Symmetry Avoidance in MACE-Style Finite Model Finding

DOI:
10.1007/978-3-030-29007-8_1

Document Version
Accepted author manuscript

Published in:
Frontiers of Combining Systems

Symmetry Avoidance in MACE-Style Finite Model Finding ⋆

Giles Reger[1], Martin Riener[1], and Martin Suda[2]

[1] University of Manchester, Manchester, UK
[2] Czech Technical University in Prague, Czech Republic

Abstract. This work considers the MACE-style approach to finite model finding for (multi-sorted) first-order logic. This existing approach iteratively assumes increasing domain sizes and encodes the corresponding model existence problem as a SAT problem. The original MACE tool and its successors have considered techniques for avoiding introducing symmetries in the resulting SAT problem, but this has never been the focus of the previous work and has not received concentrated attention. In this work we formalise the symmetry avoiding problem, characterise the notion of a sound symmetry breaking heuristic, propose a number of such heuristics and evaluate them experimentally with an implementation in the Vampire theorem prover. Our results demonstrate that these new heuristics improve performance on a number of benchmarks taken from SMT-LIB and TPTP. Finally, we show that direct symmetry breaking techniques could be used to improve finite model finding, but that their cost means that symmetry avoidance is still the preferable approach.

1 Introduction

Finding finite models of first-order problems can be useful in a number of applications. The most prominent of these being in program verification, where models correspond to bug traces under most common program encodings. This paper considers an existing finite model finding technique and how it can be optimised to handle larger and more complex problems (which correspond to programs in the previous example application).

MACE-style finite model finding (introduced in [11] and extended in [4, 14]) aims to build finite models of first-order problems by reduction to SAT. The general idea behind this approach is as follows. To determine whether a (suitably preprocessed) first-order problem has a model of size n we first instantiate the problem with n fresh constants to produce a ground problem. This ground problem is then translated into a SAT problem such that a model of the SAT problem can be translated back to a model of the first-order problem. To find finite models we then iteratively repeat this process for larger values of n. A well known issue with this approach is that the encoding introduces inherent *symmetries* into the SAT problem. That is, if the SAT problem has a model then it actually has $n!$ isomorphic models for the different permutations of fresh constants. This can have a significant impact on the finite model finding process as to find a model

⋆ This work was supported by EPSRC Grant EP/P03408X/1. Martin Suda was supported by the ERC Consolidator grant AI4REASON 649043.

of size n in the iterative setting, we must first refute the preceding $n-1$ cases and this tends to be much harder in the presence of symmetries.

The problem of introducing symmetries in the encoding process is orthogonal to the well-known problem of identifying existing symmetries in the original problem. In the main part of this paper we look at *avoiding* introducing symmetries in our encoding. At the end of the paper we consider existing work on *identifying and breaking* symmetries. The starting point of our work is that the process of processing each produced SAT problem to identify symmetries (many of which we introduce ourselves) is likely to introduce unnecessary overhead. Our experimental analysis finds that, in general, this is true, but there may be something gained on some problems by exploring a close integration of these techniques into the finite model finding process.

Previous work has considered methods for *avoiding* symmetries in the SAT encoding, but the topic has not received concentrated attention. The main approach (also taken here) is to introduce additional constraints that restrict the ways in which elements of the model may be mapped to the fresh constants. The contributions of this paper are

- a characterisation of the symmetry avoidance problem in our context (Section 3). This is an extension of restricted functional symmetry from Paradox [4] which was previously stated in a restricted way and without proof;
- a number of heuristic symmetry breaking constraints (Section 4);
- an experimental evaluation using the Vampire theorem prover [10] demonstrating their effectiveness at speeding up the finite model finding process (Section 5);
- an experimental study examining the use of static symmetry breaking techniques in our process and comparing these to symmetry avoidance (Section 6).

Before we present these contributions we briefly revisit the definition of MACE-style finite model finding (Section 2).

A note on terminology. In this paper we have chosen to call the addition of additional constraints to *avoid* symmetries introduced by our own encoding *symmetry avoidance* as we are avoiding adding symmetries. This is in contrast to the act of symmetry breaking where inherent symmetries are identified and additional constraints added to break them. We note that prior work [4] used the term symmetry breaking for what we call symmetry avoidance.

2 MACE-Style Finite Model Building for First-Order Logic

In this section we describe the finite model finding procedure (in a single-sorted setting), which is a variation of the approach taken by Paradox [4]. Our presentation here follows the one given in our previous work extending this approach to the multi-sorted setting [14]. For simplicity, we only consider the single-sorted setting here (but our later results lift to the multi-sorted setting).

Given a first-order problem S, the general idea is to create, for each integer $n \geq 1$, a SAT problem that is satisfiable if and only if problem S has a finite model of size n. To find a finite model we iterate over the domain sizes $n = 1, 2, 3, \ldots$. Below we introduce the key conceptual details and the previous work [14] provides further examples.

First-Order Logic. We consider first-order logic with equality. A term is either a variable, a constant, or a function symbol applied to terms. A literal is either a propositional symbol, a predicate applied to terms, an equality of two terms, or a negation of either. The set of function and predicate symbols with associated arities defines the signature of a problem (constants are treated as function symbols with arity zero).

We assume all formulas are *clausified* using standard techniques (e.g. [12] and our recent work in [15]). A *clause* is a disjunction of literals where all variables are universally quantified (existentially quantified variables get replaced by Skolem functions during clausification). We assume familiarity with the notion of an *interpretation* and *model* of a set of clauses.

DC-Models. Let S be a set of clauses. Let us fix an integer $n \geq 1$. We extend the language by a set of distinct constants $DC = \{d_1, \ldots, d_n\}$ not occurring in S. We will call these *domain constants*. An interpretation is a *DC-interpretation*, if (i) its domain is DC and (ii) it interprets every domain constant as itself. Every model of S that is also a DC-interpretation will be called a *DC-model* of S. If S has a model of size n, then it also has a DC-model. We say that S is n-satisfiable if it has a model of size n.

