Exploiting Symmetries in MUS Computation

Ignace Bleukx[1], Hélène Verhaeghe[1,2], Bart Bogaerts[1,3], Tias Guns[1]

[1]KU Leuven, Dept. of Computer Science; Leuven.AI, Celestijnenlaan 200A, 3000 Leuven, Belgium
[2]UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium
[3]Vrije Universiteit Brussel, Dept. of Computer Science, Pleinlaan 9, 1050 Brussels, Belgium
ignace.bleukx@kuleuven.be, helene.verhaeghe@uclouvain.be, bart.bogaerts@kuleuven.be, tias.guns@kuleuven.be

Abstract

In eXplainable Constraint Solving (XCS), it is common to extract a Minimal Unsatisfiable Subset (MUS) from a set of unsatisfiable constraints. This helps explain to a user *why* a constraint specification does not admit a solution. Finding MUSes can be computationally expensive for highly symmetric problems, as many combinations of constraints need to be considered. In the traditional context of solving satisfaction problems, symmetry has been well studied, and effective ways to detect and exploit symmetries during the search exist. However, in the setting of finding MUSes of unsatisfiable constraint programs, symmetries are understudied. In this paper, we take inspiration from existing symmetry-handling techniques and adapt well-known MUS-computation methods to exploit symmetries in the specification, speeding-up overall computation time. Our results display a significant reduction of runtime for our adapted algorithms compared to the baseline on symmetric problems.

Code — github.com/ML-KULeuven/SymmetryMUS
Extended version — arxiv.org/pdf/2412.13606

1 Introduction

The field of eXplainable Constraint Solving (XCS) is a subfield of eXplainable AI (XAI) focused on explaining the solutions, or lack thereof, of constraint (optimization) problems. Explaining *why* a set of constraints does not admit a solution is often done through a *Minimal Unsatisfiable Subset* (MUS), i.e., an irreducible subset of the constraints rendering the problem unsatisfiable. This subset of constraints is then easier for a user to analyze than the full problem. MUSes are also used for debugging constraint models (Leo and Tack 2017), including minimization of faulty models when fuzz testing (Paxian and Biere 2023), explaining *why* an objective value is optimal (Bleukx et al. 2023) or to explain *why* a fact follows from the constraints (Bogaerts, Gamba, and Guns 2021). MUS computation techniques are well-studied and well-known in the XAI literature. Frequently used algorithms can be classified into *shrinking*[1] methods (Marques-Silva 2010); *divide-and-conquer* algorithms like QuickXplain (Junker 2001) and *implicit-hitting-set based* methods (Ignatiev et al. 2015).

[1]sometimes called *destructive* or *deletion-based*

In some cases, it can be useful to not just compute a single MUS but to *enumerate* a collection of MUSes or even all of them, as some MUSes may be easier understood by a user than others. Many algorithms for computing a set of MUSes rely on the "seed-and-shrink" paradigm (Bendík and Cerná 2020), where variants of the MARCO-algorithm are among the most popular techniques (Liffiton et al. 2016). MUS-computation and enumeration techniques have been discussed extensively (Marques-Silva and Mencía 2020; Gupta, Genc, and O'Sullivan 2021).

A prime concern for MUS computation and enumeration techniques is efficiency, especially for large problems, as checking if a valid assignment exists for a set of constraints can already be NP-hard (Biere et al. 2021). In practice, many techniques exist to speed up solving, e.g., by exploiting the problem structure. One of these, which has barely been studied in the context of MUS computation, is the exploitation of symmetries in the problem formulation.

Many real-world problems exhibit some kind of (variable and/or value) symmetry. For example, packing items into equivalent trucks, assigning shifts to equivalent workers or scheduling tasks on equivalent machines. Such symmetries can slow down combinatorial solvers, as they might have to consider all symmetric alternatives to an assignment. Solvers can exploit symmetries to prune the search space and/or speed up the search (Gent, Petrie, and Puget 2006; Sakallah 2021). Symmetry exploitation techniques are either *static* or *dynamic*. Static techniques involve adding *symmetry breaking constraints* to the specification before starting the solver. Dynamic techniques exploit the symmetries during search, e.g., by automatically learning symmetric clauses (Devriendt et al. 2012; Chu et al. 2014; Mears et al. 2014; Devriendt, Bogaerts, and Bruynooghe 2017), by modifying their branching behaviour (Fahle, Schamberger, and Sellmann 2001; Sabharwal 2005) or by generating symmetry breaking constraints during the solvers' execution (Metin et al. 2018). Furthermore, several tools exist for automatically detecting symmetries in the constraint specifications for a variety of input formats (Drescher, Tifrea, and Walsh 2011; Devriendt et al. 2016; Caudenberg and Bogaerts 2022; Anders, Brenner, and Rattan 2024).

While symmetry-handling has extensively been studied for methods solving constraint satisfaction or optimization problems, they have barely been studied in the context

of MUS computation or enumeration. "Classic" symmetries are defined on assignments of variables, while MUS-computation algorithms reason over subsets of constraints. Still, symmetries over variables can also lead to symmetries over constraints and, hence, over MUSes.

In this paper, we build on this observation and investigate how to discover and exploit symmetries for MUS computation and enumeration. Our contributions are the following: (i) we formally define symmetries in MUS problems; (ii) we show how existing symmetry detection tools can be used to detect constraint symmetries by means of a reformulation with half-reified constraints (iii) we show how to modify different types of MUS-computation algorithms to speed up the search for an MUS; and (iv) we evaluate the potential runtime improvements of symmetry breaking for MUS methods in an elaborate experimental evaluation.

2 Background

The methods presented in this paper are defined on constraints with Boolean variables only, but they can be generalized to richer constraint-solving paradigms such as SMT, MIP, or CP. In this section, we recall several essential concepts and introduce the notation used throughout this paper.

