New Techniques for Static Symmetry Breaking in Many-Sorted Finite Model Finding

Joseph Poremba, Nancy A. Day, *Member, IEEE Computer Society,* and Amirhossein Vakili

Abstract—Symmetry in finite model finding problems of many-sorted first-order logic (MSFOL) can be exploited to reduce the number of interpretations considered during search, thereby improving solver performance for tools such as the Alloy Analyzer. We present a framework to soundly compose static symmetry breaking schemes for many-sorted finite model finding. Then, we introduce and prove the correctness of three static symmetry breaking schemes for MSFOL: 1) one for functions with distinct sorts in the domain and range; 2) one for functions where the range sort appears in the domain; and 3) one for predicates. We provide a novel presentation of sort inference in the context of symmetry breaking that yields a new mathematical link between sorts and symmetries. We empirically investigate how our symmetry breaking approaches affect solving performance.

Index Terms—Finite Model Finding, Symmetry Breaking

1 Introduction

Automated verification engines play key roles in many software engineering problems. Many of these problems can be represented in first-order logic (FOL) to solve. For example, the Alloy Analyzer [1] uses finite model finding (FMF) of (essentially) first-order logic[1] to find counterexamples to properties and to return satisfying instances for applications such as role-based access control [2], finding bugs in the CHORD protocol [3], understanding UML diagrams [4], checking health care workflows [5], finding errors in verification algorithms [6], reasoning about operations on sparse matrices for scientific computation [7], checking networking architectures [8], and temporal logic model checking [9] [10].

The satisfiability problem of finding a solution to a formula in FOL is undecidable. However, in finite model finding we restrict the problem to searching for solutions of a bounded size, which is decidable. If FMF discovers a solution, the formula is satisfiable. However, a formula that is unsatisfiable for FMF may be satisfiable for the unbounded satisfiability problem, due to the existence of a solution that has a larger size than the search bound (even a potentially infinite size). According to Jackson's small scope hypothesis [1], most flaws in formal models can be exposed by counterexamples of small size, so if all possible interpretations up to a sufficiently high size are explored and no counterexample is found, then we can have a high degree of confidence in the correctness of the model. Solvers over unbounded domains such as CVC4 [11] have added FMF capability to provide feedback to users [12] [13]. FMF affords a good balance between providing useful results while slightly sacrificing the generality of the result. But, it is clear that looking at larger interpretations adds confidence to the result of FMF, thus, performance improvements in these automated verification engines are key to bringing the value of formal verification to many software engineering problems.

There are two main styles of FMF: the MACE-style [14], which reduces the problem to symbolic logic and uses a logic solver; and the SEM-style [15], which has a backtracking algorithm for searching for an interpretation explicitly. Paradox [16] and Kodkod [17] are both MACE-style solvers that reduce the FMF problem to boolean satisfiability and use SAT solvers. Kodkod is used in the Alloy Analyzer. Fortress [18] is a MACE-style solver, but it reduces the FMF problem to quantifier-free first-order logic and uses an SMT solver (forcing it to search only a finite set of interpretations). Mace4 [19] is a SEM solver (unlike its predecessor Mace2, which is in the MACE-style). The SMT solver CVC4 has a hybrid algorithm where a SEM-style search for finite model finding is integrated into the logic solver [12].

We can improve the performance of finite model finders by exploiting symmetries in the set of possible interpretations. Symmetry breaking means ruling out symmetric (isomorphic) interpretations from consideration as solutions and thereby reducing the number of interpretations that must be examined to conclude a problem is satisfiable or unsatisfiable. These reductions can be done dynamically during search in a SEM-style finder or statically in a MACE-style finder. In the static approach, formulas are added to the problem that disallow interpretations from being considered as solutions either at the level of the propositional logic encoding (e.g. [20]) or at the FMF-problem level. Wang et al [21] show the effectiveness of existing problem-level static symmetry breaking (rather than symmetry breaking at the propositional logic level) on reducing the time to find a satisfying solution and reducing the number of satisfying solutions produced in the Alloy Analyzer. In this paper we study the problem of *static symmetry breaking* at the *FMF-problem* level.

- *Joseph Poremba (jporemba@cs.ubc.ca) is with the Department of Computer Science, University of British Columbia.*
- *Nancy A. Day (nday@uwaterloo.ca) and Amirhossein Vakili (avakili@uwaterloo.ca) are with the David R. Cheriton School of Computer Science, University of Waterloo.*

Manuscript received May 3, 2021; revised Nov 16, 2022.

1. Alloy is relational logic with transitive closure, but transitive closure over finite domains can be expanded to stay within first-order logic.