A *DC-instance* of a clause C is a ground clause obtained by replacing every variable in C by a constant in DC. A clause with k different variables has exactly n^k DC-instances, where n is the current number of domain constants. Let us denote by S^* the set of all DC-instances of the clauses in S.

Theorem 1. *Let I be a DC-interpretation and C a clause. Then C is true in I if and only if all DC-instances of C are true in I.*

Principal Terms. We cannot yet encode the existence of models of size n as a SAT problem, as DC-instances can contain complex terms.[3] By a *principal term* we mean either a constant or an application of a function symbol of arity greater than zero to domain constants. A ground atom is called *principal* if it is either a predicate symbol applied to domain constants or an equality between a principal term and a domain constant. We lift this notion to literals.

Theorem 2. *Let I_1, I_2 be DC-interpretations. If they satisfy the same principal atoms, then I_1 coincides with I_2.*

Theorem 1 reduces n-satisfiability of S to the existence of a DC-interpretation of the set S^* of ground clauses. Theorem 2 shows that DC-interpretations can be identified by the set of principal atoms true in them. Next we introduce a propositional variable for every principal atom and reduce the existence of a DC-model of S^* to satisfiability of a set of clauses using only principal literals.

The SAT Encoding. The main step in the reduction is to transform every C into an equivalent clause C' such that DC-instances of C' consist (almost) only of principal literals (the exceptions are equalities between domain constants, which can be trivially

[3] An alternative to encoding the problem into SAT is to target the EUF logic and use an SMT solver instead. This approach has been explored by Vakili and Day [19].

removed). This transformation is known as *flattening* and ensures that all literals are of the form $p(x_1, \ldots, x_m)$, $f(x_1, \ldots, x_m) = y$, or $x = y$ or their negation. Every DC-instance of a flat literal is either a principal literal (for the first two cases), or an equality $d_i = d_j$ between domain constants. We produce the DC-instances of each flattened clause where we immediately remove inconsistent domain constant equalities and omit instances that are tautologous due to equalities between the same constants.

The DC-instances by themselves do not sufficiently constrain the SAT problem as they do not capture what it means to be a function. To do this we add two further kinds of constraints. For each principal term p and distinct domain constants d_i, d_j we produce the *functionality axiom* $p \neq d_i \vee p \neq d_j$. These clauses guarantee that all function symbols are interpreted as (partial) functions. For each principal term p we produce the *totality axiom* $p = d_1 \vee \ldots \vee p = d_n$. These clauses guarantee, together with functionality axioms, that all function symbols are interpreted as total functions.

The following theorem underpins the SAT-based finite model building method:

Theorem 3. *Let S be a set of flat clauses and S' be the set of clauses obtained from S^* as described above. More specifically, S' consists of 1) the non-tautologous DC-instances of the flattened versions of clauses in S^*, 2) the functionality axioms corresponding to the principal terms, and 3) the totality axiom corresponding to them. Then (i) all literals in S' are principal and (ii) S is n-satisfiable if and only if S' is propositionally satisfiable (when understanding principal atoms as propositional variables).*

3 Characterising Symmetry Avoidance

The SAT problem produced above necessarily contains many symmetries. In particular, every permutation of DC applied to a DC-model will give a DC-model, and there are $n!$ such permutations. This gives the SAT solver more work to do when refuting a model size (which it has to do $k-1$ times if the smallest model is of size k) as every possible interpretation needs to be refuted.

Isomorphic DC-Models. Let M be a DC-interpretation and σ a permutation of DC, i.e. a bijective function from DC to itself. There is always a DC-interpretation M_σ obtained by "relabelling" the domain constants in M according to σ such that σ is an isomorphism between M and M_σ.[4] For example, consider the clauses

$$f(f(x)) = x \qquad a \neq b$$

that have four possible DC-models captured by the following table

	a	b	$f(d_1)$	$f(d_2)$
1	d_1	d_2	d_1	d_2
2	d_1	d_2	d_2	d_1
3	d_2	d_1	d_1	d_2
4	d_2	d_1	d_2	d_1

[4] This means that for every function symbol f of arity a we have $M(f)(\mathsf{d}_1, \ldots, \mathsf{d}_a) = \mathsf{d}$ if and only if $M_\sigma(f)(\sigma(\mathsf{d}_1), \ldots, \sigma(\mathsf{d}_a)) = \sigma(\mathsf{d})$ and for every predicate symbol p of arity b we have $M(p)(\mathsf{d}_1, \ldots, \mathsf{d}_b)$ if and only if $M_\sigma(p)(\sigma(\mathsf{d}_1), \ldots, \sigma(\mathsf{d}_b))$.

where the first line captures the model M represented[5] by the set $\{a = d_1, b = d_2, f(d_1) = d_1, f(d_2) = d_2\}$. The last line is then M_σ for $\sigma = \{d_1 \mapsto d_2, d_2 \mapsto d_1\}$. Similarly, the models represented by lines 2 and 3 are isomorphic under σ.

We can now characterise what we want to achieve via symmetry avoidance: the removal of isomorphic interpretations. To appreciate the following definition, recall that no domain constant $\mathsf{d} \in DC$ occurs in S (but some may occur in the introduced constraint $\mathcal{C}$).

Definition 1 (Symmetry Avoidance). *A set of clauses $\mathcal{C}$ is said to be a* symmetry avoiding constraint *(SAC) if*

(i) not every DC-interpretation is a model of $\mathcal{C}$,
(ii) for every set of clauses S and for every DC-model M of S there is a permutation σ of DC such that M_σ is a DC-model of $S \cup \mathcal{C}$.

For the previous example the constraint $a = d_1$ would remove the isomorphic models represented by lines 3 and 4. This constraint satisfies (i) as we have two DC-interpretations that are not models of it, and (ii) if we focus on this particular set of clauses for S we can see that we have already identified the necessary σ.