A literal ℓ is a Boolean variable x or its negation $\neg x$. We use ϕ to represent a set of constraints over Boolean variables. The set of literals in a constraint specification $\mathcal{L}(\phi)$ contains all variables occurring in ϕ and their negation. When ϕ is clear from the context, we simply write $\mathcal{L}$. An assignment α maps Boolean variables in ϕ to either true or false. An assignment is *total* if it assigns every variable in ϕ, and *partial* otherwise. Assignments can be represented as a subset of literals, namely those assigned to true (Audemard and Simon 2018). When a partial assignment μ can be *expanded* to a total one while satisfying ϕ, we write $\mu \approx\!\!\!\!\!\!\approx \phi$.

Modern SAT-solvers allow the use of assumption variables (Nadel and Ryvchin 2012; Hickey and Bacchus 2019). In combination with implication constraints, they can be used to test whether a subset of constraints is satisfiable. For each constraint c in ϕ, we introduce a Boolean indicator variable a_c and use this variable to construct the half-reification of c: $a_c \rightarrow c$ (Feydy, Somogyi, and Stuckey 2011). We refer to the set of indicators as $\mathcal{A}$ and, abusing notation, we refer to the constraint specification ϕ where all constraints are half-reified using variables a_c as $\mathcal{A} \rightarrow \phi$. Now, any full assignment of $\mathcal{A}$ (which is also a partial assignment of $\mathcal{A} \rightarrow \phi$), can be interpreted as a subset of constraints. Namely, a constraint c is part of the subset if its indicator variable a_c is set to true. So, when assuming a subset of literals $\mathcal{A}$ to be true, we can test whether a subset of constraints admits a valid assignment using a SAT-solver. Moreover, if no such assignment exists, SAT-solvers can return a sufficient set of assumption variables that cause unsatisfiability. Hence, we assume to have an oracle that takes as input a set of constraints ϕ and which returns a tuple $(\texttt{is\_sat}, \alpha, U)$. Here, `is_sat` is a Boolean flag indicating whether ϕ is satisfiable. When `is_sat` is set to true, α is the assignment found by the oracle, and $U \subseteq \phi$ is a sufficient subset of constraints causing unsatisfiability otherwise.

Constraint specifications can exhibit *symmetries*. We distinguish two types of symmetries: syntactic and semantic.

Definition 1 (Syntactic symmetry (Devriendt et al. 2016))**.** *Let π be a permutation of all literals $\mathcal{L}$ in constraints ϕ. A* syntactic symmetry *of ϕ, is a permutation π that commutes with negation (i.e., $\pi(\neg l) = \neg\pi(l)$), and that when applied to the literals in each of the constraints in ϕ, maps ϕ to itself.*

Definition 2 (Semantic symmetry (Devriendt et al. 2016))**.** *Let π be a permutation of all literals $\mathcal{L}$ in constraints ϕ. A* semantic symmetry *of ϕ is a permutation π that* commutes with negation *and that* preserves satisfaction *to ϕ (i.e., $\pi(\alpha)$ satisfies ϕ iff α satisfies ϕ).*

Any syntactic symmetry is also a semantic symmetry but not vice versa (Sakallah 2021). In practice, most symmetry-detection tools only detect syntactic symmetries (Devriendt et al. 2016), but all concepts in this paper are valid for semantic symmetries too, unless specified otherwise. Therefore, we refer to "symmetries" in general in the remainder of this paper. We write permutations using disjoint cycle notation. E.g., $\pi = (abc)(de)$ denotes the permutation with: $\pi(a) = b, \pi(b) = c, \pi(c) = a, \pi(d) = e$ and $\pi(e) = d$

Some structured symmetry groups can be summarized as a row-interchangeable symmetry (Flener et al. 2002; Devriendt, Bogaerts, and Bruynooghe 2014).

Definition 3 (Row-interchangeability)**.** *A matrix $M = (x_{rc})$ of literals describes a row-interchangeability symmetry group if for each permutation ρ of its rows, $\pi_\rho^M : x_{rc} \mapsto x_{\rho(r)c}$ is a symmetry of ϕ.*

The last required concept in this paper is that of Minimal Unsatisfiable Subsets (MUS) (Marques-Silva 2010).

Definition 4 (Minimal Unsatisfiable Subset)**.** *A Minimal Unsatisfiable Subset (MUS) of a set of constraints ϕ is a set $U \subseteq \phi$ that is unsatisfiable and for which any proper subset $U' \subsetneq U$ is satisfiable.*

Informally, an MUS is a minimal set of constraints that renders the problem unsatisfiable. Note that there may be several MUSes for a given unsatisfiable constraint problem.

3 Symmetries for the MUS problem

In this section, we define symmetries of the MUS problem. Solving the MUS problem requires reasoning over subsets of constraints, whereas the traditional setting of solving a satisfaction problem requires reasoning over assignments. Hence, symmetries for the MUS problem are symmetries of *constraints*.

Note that any syntactic symmetry of variables in a set of constraints induces a symmetry of constraints (Cohen et al. 2006), as illustrated below.

Example 1 (Pigeon hole problem)**.** *Consider a pigeon-hole problem $php(p, h)$ with p pigeons and h holes:*

$$\sum_{j=1}^{h} x_{ij} \geq 1 \qquad \forall i \in \{1, \ldots, p\} \qquad (P_i)$$

$$\sum_{i=1}^{p} x_{ij} \leq 1 \qquad \forall j \in \{1, \ldots, h\} \qquad (H_j)$$

Where x_{ij} are Boolean variables indicating whether pigeon i is assigned to hole j. We refer to the constraint that

ensures pigeon i is in a hole as P_i and the constraint ensuring at most one pigeon is in hole j is refered to as H_j.

Considering $php(4, 2)$, we identify four MUSes:

$$\{P_1, P_2, P_3, H_1, H_2\}, \{P_1, P_2, P_4, H_1, H_2\}$$
$$\{P_1, P_3, P_4, H_1, H_2\}, \{P_2, P_3, P_4, H_1, H_2\}$$

In the above specification, we notice several syntactic symmetries. E.g., $(x_{11}x_{21})(x_{12}x_{22})(x_{13}x_{23})(x_{14}x_{24})$ which in turn induces a symmetry of constraints: (P_1P_2).