Claessen and Sörenson's symmetry breaking in Paradox [16] is focused on *unsorted* (or single-sorted) FMF problems, so all elements are drawn from a single domain. Their symmetry breaking techniques are used in Kodkod [22] for unsorted relational logic. However, in software engineering, types are often valued in problem descriptions and several tools for theorem proving and model finding, such as SMT solvers [23], Fortress [18], SEM [24], and Reger et al.'s extensions to Vampire [25] use a *many-sorted first-order logic* (MSFOL). Such systems are analogous to type systems in programming languages. In unsorted logic, the universe is homogoneous. In many-sorted logic, the universe is heterogenous and partitioned into multiple sorts. The expressive power of many-sorted and unsorted FOL are equivalent since sorts can be translated into predicates, but many problems have a more natural representation in a many-sorted logic, and the use of sorts can increase performance [25] [16].

Claessen and Sörenson [16] describe basic ways to generalize their symmetry breaking techniques to many-sorted finite model finding (MSFMF). These generalizations are used in Fortress [18] and Reger et al.'s extensions to Vampire [25]. However, these generalizations are limited in the symmetry breaking they can provide for many-sorted problems. They only consider sorts in isolation and so are only applicable for constants and single-sorted functions that use only one sort for both the inputs and output. To strengthen symmetry breaking in the many-sorted setting, we want to be able to apply symmetry breaking to functions that arbitrarily mix sorts for their input and output. The complications of having a sort system mean that we must be careful about how to soundly combine symmetry breaking techniques.

The main contributions of this article are:

- We introduce a general framework of iterative symmetry breaking over *partially-interpreted* MSFMF problems (meaning problems that already include symmetry breaking formulas). Our framework gives a general mechanism to soundly compose symmetry breaking schemes for different many-sorted functions and relations that may share sorts with each other. Our framework facilitates a simple implementation for the composition of different schemes.
- We present and prove correct three symmetry breaking schemes for MSFMF, using our general framework. Two of the schemes concern many-sorted functions. They generalize ideas from Claessen and Sörenson to allow multiple sorts and partially-interpreted problems, and provide more symmetry breaking for a class of functions that we call "range-domain independent" functions. The third scheme is entirely new and describes symmetry breaking over many-sorted relations. Together, these cover all the kinds of functions and predicates that can appear in MSFOL.
- We show how our schemes benefit from sort inference, and establish a new mathematical link between sort inference and symmetry.
- We empirically demonstrate the performance improvements possible with our new techniques over Claessen and Sörenson's techniques for MSFMF via an implementation of our new symmetry breaking techniques in the Fortress [18] finite model finder.

In Section 2, we provide background on MSFOL, MSFMF, static symmetry breaking, and the results of Claessen and Sörenson. In Section 3, we describe our framework for combining symmetry breaking schemes. In Sections 4, 5, 6, we describe our new symmetry breaking schemes and prove their correctness. Section 7 discusses the effect of sorts on symmetry breaking and Section 8 is the empirical analysis of the effect of our new symmetry breaking schemes on solver performance. Section 9 describes related work and Section 10 summarizes our contributions.

2 Background

In this section, we provide background on many-sorted first-order logic (MSFOL), many-sorted finite model finding (MSFMF), static symmetry breaking, and Claessen and Sörenson's work on static symmetry breaking. We assume a general familiarity with first-order logic (FOL).

2.1 Many-Sorted First-Order Logic (MSFOL)

The formulas and terms of many-sorted first-order logic (MSFOL) are constructed with respect to a collection of symbols called a signature.

Definition 1 (Signature)**.** A **signature** is a tuple $\Sigma = (\Theta, \mathcal{F}, \mathcal{R})$ where

- Θ is a finite set of symbols called **sorts**,
- $\mathcal{F}$ is a finite set of **functional symbols**.
- $\mathcal{R}$ is a finite set of **relational symbols** (or **predicate symbols**).

A functional symbol is a name f equipped with a sequence of argument (input) sorts $A_1, \ldots, A_n$ in Θ, and an output sort B in Θ. We write such a functional symbol as $f : A_1 \times \cdots \times A_n \to B$. A relational symbol is a name R equipped with a sequence of argument (input) sorts $A_1, \ldots, A_n$ in Θ. We write such a relational symbol as $R : A_1 \times \cdots \times A_n \to \mathrm{Bool}$ [2].

Constants are nullary function symbols (i.e. no argument sorts and only an output sort). We write a constant with name c and output sort A as $c : A$. For brevity, we often refer to functional and relational symbols as functions and relations respectively, though we emphasize that they are syntactic symbols and not actual functions or relations. We include the equality symbol $=: S \times S \to \mathrm{Bool}$ with its standard interpretation for each sort S without including it explicitly in the signature.

Σ-terms and Σ-formulas (usually called simply **terms** and **formulas**) have their usual definitions in first-order logic with the addition that each term has an associated sort, determined recursively by the sorts of subterms and the functional symbol used in constructing the term. For example, if $f : A \to B$ is a functional symbol, and t is a term with sort A, then $f(t)$ is a term of sort B. Sorts do not have any relationships (such as subtyping). Unless mentioned otherwise, all terms and formulas are closed (i.e. all variables are bound).