The question is then what form the constraint $\mathcal{C}$ should take in general. Here we follow the work of Paradox [4]. We begin by assuming a total ordering on domain constants. We then fix a (finite) sequence of principal terms $\mathcal{P}$ and use this sequence to constrain the permissible DC-models. Let $\mathcal{P} = p_1, \ldots, p_m$. We want to restrict DC-models such that principal terms are assigned to domain constants *in order*, starting with $p_1 = d_1$. As S may imply equalities between principal terms we cannot straightforwardly assign $p_i = d_i$. Instead, we wish to specify that a principal term p_i is only interpreted as one of the first i domain constants, and, moreover, that the principal term p_i should only be interpreted as d_k if there is some principal term p_j such that $j < i$ and p_j is interpreted as d_{k-1}. This naturally leads to the addition of two kinds of clauses. The first kind is

$$p_i = d_1 \vee \ldots \vee p_i = d_i \tag{1}$$

for $i \leq \mathsf{min}(m, n)$. Notice that these take a form of strengthened totality constraints for the respective p_i. The second kind translates to

$$p_i \neq d_j \vee p_1 = d_{j-1} \vee \ldots \vee p_{i-1} = d_{j-1} \tag{2}$$

for $1 < i \leq m$ and $2 < j \leq i$.[6] Together these capture the above notion of order. Let $\mathcal{C}_\mathcal{P}$ be the set of all such clauses.

In our previous example, given $\mathcal{P}$ as $p_1 = a, p_2 = b$ we would add the clauses

$$a = d_1, \qquad b = d_1 \vee b = d_2,$$

which would exclude the models represented by lines 3 and 4 in the previous table. Note that in this case we did not need constraints of the second kind (as previously observed).

[5] Recall that a DC-interpretation can be identified by the set of principal atoms true in it.

[6] For $j = 2$ the clauses contain $p_1 = d_1$ which is always true given (1). For $j > i$ the literal $p_i \neq d_j$ and thus the corresponding constraint (2) follow from (1) and the functionality axioms.

Previously [4], this concept was introduced for ordering constants and extended to functions in a restricted sense. However, this previous work did not provide a proof that the approach is sound (does not exclude a possible model).

Let us, for the sake of clarity, also first consider the constant-only setting to later explain how it can be generalized.

Theorem 4. *Let $\mathcal{P} = p_1, \ldots, p_m$ be a non-empty sequence of constant symbols from the problem signature. Then $\mathcal{C}_\mathcal{P}$ is a symmetry avoiding constraint.*

Proof. We show both parts of Definition 1. For (i), notice that since $\mathcal{P}$ is non-empty, $\mathcal{C}_\mathcal{P}$ contains the unit clause $p_1 = d_1$ as an instance of (1) which is not satisfied by those DC-interpretations that do not map p_1 to d_1. For (ii), given a DC-model M of S we construct σ, a permutation of DC, such that the isomorphic model M_σ additionally satisfies $\mathcal{C}_\mathcal{P}$. We do this by describing a construction of the inverse mapping σ^{-1}. This is obviously equivalent, but makes the intuition more transparent.

Let us consider $\mathcal{P}_M = \{M(p_i) \,|\, p_i \in \mathcal{P}\}$, the set of domain constants that are interpretation by M of some element of $\mathcal{P}$, and denote its size by $k = |\mathcal{P}_M|$.[7] We set $\sigma^{-1}(d_1) = M(p_1)$ and for every $1 < i \leq k$ we pick $\sigma^{-1}(d_i) = M(p_j)$ for the smallest j such that $M(p_j)$ is not among $\{\sigma^{-1}(d_1), \ldots, \sigma^{-1}(d_{i-1})\}$. By construction, this function is injective and we can complete it to a permutation on DC, if necessary (i.e. if $k < n$), by arbitrarily "pairing up" the remaining $\{d_{k+1}, \ldots, d_n\}$ with the remaining values from $DC \setminus \mathcal{P}_M$. This construction implements the intuitive idea of using the smallest "unused" domain constant d_i for interpreting a term p_i unless it is in the model already taking a value of some "used" domain constant. It is easy to verify that M_σ satisfies both the constraints (1) and (2) and $\mathcal{C}_\mathcal{P}$ is therefore a SAC. □

The intuition for using general principal terms in $\mathcal{P}$ rather than just constants is that they provide another way of denoting domain elements in the model and may thus help us avoid further symmetries. E.g., we may not have enough constants, or the right constants. However, since non-constant principal terms directly refer to domain constants as arguments, we have an extra complication to deal with: while the construction from the proof of Theorem 4 is making sure it satisfies $\mathcal{C}_\mathcal{P}$ in M_σ, it is looking at the original model M to decide what to do with each next p_i. Thus its natural extension to non-constant terms cannot proceed, unless the arguments of p_i have already established value in M via the partially constructed σ^{-1}.

As an example of this complication, consider the one-element sequence $\mathcal{P}$ with $p_1 = f(d_1)$. Until we decide what d_1 from M_σ refers to in M, i.e. until we define $\sigma^{-1}(d_1)$, the construction cannot proceed.[8] Thus we pick $\sigma^{-1}(d_1)$ arbitrarily at which moment it becomes "used". But if f does not happen to map this domain constant to itself in M, i.e. if $M(f)(\sigma^{-1}(d_1)) \neq \sigma^{-1}(d_1)$, the smallest "unused" domain constant for p_1 in M_σ is d_2, i.e. we set $\sigma^{-1}(d_2) = M(f)(\sigma^{-1}(d_1))$. All in all, in this example, we can only restrict the symmetries by adding the following clause of the first kind (1) to $\mathcal{C}_\mathcal{P}$ on behalf of p_1:

$$f(d_1) = d_1 \vee f(d_1) = d_2,$$

[7] We necessarily have $k \leq m$ and $k < m$ implies $M(p_i) = M(p_j)$ for some $i \neq j$.

[8] Speculating what this value could be if we proceed anyway is an interesting direction for further research not covered in this paper.

but not the stronger $f(d_1) = d_1$. (It is easy to see how this would become unsound by considering an input problem containing the unit clause $f(x) \neq x$.)