3.1 Computing constraint symmetries

We propose to use existing symmetry-detection tools such as BREAKID (Devriendt et al. 2016), which traditionally detect symmetries of variable assignments, and transform the input to these tools to detect symmetries in constraints. In particular, we introduce indicator variables a_c to construct $\mathcal{A} \rightarrow \phi$ as defined in Section 2.

Before introducing symmetries that can be used when computing MUSes, we generalize the concept of symmetries of assignments to symmetries of partial assignments.

Definition 5 (Partial symmetries)**.** *Let $S \subseteq \mathcal{L}(\phi)$ be a subset of all literals $\mathcal{L}(\phi)$, closed under negation. A partial S-symmetry of a specification ϕ is a permutation of S that* commutes with negation *and that preserves satisfiability to ϕ. That is, for any assignment μ of S, $\mu \approx \phi$ iff $\pi(\mu) \approx \phi$.*

Some tools allow us to search for partial symmetries explicitly, but they can be derived from "classical symmetries" as in Definitions 1 and 2, as the following holds.

Proposition 1 (Deriving partial symmetries)**.** *Let S be a subset of $\mathcal{L}(\phi)$ closed under negation and π a symmetry of ϕ. If $\pi(S) = S$, then $\pi|_S$ is a partial S-symmetry of ϕ.*

We are now ready to fully define the concept of constraint symmetries as used in this paper.

Definition 6 (Constraint symmetry)**.** *A permutation π of the constraints in ϕ is called a* constraint symmetry *for ϕ if for any subset $C \subseteq \phi$, $\pi(C)$ is satisfiable iff C is satisfiable.*

The following proposition describes the exact relation between partial-symmetries and constraint symmetries.

Proposition 2. *Let π be a permutation of variables (that is a permutation of literals that does not cross polarity) $\mathcal{A}$ in $\mathcal{A} \rightarrow \phi$, then π directly maps to a permutation of constraints π_ϕ. Under this mapping, π is an $\mathcal{A}$-symmetry iff π_ϕ is a constraint symmetry.*

Intuitively, a subset of indicators "enables" a set of constraints and, when this initial set is (un)satisfiable, so is the set of constraints that is enabled by their symmetric image.

Hence, by computing partial symmetries of the specification $\mathcal{A} \rightarrow \phi$, we can compute constraint symmetries using any existing tool supporting it. We refer to the above method of finding constraint symmetries as CONSTRAINTSYMMETRIES(ϕ).

Following Proposition 3, constraint symmetries map MUSes to MUSes and non-MUSes to non-MUSes.

Proposition 3. *Given a constraint symmetry π of ϕ and a subset $U \subseteq \phi$. Then, $\pi(U)$ is an MUS of ϕ if and only if U is an MUS of ϕ.*

4 Constraint symmetries in MUS algorithms

Once detected, symmetries can be exploited to speed up the search for valid assignment(s) that satisfy ϕ. Static techniques include adding *symmetry breaking constraints* to the constraints specification before solving. Such constraints can speed-up the search by excluding a part of the search space. A well-known symmetry breaking constraint is the Lex-Leader constraint, which excludes, given an order of the variables, any assignment α if its symmetric counterpart $\pi(\alpha)$ is lexicographically below α. Symmetry detection tools can often generate a set of breaking constraints along with the generators for each detected symmetry group. Symmetry breaking constraints may break the symmetry completely or only partially (McDonald and Smith 2002).

Dynamic symmetry handling involves modifying the search algorithm of the solver itself to exploit the symmetries in the constraints. For example, by avoiding symmetric branches in the search tree or by modifying propagation algorithms to exploit symmetries in the problem specifically. While symmetry handling techniques have been extensively studied in the context of solving constraint programs, they do not directly apply to finding MUSes.

In particular, by adding a set of symmetry breaking constraints $\mathcal{B}$ to constraints ϕ, any MUS of the "broken specification" will be a subset of $\phi \cup \mathcal{B}$. This means that $\phi \cup \mathcal{B}$ can contain more MUSes than the original set of constraints, and new MUSes do not necessarily map to an original MUS.

In this section, we generalize symmetry handling techniques to constraint symmetries for use in MUS-finding algorithms. We explore both static symmetry breaking and techniques to exploit symmetries dynamically.

4.1 Symmetric transition constraints

One of the simplest classes of algorithms for finding an MUS are "shrinking" based methods (Marques-Silva 2010; Wieringa 2014). These methods iteratively drop a constraint c from ϕ, and call an oracle to check whether the remainder is (un)satisfiable. If the remaining core is still UNSAT, the constraint can safely be dropped from the formula. Otherwise, the constraint is required to ensure the unsatisfiability of the core and is marked as such (line 6 in Algorithm 1). When a constraint is marked as required, it is called a *transition constraint* (Belov and Marques-Silva 2011).

Definition 7 (Transition constraint)**.** *Given an unsatisfiable set of constraints U containing constraint c. If $U \setminus \{c\}$ is satisfiable, c is called a* transition constraint*.*

Belov, Lynce, and Marques-Silva (2012) propose an optimization to this simple approach called *clause set refinement*. This technique exploits the core found by the solver after it decides the input is unsatisfiable. In particular, the core returned by the solver may be smaller than the working core at that point in the algorithm. Hence, we can use the solver core U' to further shrink the working core. This technique is shown on line 13 in Algorithm 1.

To exploit symmetries in shrinking-based methods, we use the set of symmetry generators directly. In particular, we mark symmetric counterparts of transition constraints. We illustrate this in Example 2.