2. Bool is a fixed symbol not in Θ. We use Bool to indicate that the application of a relation to its arguments is a formula, not a term. Bool is not a sort considered for symmetry breaking.

In the semantics, sorts are interpreted by assigning sets to them.

Definition 2 (Domain Assignment)**.** For a given signature, a **domain assignment** $\mathcal{D}$ maps each sort S to a non-empty set called its **domain**, $\mathcal{D}(S)$. The elements of $\mathcal{D}(S)$ are called **values**. The size of a domain is called its **scope**.

Values are semantic, not syntactic, and thus do not appear in formulas. To emphasize this, we write values in a different font; for example $\mathcal{D}(S) = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. It is common to require that the domains of different sorts are disjoint, but it is not necessary.

Functional and relational symbols are interpreted with respect to a domain assignment $\mathcal{D}$. For a functional symbol $f : A_1 \times \cdots \times A_n \to B$, let $\text{Interps}(f)$ be the set of all functions from $\mathcal{D}(A_1) \times \cdots \times \mathcal{D}(A_n)$ to $\mathcal{D}(B)$. For a relational symbol $R : A_1 \times \cdots \times A_n \to \text{Bool}$, let $\text{Interps}(R)$ be the set of n-ary relations over the sets $\mathcal{D}(A_1), \ldots, \mathcal{D}(A_n)$.

Definition 3 (Interpretation)**.** For a given signature and domain assignment, an **interpretation** I assigns each functional or relational symbol f a function or relation, respectively, $f^I \in \text{Interps}(f)$.

Formulas are evaluated under interpretations using standard first-order logic evaluation rules. We say interpretation I **satisfies** formula ϕ if ϕ evaluates to True under I.

2.2 Many-Sorted Finite Model Finding (MSFMF)

In finite model finding, we are only concerned with **finite domain assignments**, in which the domain assigned to each sort is finite.

Definition 4 (Many-Sorted Finite Model Finding Problem)**.** A **many-sorted finite model finding (MSFMF) problem** P is a tuple $(\Sigma, \Gamma, \mathcal{D})$ where

- Σ is a signature,
- Γ is a finite set of formulas over Σ, and
- $\mathcal{D}$ is a finite domain assignment for Σ.

We let $\text{Interpretations}(P)$ denote the set of possible interpretations of an MSFMF problem P. We are concerned with the question of satisfiability for an MSFMF problem.

Definition 5 (Satisfiability)**.** Let $P = (\Sigma, \Gamma, \mathcal{D})$ be a MSFMF problem. An interpretation I of P **satisfies** Γ and is a **solution** of P if I satisfies every formula in Γ. If P has a solution it is **satisfiable**, otherwise it is **unsatisfiable**.

The goal of finite model finding is to determine whether a given MSFMF problem is satisfiable.

2.3 Static Symmetry Breaking

The literature on symmetry breaking for finite model finding ([26], [27], [16], [22], [28], [29], [25], [18]) focuses on a kind of value symmetry. Since values do not appear in formulas, it does not affect satisfiability whether $\mathcal{D}(A) = \{1, 2, 3, 4\}$ or $\mathcal{D}(A) = \{\mathsf{cat}, \mathsf{dog}, \mathsf{mouse}, \mathsf{bird}\}$. The values are merely labels, which can be changed by a bijection.

Definition 6 (Sorted-Value Bijection, Permutation)**.** Let $\mathcal{D}$ and $\mathcal{D}'$ be domain assignments over the same signature Σ. A **sorted-value bijection** (SV bijection) σ between $\mathcal{D}$ and $\mathcal{D}'$ is a collection of bijections, containing a bijection $\sigma_S : \mathcal{D}(S) \to \mathcal{D}'(S)$ for each sort S. If $\mathcal{D} = \mathcal{D}'$, σ is called a **sorted-value permutation** (SV permutation).

We apply an SV bijection σ between $\mathcal{D}$ and $\mathcal{D}'$ to relabel values inside interpretations. For a function $\mathcal{F} : \mathcal{D}(A_1) \times \cdots \times \mathcal{D}(A_n) \to \mathcal{D}(B)$ of an interpretation, we define $\mathcal{F}_\sigma : \mathcal{D}'(A_1) \times \cdots \times \mathcal{D}'(A_n) \to \mathcal{D}'(B)$ by

$$\mathcal{F}_\sigma(\mathsf{a}'_1, \ldots, \mathsf{a}'_n) = \sigma_B(\mathcal{F}(\sigma^{-1}_{A_1}(\mathsf{a}'_1), \ldots, \sigma^{-1}_{A_n}(\mathsf{a}'_n)))$$