Even if we require that in the sequence $\mathcal{P}$ a domain constant d_j does not occur as an argument of a principal term p_i unless $i > j$ (which solves the above complication), it is not generally sound to add clauses of the second kind (2) for non-constant principal terms. To see this, consider the sequence $\mathcal{P}$ with $p_1 = a, p_2 = f(d_1), p_3 = f(d_2)$ and assume that after the straightforward $\sigma^{-1}(d_1) = M(a)$, we learn that $M(f)(\sigma^{-1}(d_1)) = \sigma^{-1}(d_1)$ and thus we do not need to "use" a new domain constant to process p_2. However, similarly to the previous example, we are now forced to define $\sigma^{-1}(d_2)$ before we can proceed to p_3. Moreover, it is easy to imagine a model M in which any choice of such next element results in $M(f)(\sigma^{-1}(d_2)) \notin \{\sigma^{-1}(d_1), \sigma^{-1}(d_2)\}$ and we are forced to define $\sigma^{-1}(d_3) = M(f)(\sigma^{-1}(d_2))$. Thus the new model M_σ will satisfy $f(d_2) = d_3$, but also $f(d_1) \neq d_2$ and $a \neq d_2$.

The following theorem reflects these observations and formalises and further generalises the results reported in [4].

Theorem 5. *Let $\mathcal{P} = p_1, \ldots, p_m$ be a non-empty sequence of principal terms such that whenever a domain constant d_j occurs as an argument of a principal term p_i then $j < i$.*[9] *Moreover, let the domain constants appear in $\mathcal{P}$ "in order", i.e. if d_j for $j > 0$ occurs in p_i then there is $i' \leq i$ such that d_{j-1} occurs in $p_{i'}$. Let $\mathcal{D}_\mathcal{P}$ consist of all the clauses of the first kind* (1) *and of the clauses of the second kind* (2) *for any $1 < i \leq m$ and $2 < j \leq i$ such that d_{j-1} does not occur in any $p_{i'}$ for $1 \leq i' \leq i$. Then $D_\mathcal{P}$ is a symmetry avoiding constraint.*

Proof. Let us immediately focus on the sole non-trivial point of Definition 1, namely point (ii). As in the proof of Theorem 4 we recursively construct a permutation σ used for relabelling the elements of a given model M such that M_σ additionally satisfies $\mathcal{D}_\mathcal{P}$. And as before, we describe the construction of σ^{-1}. Let us by σ_i^{-1} denote the partial permutation obtained after processing the sequence $\mathcal{P}$ up to element p_i and let us initiate the construction with σ_0^{-1} as the empty mapping.

We now consider the i-th step of the construction for some $1 \leq i \leq m$ assuming σ_{i-1}^{-1} is already defined. First, if there is a domain constant d which occurs in p_i that is not in the domain of σ_{i-1}^{-1}, we pick an arbitrary domain constant e not in the range of σ_{i-1}^{-1} and set $\sigma_i'^{-1} = \sigma_{i-1}^{-1} \cup \{\mathsf{d} \mapsto \mathsf{e}\}$. If this happens, we say that d enters the domain of σ^{-1} to *define an argument* of p_i. We may need to repeat this several times until we obtain τ_i^{-1}, an extension of σ_{i-1}^{-1}, whose domain contains all the domain constants occurring in p_i. Let $p_i = f(\mathsf{d}_1, \ldots, \mathsf{d}_a)$ and let $\mathsf{e} = M(f)(\tau_i^{-1}(\mathsf{d}_1), \ldots, \tau_i^{-1}(\mathsf{d}_a))$. If e is in the range of τ_i^{-1} we set $\sigma_i^{-1} = \tau_i^{-1}$. Otherwise, let d be the least domain constant not in the domain of τ_i^{-1} and we set $\sigma_i^{-1} = \tau_i^{-1} \cup \{\mathsf{d} \mapsto \mathsf{e}\}$. In this case we say that d enters the domain of σ^{-1} to *stand for the value* of p_i. As in the proof of Theorem 4, we obtain the final σ^{-1} from σ_m^{-1} by "pairing up" the remaining domain constants "not yet" in the domain of σ_m^{-1} with the remaining domain constants "not yet" in its range arbitrarily. These domain constants are said to enter the domain of σ^{-1} to *finish it up*.

Let us now verify that M_σ satisfies $\mathcal{D}_\mathcal{P}$. We first look at clauses of the first kind (1). These are satisfied, because our construction maintains that the domain of σ_i^{-1},

[9] In particular, p_1 must be a constant.

which contains $M_\sigma(p_i)$, is always a subset of $\{d_1, \ldots, d_i\}$. To see this, we proceed by induction. First, the domain σ_0^{-1} is the empty set. Next, assuming that the domain of σ_{i-1}^{-1} is a subset of $\{d_1, \ldots, d_{i-1}\}$ (the induction hypothesis), we check that the domain of τ_i^{-1} is always a subset of $\{d_1, \ldots, d_{i-1}\}$ using the assumption that whenever a domain constant d_j occurs as an argument of a principal term p_i then $j < i$. To finish, we recall that the construction only possibly adds one more element when extending τ_i^{-1} to σ_i^{-1} and this is always the least domain constant "not yet" in the domain of τ_i^{-1}.

Finally, we look at the clauses of the second kind (2). Let $1 < i \leq m$ and $2 < j \leq i$ and let

$$C \;=\; (p_i \neq d_j \vee p_1 = d_{j-1} \vee \ldots \vee p_{i-1} = d_{j-1})$$

be one such clause. Let us assume that C is false in M_σ. Because $M_\sigma(p_i) = d_j$, neither the domain constant d_j nor d_{j-1} did enter the domain of σ^{-1} to finish it up. Moreover, since $M_\sigma(p_{i'}) \neq d_{j-1}$ for $1 \leq i' < i$ the domain constant d_{j-1} did not enter the domain of σ^{-1} to stand for the value for any of these $p_{i'}$. Thus d_{j-1} must have entered the domain of σ^{-1} to define an argument of some $p_{i'}$ for $1 \leq i' \leq i$. But then d_{j-1} occurs in some $p_{i'}$ for $1 \leq i' \leq i$ and C thus cannot be part of $\mathcal{D}_\mathcal{P}$. □

4 Symmetry Avoidance Heuristics

The previous section characterised the notion of a symmetry breaking constraint determined by a list of principal terms $\mathcal{P}$. In this section we propose a number of heuristics for selecting a good $\mathcal{P}$. The underlying idea is that as we can only add n clauses of the 'first kind' (1) we want to pick the 'best' n principal terms, i.e. those that avoid most symmetries. The best set $\mathcal{P}$ is such that S together with C_P ensures that each element of $\mathcal{P}$ must be interpreted by a distinct domain constant. However, checking this is impractical and therefore we introduce heuristics for this.