Algorithm 1: SYMM-SHRINK(ϕ, *recompute?*)

```
1: U ← φ; G ← CONSTRAINTSYMMETRIES(φ)
2: while there are unmarked constraints in U do
3:    c ← next unmarked constraint in U
4:    (is_sat, α, U') ← SAT(U \ {c})
5:    if is_sat then
6:       mark c as required
7:       if recompute? then
8:          G ← CONSTRAINTSYMMETRIES(U)
9:       for each π ∈ G do
10:         if π(U) = U then
11:            mark π(c) as required
12:   else
13:      U ← U'
14: return U
```

Example 2. *Following on Example 1, take U to be $\{P_1, P_2, P_3, H_1, H_2\}$ and H_1 and H_2 are the only constraints that are marked to be required by previous iterations. If the algorithm selects P_1 to be the next constraint to test, the oracle will report $U \setminus \{P_1\}$ is satisfiable. Hence, P_1 is a transition constraint. However, as all P_i are interchangeable for the MUS problem, this means also P_2 and P_3 are required, and they can be marked as such.*

To mark such symmetric transition constraints, we are interested in constraint symmetries mapping U to U.

Proposition 4 (Symmetric transition constraint)**.** *If π is a constraint symmetry of unsatisfiable formula $U \subseteq \phi$ and c a transition constraint for U, then $\pi(c)$ is also a transition constraint for U.*

Proof. As U is unsatisfiable and π is a constraint symmetry of U, $\pi(U)$ is a also unsatisfiable. Furthermore, as $U \setminus \{c\}$ is satisfiable, $\pi(U \setminus \{c\}) = \pi(U) \setminus \{\pi(c)\} = U \setminus \{\pi(c)\}$ is satisfiable and hence $\pi(c)$ is a transition constraint. □

Finding constraint symmetries of U can be done either by iteration over each permutation in the symmetry groups in $\mathcal{G}$ or by invoking the symmetry-detection tool again on the sub-problem U (when flag *recompute?* is true). The efficiency depends on the overhead of detecting symmetries and the structure of the global symmetry groups. Indeed, for large symmetry groups, iteration over each permutation may be infeasible or inefficient when only a subset of permutations map U to U. However, some symmetry-detection-tools can summarize structured symmetry groups using a matrix. For those matrices, we can easily find all of the symmetric images of a given variable within a given subset. Given a row-interchangeable symmetry described by matrix M, a core U and a (indicator) variable $a \in U$. Find the coordinate (i, j) of variable a in M. Now, find the indices of columns *cols* where, on row i, variables in U occur – clearly $j \in cols$. Then, iterate over each each row r in the matrix and define $cols_r$ to be the columns k where $x_{rk} \in U$. When $cols = cols_r$, then x_{rj} is a symmetric image of a in U.

To further reduce the overhead, we only project to symmetric transition constraints using the generators.

Note that our modification exploits symmetries without "breaking" them. That is, any MUS that may be returned by SHRINK may also be returned by SYMM-SHRINK.

Our approach shares similarities with *model rotation* (Belov and Marques-Silva 2011). Given a satisfiable subset of constraints to check $U \setminus \{c\}$, model rotation exploits the assignment α found by the oracle to $U \setminus \{c\}$. In particular, it searches for a variable assignment in α, such that, when the assignment is changed, c is satisfied and *exactly one* other constraint $c' \in U$ becomes unsatisfied. When such a variable assignment is found, we can be certain c' is a transition constraint for the MUS and marked as such.

Clearly, model rotation implicitly captures "simple" symmetries and hence resembles our approach. However, in SYMMSHRINK, the solution to $U \setminus \pi(c)$ may contain any number of changed variable assignments, whereas model rotation only changes a single assignment. Moreover, efficient model rotation requires the concept of a "flip-graph" which, to the best of our knowledge, can only be constructed easily for clausal input. As the algorithms described in this paper are more broadly applicable, we do not consider shrinking-based methods using model rotation as the baseline. In the future, we aim to explore the differences and similarities between model rotation and our approach further.

4.2 Symmetry breaking in MUS-computation

In this section, we investigate how symmetry breaking can be used to speed-up MUS computation. That is, instead of finding any MUS, we search for a lex-minimal MUS.

Example 3 (Lex-minimal MUS)**.** *Given the set of MUSes from Example 1, and the following constraint order:*

$$P_1 < P_2 < P_3 < P_4 < H_1 < H_2$$

Then $\{P_1, P_2, P_3, H_1, H_2\}$ is the only lex-minimal MUS.

Lex-minimal MUSes can be computed using the QuickXplain algorithm. QuickXplain takes as input a partial ordering of constraints and computes a preferred MUS based on that ordering. Hence, when providing an ordering mapping to the lex-leadership relation, QuickXplain finds a lex-minimal MUS. However, such ordering is often not reported by symmetry detection tools as they internally use the order to construct a set of breaking constraints, which can be used for static symmetry breaking. Therefore, we do not consider the QuickXplain algorithm here and instead focus on methods that can directly use symmetry breaking constraints returned by the detection tool. In particular, instead of finding any MUS, we search for an Optimal Constrained Unsatisfiable Subset (OCUS) (Gamba, Bogaerts, and Guns 2023).

Definition 8 (OCUS)**.** *Given a set of constraints ϕ, a cost function $f : 2^\phi \to \mathbb{N}$ and predicate $p : 2^\phi \to \{false, true\}$. Then $U \subseteq \phi$ is an OCUS with respect to f and p if:*

- *U is unsatisfiable*
- *$p(U)$ is true*
- *for all other unsatisfiable subsets $U' \subseteq \phi$ for which $p(U')$ holds, $f(U) \leq f(U')$*

In this paper, we use the concept of an OCUS to find an MUS with minimal cardinality by using a linear cost function f with equal weights. Combined with a set of constraint

Algorithm 2: SYMM-OCUS(ϕ, f, $dynamic?$)

```
1:  H ← ∅; G ← ComputeSymmetries(φ);
    B ← GetLexLeaderConstraints(G)
2:  while true do
3:      S ← Symm-CondOptHittingSet(H, f, B)
4:      (is_sat, α, U) ← SAT(S)
5:      if ¬is_sat then
6:          return (S, status)
7:      if dynamic? then
8:          K ← CorrSubsets(S, φ, G)
9:      else
10:         K ← CorrSubsets(S, φ)
11:     H ← H ∪ K
```

Algorithm 3: CORRSUBSETS($S, \phi, \mathcal{G}$)

```
1: K ← ∅; S' ← S; (is_sat, α, U) ← SAT(S')
2: while is_sat do
3:     C ← {c | c ∈ φ, c is unsatisfied by α}
4:     for π ∈ G do
5:         S' ← S' ∪ π(C)
6:         K ← K ∪ {π(C)}
7:     (is_sat, α, U) ← SAT(S')
8: return K
```

symmetry breaking constraints as predicate p, this ensures the resulting OCUS is a smallest, lex-minimal MUS of ϕ.