That is, if $\mathcal{F}$ has the mapping $(\mathsf{a}_1, \ldots, \mathsf{a}_n) \mapsto \mathsf{b}$, then $\mathcal{F}_\sigma$ has the mapping $(\sigma_{A_1}(\mathsf{a}_1), \ldots, \sigma_{A_n}(\mathsf{a}_n)) \mapsto \sigma_B(\mathsf{b})$. For a relation $\mathcal{R} \subseteq \mathcal{D}(A_1) \times \cdots \times \mathcal{D}(A_n)$ of an interpretation, we define the relation $\mathcal{R}_\sigma \subseteq \mathcal{D}'(A_1) \times \cdots \times \mathcal{D}'(A_n)$ by

$$(\mathsf{a}'_1, \ldots, \mathsf{a}'_n) \in \mathcal{R}_\sigma \text{ iff } (\sigma^{-1}_{A_1}(\mathsf{a}'_1), \ldots, \sigma^{-1}_{A_n}(\mathsf{a}'_n)) \in \mathcal{R}$$

For an interpretation I over $\mathcal{D}$, we define an interpretation $\sigma \bullet I$ over $\mathcal{D}'$ by applying σ as above to each function and relation in the interpretation I. If σ is an SV permutation, we say I and $\sigma \bullet I$ are **isomorphic**.

It is well known (see for example the works by Claessen and Sörenson [16] and Peltier [27]) that this relabelling operation preserves whether an interpretation is a solution, thus we state the following theorem without proof.

Theorem 7 (SV Bijections Preserve Solutions)**.** *Let $\mathcal{D}$ and $\mathcal{D}'$ be domain assignments for the same signature Σ, and let σ be an SV bijection between $\mathcal{D}$ and $\mathcal{D}'$. For any interpretation I over $\mathcal{D}$ and formula ϕ, I satisfies ϕ if and only if $\sigma \bullet I$ satisfies ϕ.*

In symmetry breaking, we are primarily concerned with SV permutations and isomorphic interpretations. For two isomorphic interpretations, one is a solution if and only if the other is a solution. In general, an interpretation is isomorphic to numerous other interpretations (its **isomorphism class**). Since SV permutations preserve whether an interpretation is a solution, only one representative from each class need be checked to determine if the problem is satisfiable.

Symmetry breaking is a general name for any process that reduces the number of isomorphic interpretations that are checked when searching for a solution. SEM-style solvers perform **dynamic symmetry breaking**, where intelligent decisions are made during search to avoid isomorphic branches. Since MACE-style solvers do not perform a search directly, but reduce the problem to symbolic logic and invoke an external solver, they use **static symmetry breaking**, where extra formulas called **symmetry breaking formulas** are added to the problem that are satisfied by at least one member of an isomorphism class, but not all. The external solver can use these formulas to prune the search space, greatly improving their performance. Crawford et al [30] first introduced the concepts behind static symmetry breaking for SAT solving, but Claessen and Sörenson first brought it to the context of FMF. It is quite different in these two contexts: SAT relies on interchangeable variables rather than values. In this paper we focus on static symmetry breaking for FMF.

Once a symmetry breaking formula ϕ has been identified, it is added to the MSFMF problem P, generating a new problem P'. Since ϕ is satisfied by at least one member of each isomorphism class, the problems P and P' are

equisatisfiable. Thus a model finder can solve P' instead, which is often solved more quickly.

Symmetry breaking formulas blur the line between syntax and semantics. While values are not allowed to appear in traditional formulas (which makes values interchangeable in the first place), values are allowed to appear as terms in symmetry breaking formulas. In particular, if S is a sort, then any value $\mathsf{v} \in \mathcal{D}(S)$ may appear as a term of sort S. Semantically, such value-terms evaluate to themselves. Terms, formulas, and MSFMF problems that do not have values in terms are called **pure**. Terms, formulas, and MSFMF problems that have values in terms are called **partially interpreted**. We emphasize that partially-interpreted formulas are only introduced through symmetry breaking formulas; the formulas in the base problem are pure.

In addition to performance improvements from pruned search spaces, symmetry breaking provides benefits for counterexample generation. When one wishes to examine flaws in a formal model, viewing isomorphic counterexamples, which are identical modulo a renaming of values, may be undesirable: ideally, counterexamples which are structurally different should be presented. Symmetry breaking reduces the number of such redundant counterexamples.