To ensure completeness, we optionally enforce the constraints set out in Theorems 4 and 5 from the previous section by limiting the principal terms added to P where they would otherwise break these constraints. Note that the *diagonal* approach below naturally preserves these constraints in all cases and in most cases it is not necessary to restrict P. We preserve the option to run in an incomplete mode where it is no longer possible to report that a model cannot be found.

Ordering function symbols. The first heuristic considers how function symbols should be ordered. Consider the problem $S = \{a = b, a = c, a \neq d\}$, selecting $p_1 = a, p_2 = b$ will not be as effective as selecting $p_1 = a, p_2 = d$. In the first case, the equality $a = b$ induces a stronger constraint than the ordering. In the second case, the ordering constraint is stronger than that induced by the inequality $a \neq d$. We consider the following variations:

- *Occurrence.* By default, function symbols are ordered by their order of appearance in the input problem. This may perform poorly if similar functions (those whose interpretations overlap significantly e.g. principal terms are interpreted as the same domain constants) are defined close together in the input file; conversely it may perform well if differing function symbols are defined close together.

- *Input Usage Frequency.* This orders symbols by their frequency in the input.
- *Preprocessed Usage Frequency.* This orders symbols by their frequency in the pre-processed clauses (pre-processing may copy some symbols many times).
- *Arity.* This orders symbols from the smallest to largest arity. The reasoning here is that it is simpler to show that functions with lower arity are distinct.

The hypothesis around using frequency is that the most used symbols are likely to be distinct. In case the opposite is true, in both frequency cases we also add their reverse. We also consider a *randomised* order.

Ordering the construction of principal terms. We consider how complex principal terms are ordered. One approach is to put all principal terms for one function before those for the other. But if the problem contains, e.g. $f(x) = a$, all principal f- terms already have the same interpretation and cannot be strictly ordered. Conversely, we may wish to order by argument value (all those with d_1 before those with d_2). But if the problem contains, e.g. $f(x) = g(x)$ then again the interpretation of the principal f-terms must agree with the succeeeding g-term in the sequence $f(d_1), g(d_1), f(d_2), g(d_2), \ldots$ such that their ordering constraint becomes ineffective. Based on these observations we consider the following variations which make use of an ordering $<_f$ on function symbols and the ordering $<_{DC}$ on domain constants.

- *Function First.* Orders principal term by $<_f$ and then $<_{DC}$
- *Argument First.* Orders principal terms by $<_{DC}$ and then $<_f$
- *Diagonal.* Orders principal terms for each function by $<_{DC}$ and then for each function symbol in turn (according to $<_f$) selects the next principal term *starting* with the ith term for the ith function e.g. we may have $f(d_1), g(d_2), h(d_3), f(d_2), \ldots$.

We also consider a *randomised* order.

Restricting Symmetry Avoidance Clauses. This heuristic does not consider the order of $\mathcal{P}$ but the clauses we add for $\mathcal{P}$. Given principal terms $\mathcal{P}$ and a target model size n, we add n clauses of the first kind and $|\mathcal{P}| \times n$ clauses of the second kind. The large number of these second kind of clauses may become too expensive for the SAT solver. Therefore, by default we restrict $\mathcal{P}$ to have at most n elements and we can optionally add a multiplier k (such that $k \leq |\mathcal{P}| \times n$) to this.

5 Experimental Evaluation

In this section we experimentally address a number of research questions, evaluating the effectiveness of the techniques introduced earlier. Vampire relies on a schedule of strategies for attacking a problem and our evaluation reflects our desire to identify options of complementary strengths, as discussed elsewhere [13].

Experimental Setup. We considered problems from the TPTP [18] library (version 7.0) in the FOF or CNF format that were either (counter-)satisfiable or belong to the effectively propositional (Bernays–Schönfinkel) fragment (as this process is complete for

-fmbsso	-fmbswo	-fmbse
occurrence	**function_first**	**0** = $\mathcal{P}$ as defined
input_usage	argument_first	1 = empty $\mathcal{P}$
preprocessed_usage	diagonal	2 = $\mathcal{P}$ restricted to constants
random	random	
reverse_input_usage		
reverse_preprocessed_usage		
arity		

Fig. 1. Option values for symmetry avoiding strategies (defaults in bold).

this fragment). We removed all problems known to only have infinite models (determined by Infinox [3]). This led to a set of 2790 problems of which 1512 are known to be satisfiable, 969 are known to be unsatisfiable and 23 are open problems.

The techniques described in the previous sections were implemented in the Vampire theorem prover.[10] The version of Vampire used in these experiments can be found online.[11] Experiments were run on the StarExec cluster [17], whose nodes are equipped with Intel Xeon 2.4GHz processors and 128 GB of memory. For each experiment we will report the number of problems solved with the time limit of 600 seconds.

The options related to symmetry avoidance covered were the order of symbols (-fmbsso) and the enumeration strategies between functions applied to domain constants only (-fmbswo). Further, we added options to turn off symmetry avoidance altogether (-fmbse 1) and to order only constants (-fmbse 2). We also limited vampire's proof search strategy to MACE style finite model finding (-sa fmb). Figure 1 summarises the options and their values (which correspond directly to those described in Section 4).