The only OCUS computation algorithm we are aware of (Gamba, Bogaerts, and Guns 2023) is a modification of the well-known *smallest MUS* algorithm, which is based on the hitting-set dualization between MUSes and MCSes (Ignatiev et al. 2015).

Definition 9 (Minimal Correction Subset)**.** *Given a set of constraints ϕ, a Minimal Correction Subset (MCS) is a subset $C \subseteq \phi$ for which $\phi \setminus C$ is satisfiable and for which any proper subset $C' \subsetneq C$ it holds that $\phi \setminus C'$ is unsatisfiable.*

Informally, an MCS is a minimal set of constraints which, when relaxed, render the problem satisfiable.

Proposition 5 (Hitting set dualization (Ignatiev et al. 2015))**.** *Given a set of constraints ϕ, let $MUSes(\phi)$ and $MCSes(\phi)$ be the set of all MUSes and MCSes of ϕ, respectively. Then, the following holds*

1. *A subset U of ϕ is an MUS if and only if U is a minimal hitting set of $MCSes(\phi)$; and*
2. *A subset C of ϕ is an MCS if and only if C is a minimal hitting set of $MUSes(\phi)$.*

The Implicit-Hitting-Set (IHS) algorithm to find an OCUS keeps a set H of correction subsets (initially empty). In each iteration, an optimal and constrained hitting set of H is computed, and the satisfiability of this set is checked using an oracle. If the hitting set is satisfiable, one or more correction subsets are generated from it using Algorithm 3 and are added to the sets to hit H.

This process is repeated until the optimal hitting set is UNSAT, and therefore, is an MUS. The pseudo-code for this algorithm is shown in Algorithm 2 (the $dynamic?$ parameter will be explained later).

To illustrate how breaking symmetries in IHS-algorithms may be useful, consider the following example.

Example 4 (Lex-minimal hitting set)**.** *Consider again $php(4, 2)$. After computing constraint symmetries, we can construct the following symmetry breaking constraints: $\{P_1 \vee \neg P_2, P_2 \vee \neg P_3, P_3 \vee \neg P_4, H_1 \vee \neg H_2\}$ which enforce a preference for lower-indice pigeon constraints and hole constraints. Imagine during the run of the algorithm, the sets to hit (i.e., the correction subsets enumerated by previous iterations) are $\{H_2\}$ and $\{P_3, P_4\}$*

Then, a minimal hitting set is $\{H_2, P_3\}$, which is satisfiable and thus clearly not an MUS. However, the smallest set that satisfies the breaking constraints, and that hits $\{H_2\}$ and $\{P_3, P_4\}$ is $\{P_1, P_2, P_3, H_1, H_2\}$. This subset is UNSAT and, indeed, an MUS, and hence the algorithm terminates.

4.3 Enumeration of symmetric MCSes

Implicit Hitting Set algorithms for computing MUSes are based on the hitting set dualization as stated in Proposition 5. They construct the set of minimal correction subsets lazily, by iteratively calling a hitting set solver and computing one or more correction subsets from the given hitting set. In general, the more correction subsets can be added in each iteration of the algorithm, the fewer iterations are required to find an MUS. Indeed, as shown by Ignatiev et al. (2015), enumeration of disjoint MCSes using for example Algorithm 3, significantly improves the algorithm's runtime.

Our contribution in this section also adds more MCSes in a single iteration of the algorithm. We propose to use the symmetry-groups detected directly by enumerating symmetric images of correction subsets (line 8 of Algorithm 2 when the $dynamic?$ flag is enabled).

We illustrate our idea in the following example:

Example 5. *Take again the running example of $php(4, 2)$. Imagine the sets to hit at some point in the algorithm are:*

$$\{H_1\}, \{P_1, P_2\}$$

Then, a minimal hitting set is $\{H_1, P_1\}$, and from this subset, we can compute a minimal correction subset, for example, $\{P_2, P_3\}$. As all pigeon constraints are interchangeable, we can find the symmetric versions of the above MCS:

$$\{P_1, P_2\}, \{P_1, P_3\}, \{P_1, P_4\}, \{P_2, P_4\}, \{P_3, P_4\}$$

By adding all of these correction subsets to the sets to hit, the hitting set computed in the next iteration of the algorithm is guaranteed to include at least 3 out of the 4 pigeon constraints, as required in any MUS for this problem.

Note that the number of symmetric MCSes may be exponential. Therefore, enumerating *all* of the MCSes can actually slow down the algorithm instead. Either because just enumerating them is hard, or because the hitting set solver is slowed down significantly by the surplus in sets to hit. In the experimental section, we evaluate several settings of the algorithm to investigate good values for the upper bound on the number of symmetric MCSes to add in each iteration.

In the implementation of the algorithm, we avoid regenerating MCSes by keeping track of the global set H.

Algorithm 4: LEX-MARCO(ϕ)

```
1: M ← ∅; G ← ComputeSymmetries(φ);
   B ← GetBreakingConstraints(G)
2: while M is satisfiable do
3:    S ← Symm-GetUnexplored(M, B)
4:    (is_sat, α, U) ← SAT(S)
5:    if is_sat then
6:       C ← Grow(S, φ)
7:       Block down C
8:    else
9:       U' ← Shrink(U)
10:      Block up U'
11: return U
```

The enumeration of extra MCSes and the addition of symmetry breaking constraints (Section 4.2) are complementary. That is, for some hitting sets, the lex-leader constraints do not yield a larger hitting set (e.g., such as the one in Example 5). However, as discussed above, the enumeration of MCSes also comes at a cost, so it might be better for some problems to use the lex-leader constraints in the hitting-set solver instead. The combination and comparison of both techniques are presented in the experimental section.