2.4 Claessen and Sörenson's Symmetry Breaking

Claessen and Sörenson [16] introduced a few kinds of static symmetry breaking formulas for unsorted FMF problems. We can view these as MSFMF problems with a single sort A. Their first technique is applied to constants. Suppose sort A has constants $c_1 : A, \ldots, c_m : A$ and values $\mathsf{a}_1, \ldots, \mathsf{a}_n$. Intuitively, values are just interchangeable labels. For example, if there is a solution where c_1 is assigned value a_1, then there is a solution where c_1 is assigned a_2, and vice versa. So, for the purposes of checking satisfiability, we can assume that c_1 is assigned a_1. More generally, we assume the following for all $i \in \{1, \ldots, \max\{m, n\}\}$:

1) the i-th constant is assigned one of the first i values (constants do not have to have distinct values), and
2) if the i-th constant is assigned the j-th value, then an earlier constant must have been assigned the $(j-1)$-th value (for $2 \leq j \leq i$).

Item 1) is encoded by the following formulas (where $t \in \{t_1, \ldots, t_n\}$ is shorthand for $t = t_1 \vee \cdots \vee t = t_n$):

$$
\begin{aligned}
&c_1 = \mathsf{a}_1 \\
&c_2 \in \{\mathsf{a}_1, \mathsf{a}_2\} \\
&c_3 \in \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\} \\
&\ldots
\end{aligned}
$$

Item 2) is encoded by the following formulas:

$$
\begin{aligned}
&c_2 = \mathsf{a}_2 \implies \mathsf{a}_1 = c_1 \\
\\
&c_3 = \mathsf{a}_3 \implies \mathsf{a}_2 = c_2 \\
&c_3 = \mathsf{a}_2 \implies \mathsf{a}_1 \in \{c_1, c_2\} \\
\\
&c_4 = \mathsf{a}_4 \implies \mathsf{a}_3 = c_3 \\
&c_4 = \mathsf{a}_3 \implies \mathsf{a}_2 \in \{c_2, c_3\} \\
&c_4 = \mathsf{a}_2 \implies \mathsf{a}_1 \in \{c_1, c_2, c_3\} \\
&\ldots
\end{aligned}
$$

The conjunction of these formulas is a symmetry breaking formula.

Claessen and Sörenson's second technique is applied to functions. If, after the above formulas are added, there are still unused values remaining in the sort, say $\mathsf{a}_{k+1}, \ldots, \mathsf{a}_n$, then a function symbol $f : A \to A$ can be selected, and the following can also conjuncted with the above formulas:

$$
\begin{aligned}
&f(\mathsf{a}_1) \in \{\mathsf{a}_1, \ldots, \mathsf{a}_k, \mathsf{a}_{k+1}\} \\
&f(\mathsf{a}_2) \in \{\mathsf{a}_1, \ldots, \mathsf{a}_k, \mathsf{a}_{k+1}, \mathsf{a}_{k+2}\} \\
&f(\mathsf{a}_3) \in \{\mathsf{a}_1, \ldots, \mathsf{a}_k, \mathsf{a}_{k+1}, \mathsf{a}_{k+2}, \mathsf{a}_{k+3}\} \\
&\ldots
\end{aligned}
$$

A higher-arity single-sorted function $g : A \times \cdots \times A \to A$ could have been chosen instead, using $g(\mathsf{a}_i, \mathsf{a}_i, \ldots, \mathsf{a}_i)$ in place of $f(\mathsf{a}_i)$.

If $k = 0$ (i.e. no constants were constrained), then when adding constraints on f we cannot start with $f(\mathsf{a}_1) = \mathsf{a}_1, f(\mathsf{a}_2) \in \{\mathsf{a}_1, \mathsf{a}_2\}$, and so on. Intuitively, the reason why we can constrain $c_1 = \mathsf{a}_1$ is that all of the values are "symmetric choices" for c_1: it does not matter for the purposes of satisfiability which value c_1 is assigned. For an interpretation where c_1 is assigned a_1, there is another interpretation with the same structure, modulo a renaming of the values, where c_1 is assigned a_2. But for $f(\mathsf{a}_1)$, it changes the structure of the solution whether $f(\mathsf{a}_1)$ is assigned a_1 or some other a_i, since the former gives f a fixed point while the latter does not. However, a_2 is a symmetric choice to a_i for any $i \geq 2$, so we can use the following instead:

$$
\begin{aligned}
&f(\mathsf{a}_1) \in \{\mathsf{a}_1, \mathsf{a}_2\} \\
&f(\mathsf{a}_2) \in \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\} \\
&f(\mathsf{a}_3) \in \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4\} \\
&\ldots
\end{aligned}
$$

We call these symmetry breaking techniques the **Claessen and Sörenson constant (CSC) and function (CSF) schemes**, respectively. We refer to these as symmetry breaking schemes or techniques, since the actual formulas generated depend on the chosen orderings of values, constants, and functions. The correctness of CSC and CSF was proven in Reger, Riener, and Suda [31].