Summary. We ran 30 experiments with sensible[12] combinations of the above options. Across all experiments we solved 1901 out of 2790 problems. Out of these 1150 were shown to be satisfiable and 734 were shown to be unsatisfiable. The mean solution time for satisfiable problems was 8.3 seconds and for unsatisfiable problems it was 9.2 seconds. Table 1 provides some general statistics. On the left we see the best, mean, and worst solving times for problems. This means that the majority of problems are solved quickly by some strategy. But, only 58 problems were solved by all experiments. There were 264 problems that took longer than 10 seconds to solve where the difference between best and worst experiment was at least 5 seconds. These are interesting problems as they demonstrate real differences in solution times. Within this set, there is considerable variation between the best and worst solving times. Figure 2 illustrates the distribution of the *speedup* between best and worst strategy on this set. Very large speedups are seen where problems are solved in minutes by one strategy and seconds by another.

[10] https://vprover.github.io/

[11] https://derivation.org/frocos2019

[12] Some combinations are not sensible. For example, randomising the ordering of principal terms means that any ordering of function symbols will be ignored.

Table 1. General statistics about problems solved.

	# problems solved in X time				
	$< 10s$	$< 30s$	$< 1m$	$< 5m$	Total
Best	1715	1797	1828	1888	1901
Mean	1673	1773	1817	1885	1901
Worst	1593	1686	1753	1859	1901

	# problems solved in X experiments					
	All	< 25	< 20	< 10	< 5	1
Satisfiable	46	1119	21	15	15	10
Unsatisfiable	12	724	63	2	2	0

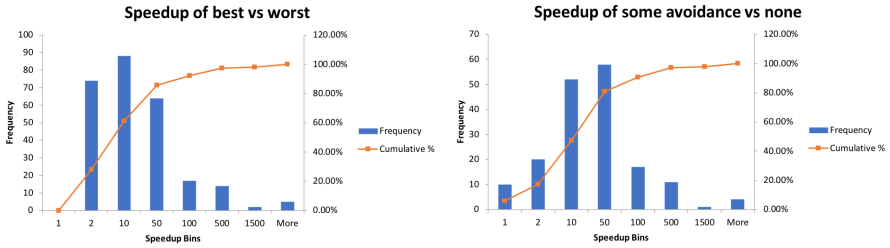

Fig. 2. Histograms of speedups comparing best and worst strategies and no avoidance with best strategy for our 264 interesting problems.

Which Ordering Heuristics Perform Best? Table 2 presents the results for comparing the different ordering heuristics introduced earlier. Since testing all combinations of options would lead to 84 constellations, we always vary one option and leave all others at their respective defaults. In each case we record how many problems that option was the best (fastest) for and what the mean speedup (over the second best) was in the case where the option was the best. Note that each line includes all strategies where that option was selected i.e. more than one experiment. Here we focus explicitly on problems taking $> 10s$ as these are the ones that are, in principle, the harder problems.

Here we can see that the performance of different values is varied. Note that the speedup value addresses the question of how much we gain by adding a single strategy on top of the rest. However, this hides particular outlier cases. For example, the problem `HWV052-1.007.004` was solved in $13s$ with the diagonal approach, the other principal term orders take at least $535s$, a speedup of a factor of almost 38. Conversely, for `NLP077-1.p` the diagonal approach took 10% longer.

Of the function ordering options, the reverse frequency options fared worse than the standard frequency options, which supports a hypothesis that it is better to avoid symmetries on common symbols. Interestingly, randomising the order was hardly ever the best approach, suggesting that there is a benefit from our heuristic orderings. We are surprised that the arity option did not fare well. However, this may be attributable to the fact that the majority of functions in problems are typically of low arity anyway.

Of the principal term ordering options, the best approach was the function-first approach. This suggests that problems typically contain functions which are distinct in their arguments. It is interesting to note that the randomisation approach here fared very well. This suggests that there are orderings that perform well outside of our heuristics

Table 2. Comparing the different options for ordering heuristics for problems $> 10s$.

Value	Best	Mean Speedup
occurrence	61	1.04
input_usage	49	1.89
reverse_input_usage	11	1.16
preprocess_usage	44	1.04
reverse_preprocess_usage	13	1.03
arity	12	1.04
random	2	1.01

Value	Best	Mean Speedup
function_first	72	1.06
argument_first	27	1.03
diagonal	36	2.22
random	57	1.03

and we should inspect what elements of these random orders were beneficial and attempt to encode them in new heuristics.

Table 3. Comparing the different values for limiting symmetry avoiding clauses.

Value	Solved	Best	Mean Speedup
1	1884	67	5.12
5	1882	127	4.03
10	1883	130	7.71
100	1886	88	4.23
1000	1886	131	9.37

What is the Effect of Limiting Symmetry Avoidance Clauses? Table 3 compares the results of limiting the size of $\mathcal{P}$ as some multiple of n. Here we can see that the number of solved problems increases monotonically. However, the amount of time taken to find solutions varies and in some cases restricting to n provides the best (fastest) solution, whereas including more and more values in $\mathcal{P}$ can help in other situations. It is interesting to note that for the largest multiplier we see the biggest speedup. This suggests that where a large multiplier can be of use it will make a large difference. We will keep this option and the various values for strategy building.

Does Symmetry Avoidance Always Help? Next we question whether adding symmetry avoidance constraints is always helpful. Overall, there were 96 problems where the fastest solution was to not add symmetry avoiding constraints. On average the next fastest solution was 24% slower. The majority of these were short runs (under 10s), but in some cases the difference was significant. For example, problem `ALG333-1.p` was solved in $32s$ without symmetry avoiding constraints, but the next best solution solved it in $54s$.

Furthermore, there were many problems only solved using symmetry avoidance. Without symmetry avoidance we only solve 173 of our 264 interesting problems (with 91 unsolved). On these problems, the resulting speedups are given in Figure 2. Again, we see that symmetry avoidance brings large performance gains. Although there are 12 problems where solving without symmetry avoidance is the best (fastest) strategy.

Our final question is whether restricting symmetry avoiding constraints to constants only has any benefit, or conversely whether there are cases where we need to avoid symmetries on non-constant terms. There were 55 problems where it was better (i.e. the solution was faster) to exclude non-function symbols from symmetry avoiding. This means that ordering principal terms is an interesting research question.