4.4 Symmetries for MUS-enumeration

The previous sections describe modifications to algorithms for computing *one* MUS. In some applications, users may be interested in a collection of MUSes instead. To this end, Liffiton et al. (2016) proposed the MARCO algorithm (Algorithm 4).

Similar to the IHS algorithm presented before, the MARCO algorithm is based on the hitting set dualization of MUSes and MCSes (Proposition 5) and explores the powerset of all constraints in an efficient way. In particular, instead of calculating a *minimal* hitting set to the set of correction subsets, MARCO computes *any* hitting set (line 3 in Algorithm 4), which is called the *seed*. Next, a SAT oracle is invoked to check whether the computed seed is satisfiable. If this is the case, the seed is used to calculate a correction subset, which is then added to the hitting-set solver. When the seed is unsatisfiable, it is shrunk further down to an MUS, and the hitting-set-solver is instructed to exclude any superset of the found MUS as next seeds.

When many symmetries are present in the constraint specification, the problem may contain many MUSes that are similar from a user perspective (Leo et al. 2024). To reduce the cognitive load on a user, we propose to use symmetry breaking constraints in the map-solver to specify a preference on the seed to compute from the map. This modification will only generate seeds that adhere to the lex-leadership relation defined by the symmetry breaking constraints, and hence the number of MUSes enumerated by the algorithm is reduced. When the full set of MUSes is required for the application at hand, it can be reconstructed in post-processing based on Proposition 3. This is similar to how symmetries are used when counting solutions to a satisfaction problem (Wang et al. 2020).

Note that any method may be used to grow and shrink the seed in the MARCO algorithm. Therefore, any symmetry-related optimizations to either shrinking or growing-algorithms may directly benefit the performance of MARCO as well. E.g., for shrinking, one can use Symm-Shrink (Algorithm 1) and for growing similar symmetry-inspired techniques can be devised.

5 Experiments

In this section, we evaluate our proposed modifications to algorithms for MUS-computation and enumeration on a set of benchmark instances. We aim to answer the following experimental questions:

- **EQ1** To what extent is making MUS-computation symmetry-aware beneficial in terms of runtime?
- **EQ2** How can symmetries be used for MUS enumeration?
- **EQ3** How do MUS-computation algorithms benefit from the detection of row-interchangeability symmetries?

We run all methods presented in this paper on unsatisfiable constraint problems encoded as pseudo-Boolean problems. Our benchmark consists of 272 instances with 146 pigeon-hole problems, 66 n+k-queens problems, and 60 bin-packing problems.

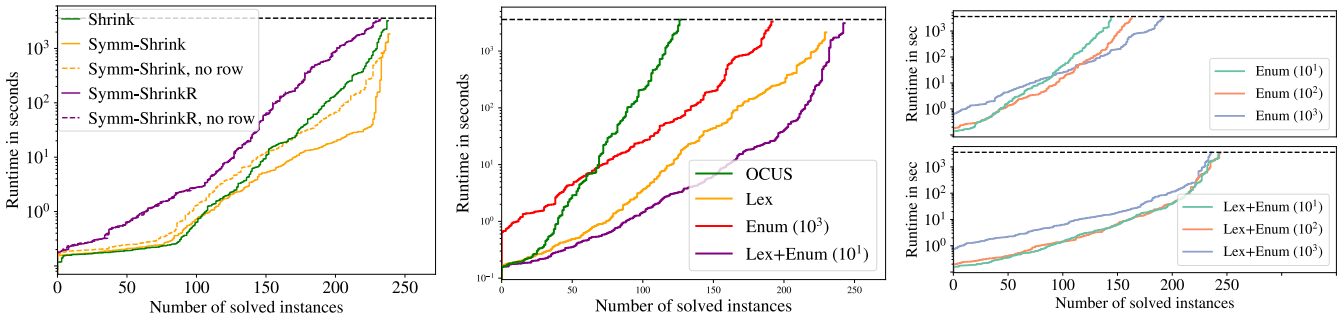
We implemented all MUS-finding algorithms on top of the CPMpy constraint modeling library (Guns 2019), version 0.9.20 in Python 3.10.14. Pseudo-Boolean solver Exact v1.2.1 (Devriendt 2023; Elffers and Nordström 2018) is used as SAT-oracle and Gurobi v11.0.2 as hitting-set-solver. Symmetries are computed using a custom branch of BreakID[2] (Devriendt et al. 2016). All methods were run on a single core of an Intel(R) Xeon(R) Silver 4214 CPU with 128GB of memory on Ubuntu 20.04. We used a time-out of 1h which includes symmetry-detection by BreakID and unrolling to symmetric MUSes in Lex-Marco.

5.1 MUS computation

Figure 1 shows the runtime of all methods for computing MUSes discussed in this paper. We first focus on Figure 1a, which compares the runtime of shrink-based methods. We compare the default algorithm (Shrink), the version removing symmetric transition constraints (Symm-Shrink), and its version when recomputing symmetries in each iteration of the algorithm (Symm-ShrinkR). For both symmetry-aware algorithms, we also compare the runtime when instructing BreakID to detect no row-interchangeability symmetries.

We can clearly see the removal of symmetric transition constraints is beneficial for the runtime of the algorithm as Symm-Shrink solves all instances between 5 and 10 times faster compared to the default. Still, the detection of row-interchangeability symmetries is essential for a successful implementation of this approach. For most instances in our benchmarks, recomputing symmetries in each iteration of the algorithm proves to be less efficient, even compared to the default algorithm (EQ3).

[2] on commit `4e9b15fd`



(a) Runtime for shrink-based methods

(b) Runtime for IHS-based methods

(c) MCS enumeration for IHS-based methods

Figure 1: Runtime for MUS-computation methods, measured across all 272 instances with a time-limit of 1h.