These schemes are used in modern MACE-style model finders, for example in Reger et al's extensions to the Vampire theorem prover [25] and Vakili and Day's Fortress model finder [18]. For many-sorted problems they use a slight generalization, applying these schemes for each sort independently. That is, apply the schemes for constants of sort A and a function $f : A \to A$ (using values in $\mathcal{D}(A)$), then apply the above for constants of sort B and a function $g : B \to B$ (using values in $\mathcal{D}(B)$), and so on.

3 Iterative Symmetry Breaking Framework

In this section, we develop a framework that uses partially-interpreted MSFMF problems to describe symmetry breaking as an iterative process. This framework enables us to prove the correctness of symmetry breaking schemes while showing how they can be soundly combined.

In a pure MSFMF problem, values of sorts cannot appear in terms. This assumption is key for symmetry breaking,

since it makes all the values interchangeable. However, once we add symmetry breaking formulas, some values appear in the formulas and therefore, we have a partially-interpreted problem.

Definition 8 (Stale and Fresh Values)**.** Let $P = (\Sigma, \Gamma, \mathcal{D})$ be a partially-interpreted MSFMF problem. A value $\mathsf{v} \in \mathcal{D}(S)$ is **stale** for sort S if v appears in any formula of Γ as a term of sort S. Otherwise, v is **fresh** for S. We denote by $\mathrm{Stale}(S)$ and $\mathrm{Fresh}(S)$ the set of values which are stale and fresh for sort S respectively.

Stale values are not necessarily interchangeable. For example, consider a constant $c : A$, distinct values $\mathsf{a}_1, \mathsf{a}_2 \in \mathcal{D}(A)$, and the partially-interpreted formula $c = \mathsf{a}_1$. An interpretation I where $c^I = \mathsf{a}_1$ satisfies this formula. However, swapping a_1 and a_2 produces a new interpretation I' where $c^{I'} = \mathsf{a}_2$, which does not satisfy the formula $c = \mathsf{a}_1$. The permutation does not preserve that I is a solution. Symmetries are those SV permutations that preserve whether or not an interpretation is a solution.

Definition 9 (SV Symmetry)**.** Let P be a partially-interpreted MSFMF problem. An SV permutation σ of P is a **sorted-value symmetry (SV symmetry)** if, for every interpretation I, $\sigma \bullet I$ satisfies P if and only if I satisfies P.

The set of SV symmetries equals the set of SV permutations in pure problems, but not all SV permutations will be SV symmetries in partially-interpreted problems. For a given problem P, we let $\mathrm{Sym}(P)$ be the set of all SV symmetries of P. Now, we say two interpretations I and I' of P are **isomorphic** if $I' = \sigma \bullet I$ for some SV symmetry σ.

While stale values might not be interchangeable, any remaining fresh values are still interchangeable. We give a special name to SV permutations that only permute fresh values.

Definition 10 (Stale-Fixing)**.** Let P be a partially-interpreted MSFMF problem. An SV permutation σ of P is **stale-fixing** if for every sort S, σ_S acts as the identity on $\mathrm{Stale}(S)$.

Stale-fixing permutations are symmetries because they only swap fresh values, which are interchangeable.

Proposition 11 (Stale-Fixing Permutations are SV Symmetries)**.** *Let P be a partially-interpreted MSFMF problem with stale-fixing SV permutation σ. For any interpretation I of P, I satisfies P if and only if $\sigma \bullet I$ satisfies P, and therefore σ is an SV symmetry.*

The presence of fresh values (and of non-trivial SV symmetries) in a partially-interpreted MSFMF problem suggests we can do more symmetry breaking. Therefore the question of how to *sequence* or *compose* multiple symmetry breaking schemes is important.

Claessen and Sörenson only lightly acknowledge this question of composition. The set of values used in the CSF scheme depends on which values were used in the CSC scheme. However, since only one function is ever chosen, it is easy to view their symmetry breaking as a monolithic process, where the constraints are all generated and added at once to the original pure problem.

Composition is significantly more challenging in the many-sorted case. Suppose we have sorts A, B, C, D, and have added some symmetry breaking formula ϕ which uses some sort values. Now consider a function $f : A \times B \to C$ on which we wish to perform symmetry breaking. Does it matter which values of sort A are still fresh? What about sort C? Supposing that we determine some symmetry breaking formulas using f (which would use values of sorts A, B, and C), what then can we do for a function $h : D \times A \to D$?

To tackle this complexity, instead of thinking of adding symmetry breaking formulas all at once to a pure problem, we instead answer a more general question: what symmetry breaking can be done for an MSFMF problem that is already partially-interpreted? Answering this question allows us to view symmetry breaking as an iterative process as in Algorithm 1. We use the notation $P + \phi$ to denote the problem obtained from P by adding the formula ϕ.

