25 - may 1, 2010.

[3] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Dynamic symmetry-breaking for boolean satisfiability," *Ann. Math. Artif. Intell.*, vol. 57, no. 1, pp. 59–73, 2009.

[4] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2004, vol. 2919, ch. 37, pp. 333–336.

[5] L. Zhang and C. F. Madigan, "Efficient conflict driven learning in a boolean satisfiability solver," in *In ICCAD*, 2001, pp. 279–285.

[6] B. Benhamou, T. Nabhani, R. Ostrowski, and M. R. Sadi, "Enhancing clause learning by symmetry in SAT solvers." in *ICTAI (1)*. IEEE Computer Society, 2010, pp. 329–335.

[7] C. Mears, M. Garcia de la Banda, B. Demoen, and M. Wallace, "Lightweight dynamic symmetry breaking," in *Eighth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'08)*, 2008.

[8] I. P. Gent and B. M. Smith, "Symmetry breaking in constraint programming," in *Proceedings of ECAI-2000*. IOS Press, 2000, pp. 599–603.

[9] J. Devriendt, "An implementation of Symmetry Propagation in Minisat on Github," www.github.com/JoD/minisat-SPFS.

[10] F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Shatter: efficient symmetry-breaking for boolean satisfiability." in *DAC'03*, 2003, pp. 836–839.

[11] K. A. Sakallah, "Symmetry and satisfiability," in *Handbook of Satisfiability*, 2009, pp. 289–338.

[12] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting Structure in Symmetry Detection for CNF," in *In Proceedings of the 41st Design Automation Conference*, 2004, pp. 530–534.

[13] H. Katebi, K. Sakallah, and I. Markov, "Symmetry and satisfiability: An update," in *Theory and Applications of Satisfiability Testing SAT 2010*, ser. Lecture Notes in Computer Science, O. Strichman and S. Szeider, Eds. Springer Berlin / Heidelberg, 2010, vol. 6175, pp. 113–127.

[14] G.-J. Nam, F. Aloul, K. A. Sakallah, and R. A. Rutenbar, "A comparative study of two boolean formulations of fpga detailed routing constraints," *IEEE Trans. Comput.*, vol. 53, pp. 688–696, June 2004.

[15] A. Urquhart, "Hard examples for resolution," *J. ACM*, vol. 34, pp. 209–219, January 1987.

[16] "The international SAT Competitions web page," www.satcompetition.org.

[17] A. Keur, C. Stevens, and M. Voortman, "CNF Symmetry Breaking Options in Conflict Driven SAT Solving," 2005.

[18] A. Sabharwal, "SymChaff: exploiting symmetry in a structure-aware satisfiability solver," *Constraints*, vol. 14, no. 4, pp. 478–505, Dec. 2009.

[19] B. Schaafsma, M. Heule, and H. van Maaren, "Dynamic symmetry breaking by simulating zykov contraction," in *Theory and Applications of Satisfiability Testing - SAT 2009*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed. Springer Berlin / Heidelberg, 2009, vol. 5584, pp. 223–236.