Next, we compare the runtime of each version of the IHS algorithms for computing OCUSes. We compare the default algorithm (OCUS), the addition of symmetry breaking constraints (Lex), the enumeration of symmetric MCSes (Enum(u)), and lastly, the combination (Lex+Enum(u)). Here, u is the upperbound on the number of MCSes added in each iteration. From Figure 1b, we see that any version of the algorithm performs better compared to OCUS, and the most efficient version uses a combination of symmetry breaking constraints and the enumeration of symmetric MCSes. Indeed, compared to the OCUS, Lex+Enum finds an MUS for almost double the number of instances.

Comparing the different versions of Enum, we notice an increase in runtime for easier instances when adding more MCSes. This can clearly be seen from Figure 1b and from the top of Figure 1c. Still, Enum can solve more instances as the number of MCSes increases, but there clearly is a limit to where this method can be pushed. When combining symmetry-breaking and MCS-enumeration, more MCSes do not yield in more instances solved, and the performance slightly degrades instead. This can be seen from the bottom of Figure 1c. This is due to the overhead in the hitting set solver when dealing with both the lex-leader constraints and the surplus in sets to hit. Hence, for Lex+Enum, it is better to keep the number of extra MCSes low.

Overall, we can conclude that making MUS-computation techniques aware of symmetries in the unsatisfiable problem has a positive impact on their runtime (EQ1).

5.2 MUS enumeration

We use the MARCO algorithm and Lex-MARCO with post-processing to unroll the set of lex-minimal MUSes to the full set. Both algorithms used the non-symmetric version of SHRINK for shrinking and the BLS procedure from Marques-Silva et al. (2013) for growing. Figure 2 shows the number of MUSes enumerated within 1h. From the left-hand side, it is clear the number of MUSes computed from the lex-minimal seeds is reduced for most of the instances compared to running vanilla MARCO. Instances found above the diagonal timed out for MARCO and the number of MUSes enumerated so far was smaller than the number of MUSes computed from the lex-minimal seeds by Lex-MARCO.

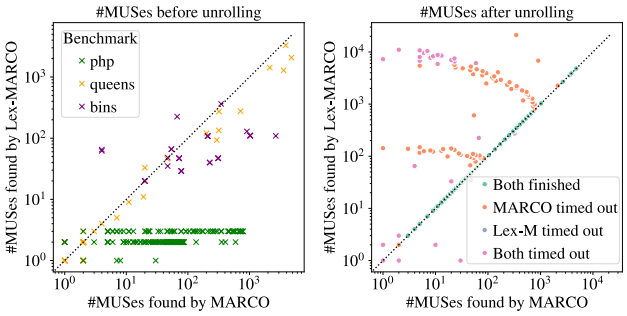


Figure 2: Number of MUSes enumerated within 1h

On the right-hand side of Figure 2, we show that the total number of MUSes after unrolling is higher compared to using the baseline MARCO algorithm. Indeed, Lex-MARCO completely enumerates the set of MUSes for 197 instances compared to MARCO completing 112. When both algorithms reach the timeout, MARCO may compute more MUSes as Lex-MARCO cannot start, or cannot complete unrolling within the given time-budget.

Overall, we can conclude that exploiting constraint symmetries can aid MUS enumeration by reducing the number of returned MUSes before unrolling and speeding up MUS enumeration when the full set is to be enumerated (EQ2).

6 Conclusion and outlook

In this paper, we propose various modifications to algorithms for finding MUSes inspired by symmetry breaking in SAT-and PB-solving. We demonstrated how to use off-the-shelf symmetry-detection tools to find symmetries in constraint specifications and how to use those symmetries in MUS computation techniques. Our results show that the presence of symmetry can indeed slow down MUS-finding algorithms, and symmetry handling methods are essential for finding MUSes quickly. This paper opens the door for future symmetry-inspired enhancements for computing MUSes, and as a next step, we plan to implement our methods into an existing state-of-the-art MUS finder. This allows us to evaluate our methods on competition benchmarks and compare them to other techniques, such as model rotation.

Acknowledgements

This work was partially supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G070521N), by the European Union (ERC, CertiFOX, 101122653 & ERC, CHAT-Opt, 01002802 & Europe Research and Innovation program TUPLES, 101070149). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

Anders, M.; Brenner, S.; and Rattan, G. 2024. satsuma: Structure-Based Symmetry Breaking in SAT. In *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, 4:1–4:23.

Audemard, G.; and Simon, L. 2018. On the Glucose SAT Solver. *Int. J. Artif. Intell. Tools*, 27(1): 1840001:1–1840001:25.

Belov, A.; Lynce, I.; and Marques-Silva, J. 2012. Towards efficient MUS extraction. *AI Commun.*, 25(2): 97–116.

Belov, A.; and Marques-Silva, J. 2011. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, 37–40. FMCAD Inc.

Bendík, J.; and Cerná, I. 2020. Replication-Guided Enumeration of Minimal Unsatisfiable Subsets. In *CP*, volume 12333 of *Lecture Notes in Computer Science*, 37–54. Springer.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2021. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. ISBN 978-1-64368-160-3.

Bleukx, I.; Devriendt, J.; Gamba, E.; Bogaerts, B.; and Guns, T. 2023. Simplifying Step-Wise Explanation Sequences. In *CP*, volume 280 of *LIPIcs*, 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Bogaerts, B.; Gamba, E.; and Guns, T. 2021. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artif. Intell.*, 300: 103550.

Caudenberg, D. V.; and Bogaerts, B. 2022. Symmetry and Dominance Breaking for Pseudo-Boolean Optimization. In *BNAIC/BENELEARN*, volume 1805 of *Communications in Computer and Information Science*, 149–166. Springer.

Chu, G.; de la Banda, M. G.; Mears, C.; and Stuckey, P. J. 2014. Symmetries, almost symmetries, and lazy clause generation. *Constraints An Int. J.*, 19(4): 434–462.