Algorithm 1: Iterative Symmetry Breaking

```
input : A pure MSFMF problem P
output: A partially-interpeted MSFMF problem P',
        which is equisatisfiable to P
P' ← P
while there exists a symmetry breaking formula of P' do
    φ ← a symmetry breaking formula of P'
    P' ← P' + φ
end
return P'
```

To make this approach work, we define what a symmetry breaking formula is for partially-interpreted MSFMF problems, and ensure their addition preserves satisfiability.

Definition 12 (Symmetry Breaking Formula)**.** Let $P = (\Sigma, \Gamma, \mathcal{D})$ be a partially-interpreted MSFMF problem. A **symmetry breaking formula** is a partially-interpreted formula ϕ such that, for any interpretation I of P, at least one member of its isomorphism class under $\mathrm{Sym}(P)$ satisfies the formula ϕ (i.e. there exists $\sigma \in \mathrm{Sym}(P)$ where $\sigma \bullet I$ satisfies ϕ).

With this definition, adding symmetry breaking formulas preserves satisfiability [3].

Theorem 13 (Symmetry Breaking Preserves Satisfiability)**.** *If ϕ is a symmetry breaking formula of a partially-interpreted MSFMF problem $P = (\Sigma, \Gamma, \mathcal{D})$, then $P' = (\Sigma, \Gamma \cup \{\phi\}, \mathcal{D})$ is equisatisfiable to P.*

Proof. Since the formulas of P are a subset of the formulas of P', if P' is satisfiable, then so too is P. For the converse, suppose that P is satisfiable with solution I. Since ϕ is a symmetry breaking formula, there exists an SV symmetry σ such that $\sigma \bullet I$ satisfies ϕ. Additionally, since I is a solution of P, I satisfies Γ. By Definition 9, $\sigma \bullet I$ also satisfies Γ. Hence $\sigma \bullet I$ satisfies $\Gamma \cup \{\phi\}$, and thus satisfies P'. □

Sometimes, it is easier to handle collections of symmetry breaking formulas instead of single formulas. It follows triv-

3. Theorem 13 still holds even if we weaken Definition 12 to quantify over only interpretations which are solutions to P. However, this would not help much in practice to prove a formula is symmetry breaking, since we know little about the structure of solutions before search.

ially from Theorem 13 that the addition of these collections also preserves satisfiability, so they could also be used in Algorithm 1.

Definition 14 (Symmetry Breaking Set, Chain)**.** A finite set Φ of partially-interpreted formulas is a **symmetry breaking set** of P if its conjunction $\bigwedge_{\phi\in\Phi}\phi$ is a symmetry breaking formula. A sequence $\phi_1,\ldots,\phi_k$ of partially-interpreted formulas is a **symmetry breaking chain** of P if for $i=1,\ldots,k$, ϕ_i is a symmetry breaking formula for $P+\phi_1+\ldots+\phi_{i-1}$.

For generality in symmetry breaking, our goal is to derive symmetry breaking schemes for partially-interpreted MSFMF problems. The correctness of a scheme is demonstrated by proving that the formulas generated are symmetry breaking formulas, sets, or chains, under Definitions 12 or 14. Then, we are able to use Algorithm 1 and Theorem 13 to iteratively sequence many symmetry breaking schemes while preserving satisfiability. This framework is powerful and robust for designing and combining symmetry breaking schemes.

In Sections 4, 5, and 6, we design symmetry breaking schemes to use in this framework. For these schemes, we are only concerned with stale-fixing symmetries. In a general partially-interpreted problem, some symmetries may arise from the stale values. For example, in the partially-interpreted formula $f(1)=1\wedge f(2)=2$, the values 1 and 2 are stale but still interchangeable. Swapping 1 and 2 is an SV symmetry that is not stale-fixing. However, in our framework, the only stale values are those introduced by our own symmetry breaking formulas. These formulas are designed to be highly asymmetric in their use of values, so the only remaining useful symmetries are those arising from permuting fresh values.

4 Scheme for Range-Domain Independent Functions

For many-sorted problems, the CSC and CSF schemes are limited in applicability because they are restricted to constants and single-sorted functions. Many kinds of functions in MSFOL are not addressed, such as $f:A\to B$, or $g:A\times B\to A$. To increase symmetry breaking for MSFMF, we create schemes for such functions. We categorize an MSFOL function into one of two classes: 1) its output sort is distinct from its input sorts, or 2) its output sort is one of its input sorts.

Definition 15 (Range-Domain Independent/Dependent)**.** A function symbol $f:A_1\times\cdots\times A_k\to B$ is **range-domain independent (RDI)** if its output sort B is distinct from each of its input sorts $A_1,\ldots,A_k$. Otherwise, the symbol is **range-domain dependent (RDD)**.

In the above definition it does not matter whether the input sorts $A_1,\ldots,A_k$ are distinct. In this section, we develop a static symmetry breaking scheme for RDI functions, and prove its correctness using our framework of partially-interpreted MSFMF problems. In Section 5, we discuss symmetry breaking for RDD functions.