How does this compare to Paradox? Finally, we compare our results to Paradox. Overall, Paradox solves 48 problems that we do not solve and we solve 54 problems unsolved

by Paradox. All 54 of these problems rely on symmetry avoidance options. Of the interesting problems, roughly half (145) are solved more quickly by Paradox and the rest are solved more quickly by Vampire, out of these 36 problems are solved at least 10 times faster with Vampire.

Discussion. The above experimental results show that the issue of symmetry avoiding is important and that a portfolio solver such as Vampire needs many options available to it. These experiments have allowed us to *prioritise* options within our portfolio and suggest further exploration is required. In particular, we need to explore the correlation between the success of symmetry avoiding heuristics and the structure of properties, especially the number and distribution of function symbols with different arities.

6 Comparing Symmetry Breaking and Symmetry Avoidance

So far we have focussed on avoiding introducing new symmetries into the SAT problem. There also exist tools for identifying symmetries in SAT problems. In the final part of this paper we utilise one such tool to answer the following two questions:

1. Could incorporating static symmetry breaking improve the performance of finite model finding?
2. Are symmetry avoidance and symmetry breaking complementary (i.e. do the avoidance constraints help symmetry breaking) or is avoidance subsumed by breaking?

The Symmetry Breaking Problem. The symmetry breaking problem is similar to the symmetry avoiding one, but in a more general setting. Slightly informally, given a SAT problem S is it possible to produce some constraints $\mathcal{C}$ such that the models of S and $S \cup \mathcal{C}$ are the same up to isomorphism, but there are fewer models of $S \cup \mathcal{C}$. The problem of symmetries has been studied extensively in the constraint programming, the ASP and the SAT communities [8, 7, 16]. The main differentiation of the techniques lies between dynamic [20, 21, 2] and static [1, 5] symmetry breaking. The dynamic setting aims to identify and break symmetries during the solving process whilst the static setting updates the problem directly. In this work we focus on one of the best existing static symmetry breaking tools BreakID [5].

Experimental Setup. We select the same 2970 problems as in Section 5 and run finite-model finding on each problem for 60 seconds, recording the SAT problems produced for each model size in the DIMACS format [9]. Note that for each problem where we explore up to model size n we will produce at least $n - 1$ unsatisfiable SAT problems. Therefore, the majority of the SAT problems will be unsatisfiable.

We then run the BreakID static symmetry breaker [5] on each SAT problem for 60 seconds. BreakID produces a copy of the problem with additional constraints added that break identified symmetries in the problem. This will cover both symmetries in the original problem and any symmetries introduced via our encoding. Finally, we run Minisat [6] on each SAT problem (this is the SAT solver used by Vampire internally).

We repeat the above experiment for different heuristics. To establish a baseline, we start without symmetry avoidance and investigate symbol orders by occurrence and

Table 4. Comparing solving with and without breaking. T/O means timeout and BiD means BreakID.

Without Breaking			With Breaking				Gained		Lost		Loss / Gain
Sat	Unsat	T/O	Sat	Unsat	T/O (BiD)	T/O (Sat)	Sat	Unsat	Sat	Unsat	
1,194	12,919	423	954	11,991	1,435	156	3	191	262	1,171	7.39

Table 5. Solving statistics by SAT problem. T/O means timeout and BiD means BreakID.

Options	Total	Without BreakID			With BreakID				Gained		Lost		Loss/
		Sat	Unsat	T/O	Sat	Unsat	T/O (BiD)	T/O (Sat)	Sat	Unsat	Sat	Unsat	Gain
prepro, ff	13,791	1,289	12,242	260	1,067	11,441	1,062	221	4	15	230	831	55.84
occ, ff	13,788	1,272	12,254	262	1,062	11,421	1,063	221	2	15	224	861	63.82

preprocessed_usage that fared well before (see Table 2). In both cases we construct terms by functions first. The system used for the experiments is an Intel Xeon E5520 with 2.27GHz and 16GB memory.

6.1 The Effect of Symmetry Breaking

First we look at the effect that static symmetry breaking can have on the finite-model finding process independently of our symmetry avoidance heuristics. Running finite-model finding using the default strategy (without symmetry avoidance) produces 14,536 SAT problems. Table 4 reports the difference between running with and without static symmetry breaking. Overall, more problems are solved without symmetry breaking than with. However, this is mainly due to the timing out of the static symmetry breaking process. There are 194 SAT problems that are solved with static symmetry breaking that were not solved without it. This represents an opportunity for making further progress in the finite model finding process. As expected, this has a far greater effect on the unsatisfiable problems, which will partly be due to the fact that over 80% of problems are unsatisfiable and partly due to the fact that these are fundamentally harder.

In this we spent 60 seconds on static symmetry breaking and 60 seconds on SAT solving. The next question to ask is whether the time spent on static symmetry breaking can be justified. In 1,811 experiments the time spent on breaking and solving combined is roughly equivalent to that of solving by itself without breaking. In 1,062 problems the solution was faster without breaking, leaving 56 problems where the combination of breaking and solving performed faster than solving without breaking.

This experiment shows that whilst static symmetry breaking can help on a small number of problems, in general it reduces performance.

6.2 Comparing Breaking and Avoidance

Next we want to see what happens when we combine the symmetry avoiding heuristics with static symmetry breaking. To do this we run the two best symmetry avoiding strategies from the previous section and repeat the above experiment.

Table 5 reports only 13,791 and 13,788 generated files. This is due to the time spent in symmetry avoidance. Compared to the baseline, there are fewer time-outs and solved unsatisfiable problems, but more satisfiable ones. The rate of time-outs (1,062 and 1,063) during symmetry breaking is also similar, which leads to a high number of lost solutions. However, the number of solutions gained by symmetry breaking over avoidance is significantly lower (19 and 17). This suggests that symmetry avoidance was already having a significant impact on solving times.

It is possible that the distinction between solutions gained and lost is too rough. Next we investigate the speed-ups in timing between problems of unsatisfiable solutions for pairs of symmetry-avoidance and symmetry-breaking options. We also restrict the problems to those where the model size is larger than the number of constants. In these cases, not all domain constants can be assigned to input constants which leaves room for the different symmetry orders with regard to functions. Table 6 shows the number of problems that were solved faster and slower. The time for BreakID includes the time taken for static symmetry breaking. Most timings were sufficiently close that jitter effects could tip the balance either way. For this reason all results within 2 seconds were excluded.