Cohen, D. A.; Jeavons, P.; Jefferson, C.; Petrie, K. E.; and Smith, B. M. 2006. Symmetry Definitions for Constraint Satisfaction Problems. *Constraints An Int. J.*, 11(2-3): 115–137.

Devriendt, J. 2023. Exact Solver.

Devriendt, J.; Bogaerts, B.; and Bruynooghe, M. 2014. BreakIDGlucose: On the importance of row symmetry in SAT. In *Proceedings 4th International Workshop on the Cross-Fertilization Between CSP and SAT*, 1–17.

Devriendt, J.; Bogaerts, B.; and Bruynooghe, M. 2017. Symmetric Explanation Learning: Effective Dynamic Symmetry Handling for SAT. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, 83–100. Springer.

Devriendt, J.; Bogaerts, B.; Bruynooghe, M.; and Denecker, M. 2016. Improved Static Symmetry Breaking for SAT. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, 104–122. Springer.

Devriendt, J.; Bogaerts, B.; Cat, B. D.; Denecker, M.; and Mears, C. 2012. Symmetry Propagation: Improved Dynamic Symmetry Breaking in SAT. In *ICTAI*, 49–56. IEEE Computer Society.

Drescher, C.; Tifrea, O.; and Walsh, T. 2011. Symmetry-breaking answer set solving. *AI Commun.*, 24(2): 177–194.

Elffers, J.; and Nordström, J. 2018. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. In *IJCAI*, 1291–1299. ijcai.org.

Fahle, T.; Schamberger, S.; and Sellmann, M. 2001. Symmetry Breaking. In *CP*, volume 2239 of *Lecture Notes in Computer Science*, 93–107. Springer.

Feydy, T.; Somogyi, Z.; and Stuckey, P. J. 2011. Half Reification and Flattening. In *CP*, volume 6876 of *Lecture Notes in Computer Science*, 286–301. Springer.

Flener, P.; Frisch, A. M.; Hnich, B.; Kiziltan, Z.; Miguel, I.; Pearson, J.; and Walsh, T. 2002. Breaking Row and Column Symmetries in Matrix Models. In *CP*, volume 2470 of *Lecture Notes in Computer Science*, 462–476. Springer.

Gamba, E.; Bogaerts, B.; and Guns, T. 2023. Efficiently Explaining CSPs with Unsatisfiable Subset Optimization. *J. Artif. Intell. Res.*, 78: 709–746.

Gent, I. P.; Petrie, K. E.; and Puget, J. 2006. Symmetry in Constraint Programming. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, 329–376. Elsevier.

Guns, T. 2019. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19.

Gupta, S. D.; Genc, B.; and O'Sullivan, B. 2021. Explanation in Constraint Satisfaction: A Survey. In *IJCAI*, 4400–4407. ijcai.org.

Hickey, R.; and Bacchus, F. 2019. Speeding Up Assumption-Based SAT. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, 164–182. Springer.

Ignatiev, A.; Previti, A.; Liffiton, M. H.; and Marques-Silva, J. 2015. Smallest MUS Extraction with Minimal Hitting Set Dualization. In *CP*, volume 9255 of *Lecture Notes in Computer Science*, 173–182. Springer.

Junker, U. 2001. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, volume 4. Citeseer.

Leo, K.; Gange, G.; de la Banda, M. G.; and Wallace, M. 2024. Automatic Core-Guided Reformulation via Constraint

Explanation and Condition Learning. In *AAAI*, 8065–8072. AAAI Press.

Leo, K.; and Tack, G. 2017. Debugging Unsatisfiable Constraint Models. In *CPAIOR*, volume 10335 of *Lecture Notes in Computer Science*, 77–93. Springer.

Liffiton, M. H.; Previti, A.; Malik, A.; and Marques-Silva, J. 2016. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2): 223–250.

Marques-Silva, J. 2010. Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper). In *ISMVL*, 9–14. IEEE Computer Society.

Marques-Silva, J.; Heras, F.; Janota, M.; Previti, A.; and Belov, A. 2013. On Computing Minimal Correction Subsets. In *IJCAI*, 615–622. IJCAI/AAAI.

Marques-Silva, J.; and Mencía, C. 2020. Reasoning About Inconsistent Formulas. In *IJCAI*, 4899–4906. ijcai.org.

McDonald, I.; and Smith, B. M. 2002. Partial Symmetry Breaking. In *CP*, volume 2470 of *Lecture Notes in Computer Science*, 431–445. Springer.

Mears, C.; de la Banda, M. G.; Demoen, B.; and Wallace, M. 2014. Lightweight dynamic symmetry breaking. *Constraints An Int. J.*, 19(3): 195–242.

Metin, H.; Baarir, S.; Colange, M.; and Kordon, F. 2018. CDCLSym: Introducing Effective Symmetry Breaking in SAT Solving. In *TACAS (1)*, volume 10805 of *Lecture Notes in Computer Science*, 99–114. Springer.

Nadel, A.; and Ryvchin, V. 2012. Efficient SAT Solving under Assumptions. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, 242–255. Springer.

Paxian, T.; and Biere, A. 2023. Uncovering and Classifying Bugs in MaxSAT Solvers through Fuzzing and Delta Debugging. In *POS@SAT*, volume 3545 of *CEUR Workshop Proceedings*, 59–71. CEUR-WS.org.

Sabharwal, A. 2005. SymChaff: A Structure-Aware Satisfiability Solver. In *AAAI*, 467–474. AAAI Press / The MIT Press.

Sakallah, K. A. 2021. Symmetry and Satisfiability. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 509–570. IOS Press.

Wang, W.; Usman, M.; Almaawi, A.; Wang, K.; Meel, K. S.; and Khurshid, S. 2020. A Study of Symmetry Breaking Predicates and Model Counting. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, 115–134. Springer.

Wieringa, S. 2014. *Incremental satisfiability solving and its applications*. Ph.D. thesis, Aalto University School of Science, Department of Computer Science and Engineering.