When applying BreakId, about 10% of the SAT problems cannot be processed within the 60 seconds time limit of the full input problem. This leads to a high ratio of problems lost due to symmetry breaking against the new solutions gained. There is also a consistent disparity in the gain/loss ratio between satisfiable and unsatisfiable problems. Two factors could contribute to this phenomenon. First, the separation into symmetry breaking and SAT solving comes with a significant overhead in parsing and duplication of data structures. Moreover, BreakId itself depends on the automorphism library saucy[**?**,**?**] which leads to another duplication of data-structures. Second, we need to take the whole sequence of models generated into account. When BreakId times out already for small model sizes the larger model sizes are likely to follow. This artificially amplifies the number of lost unsatisfiable solutions. On the other hand, the satisfiable solutions depend more strongly on the heuristics of the SAT solver which leads to less predictable timings.

As a consequence, we compare the gains and losses between the baseline and the two symmetry orders as well. The baseline loses about 7 times as many problems as gained by symmetry breaking. Both the preprocessed and occurrence symmetry order retain a similar number of lost problems. Also the number of satisfiable problems gained is similar to the baseline. The main improvement of symmetry avoidance lies with the unsatisfiable lost problems where more than 90% of the problems gained versus the baseline can be recovered by the heuristics.

Most results are indistinguishable. Both symmetry avoidance options tend to speed solving up more than slowing it down when compared to the baseline, but they themselves are indistinguishable. Even without time-outs, symmetry breaking tends to be slower than symmetry avoidance. Combining symmetry breaking and symmetry avoidance mostly improves the solving times. Again there is no distinguishable difference between the two avoidance options.

Table 6. Pairwise comparison of SAT problems with model size > number of constants

A	B	A slower than B	A faster than B	Too Close
baseline	preprocessed, ff	147	16	1,063
baseline	occ, ff	145	16	1,064
preprocessed, ff	occ, ff	0	0	1,640
baseline	baseline+BreakId	56	1,062	1,811
preprocessed, ff	preprocessed, ff+BreakId	13	408	1,139
occ, ff	preprocessed, ff+BreakId	13	405	1,139
baseline+BreakId	preprocessed, ff+BreakId	279	7	877
baseline+BreakId	occ, ff+BreakId	276	7	870
preprocessed, ff+BreakId	occ, ff+BreakId	0	0	1550

6.3 Discussion

We summarise answers to our two initial research questions. Symmetry breaking can help solve more problems, but in the majority of cases, the cost of static symmetry breaking is higher than symmetry avoidance. When considered alongside symmetry avoidance, the benefits of symmetry breaking are more modest, suggesting that overall the effort of incorporating these techniques directly into the finite model finding process may not be worthwhile.

7 Conclusion and Future Work

In this paper, we have characterised the symmetry avoidance problem for MACE-style finite model finding, suggested a number of sound heuristics for symmetry avoidance, and experimentally evaluated these heuristics. We found that some of these variations can significantly speed up the finite model finding process. Finally, we looked at whether directly identifying and breaking symmetries in the SAT problems would give any further improvements. In further work we would like to explore further heuristics and the correlation between the ordering heuristics and the signature of a problem.

Acknowledgement. We thank the anonymous reviewers for critically reading the paper and suggesting substantial improvements.

References

1. F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pp. 836–839, 2003.
2. G. Audemard and L. Henocque. The extended least number heuristic. In *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, pp. 427–442, 2001.
3. K. Claessen and A. Lilliesträm. Automated inference of finite unsatisfiability. *J. Autom. Reasoning*, 47(2):111–132, 2011.

4. K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *CADE-19 Workshop: Model Computation - Principles, Algorithms and Applications*, 2003.
5. P. T. Darga, H. Katebi, M. Liffiton, I. L. Markov, and K. Sakallah. Saucy Homepage. http://vlsicad.eecs.umich.edu/BK/SAUCY.
6. P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pp. 530–534. ACM, 2004.
7. J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. Improved static symmetry breaking for SAT. In *SAT 20127*, pp. 104–122, 2016.
8. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pp. 502–518, 2003.
9. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
10. I. P. Gent, K. E. Petrie, and J. Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, pp. 329–376. 2006.
11. S. C. Homepage. SAT Competition 2009: Benchmark Submission Guidelines. https://www.satcompetition.org/2009/format-benchmarks2009.html.
12. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35, 2013.
13. W. Mccune. A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical report, Argonne National Laboratory,, 1994.
14. A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning (in 2 volumes)*, pp. 335–367. 2001.
15. G. Reger, M. Suda, and A. Voronkov. The challenges of evaluating a new feature in vampire. In *Proceedings of the 1st and 2nd Vampire Workshops*, vol. 38 of *EPiC Series in Computing*, pp. 70–74. EasyChair, 2016.
16. G. Reger, M. Suda, and A. Voronkov. Finding finite models in multi-sorted first-order logic. In *Theory and Applications of Satisfiability Testing – SAT 2016*, pp. 323–341. Springer International Publishing, 2016.
17. G. Reger, M. Suda, and A. Voronkov. New techniques in clausal form generation. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, vol. 41 of *EPiC Series in Computing*, pp. 11–23. EasyChair, 2016.
18. K. A. Sakallah. Symmetry and satisfiability. In *Handbook of Satisfiability*, pp. 289–338. 2009.
19. A. Stump, G. Sutcliffe, and C. Tinelli. StarExec, a cross community logic solving service. https://www.starexec.org, 2012.
20. G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
21. A. Vakili and N. A. Day. Finite model finding using the logic of equality with uninterpreted functions. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, vol. 9995 of *Lecture Notes in Computer Science*, pp. 677–693, 2016.
22. J. Zhang and H. Zhang. SEM: a system for enumerating models. In *IJCAI 95*, pp. 298–303, 1995.
23. J. Zhang and H. Zhang. System description: Generating models by SEM. In *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, pp. 308–312, 1996.