4.1 Symmetry Breaking Scheme

We begin by considering a pure problem where all values are fresh, and the simplest RDI function $f:A\to B$. Suppose $\mathcal{D}(A)=\{\mathsf{a}_1,\ldots,\mathsf{a}_m\}$ and $\mathcal{D}(B)=\{\mathsf{b}_1,\ldots,\mathsf{b}_n\}$.

The CSF scheme is complicated by the fact that the range and domain depend on each other, but this is not an issue for RDI functions. We can effectively consider $f(\mathsf{a}_1),\ldots,f(\mathsf{a}_k)$ to be constants from the perspective of B. Thus, we are able to add similar symmetry breaking formulas as the CSC scheme. First, we require that f sends a_i to one of the first i values in $\mathcal{D}(B)$:

$$\begin{aligned}&f(\mathsf{a}_1)=\mathsf{b}_1\\&f(\mathsf{a}_2)\in\{\mathsf{b}_1,\mathsf{b}_2\}\\&f(\mathsf{a}_3)\in\{\mathsf{b}_1,\mathsf{b}_2,\mathsf{b}_3\}\\&\ldots\end{aligned}$$

Second, we require that if $f(\mathsf{a}_i)=\mathsf{b}_j$ then some earlier a_l is assigned b_{j-1}:

$$f(\mathsf{a}_2)=\mathsf{b}_2\implies\mathsf{b}_1=f(\mathsf{a}_1)$$

$$\begin{aligned}f(\mathsf{a}_3)=\mathsf{b}_3&\implies\mathsf{b}_2=f(\mathsf{a}_2)\\f(\mathsf{a}_3)=\mathsf{b}_2&\implies\mathsf{b}_1\in\{f(\mathsf{a}_1),f(\mathsf{a}_2)\}\end{aligned}$$

$$\begin{aligned}f(\mathsf{a}_4)=\mathsf{b}_4&\implies\mathsf{b}_3=f(\mathsf{a}_3)\\f(\mathsf{a}_4)=\mathsf{b}_3&\implies\mathsf{b}_2\in\{f(\mathsf{a}_2),f(\mathsf{a}_3)\}\\f(\mathsf{a}_4)=\mathsf{b}_2&\implies\mathsf{b}_1\in\{f(\mathsf{a}_1),f(\mathsf{a}_2),f(\mathsf{a}_3)\}\end{aligned}$$

$$\ldots$$

In partially-interpreted MSFMF, not all values are fresh, and we cannot just require $f(\mathsf{a}_i)$ to be one of the first i values in $\mathcal{D}(B)$, as some are no longer interchangeable. Instead, we require that $f(\mathsf{a}_i)$ is either any of the stale values of B or one of the first i fresh values of B. Supposing that the fresh values in $\mathrm{Fresh}(B)$ are $\mathsf{b}'_1,\ldots,\mathsf{b}'_k$, this gives us the formulas:

$$\begin{aligned}&f(\mathsf{a}_1)\in\mathrm{Stale}(B)\cup\{\mathsf{b}'_1\}\\&f(\mathsf{a}_2)\in\mathrm{Stale}(B)\cup\{\mathsf{b}'_1,\mathsf{b}'_2\}\\&f(\mathsf{a}_3)\in\mathrm{Stale}(B)\cup\{\mathsf{b}'_1,\mathsf{b}'_2,\mathsf{b}'_3\}\\&\ldots\end{aligned}$$

Similarly, we require that, if f assigns some a_i the j-th fresh value, then some earlier value in $\mathcal{D}(A)$ was assigned the $(j-1)$-th fresh value:

$$f(\mathsf{a}_2)=\mathsf{b}'_2\implies\mathsf{b}'_1=f(\mathsf{a}_1)$$

$$\begin{aligned}f(\mathsf{a}_3)=\mathsf{b}'_3&\implies\mathsf{b}'_2=f(\mathsf{a}_1)\\f(\mathsf{a}_3)=\mathsf{b}'_2&\implies\mathsf{b}'_1\in\{f(\mathsf{a}_1),f(\mathsf{a}_2)\}\end{aligned}$$

$$\begin{aligned}f(\mathsf{a}_4)=\mathsf{b}'_4&\implies\mathsf{b}'_3=f(\mathsf{a}_3)\\f(\mathsf{a}_4)=\mathsf{b}'_3&\implies\mathsf{b}'_2\in\{f(\mathsf{a}_2),f(\mathsf{a}_3)\}\\f(\mathsf{a}_4)=\mathsf{b}'_2&\implies\mathsf{b}'_1\in\{f(\mathsf{a}_1),f(\mathsf{a}_2),f(\mathsf{a}_3)\}\end{aligned}$$

$$\ldots$$

Generalizing to higher-arity RDI functions like $h : A \times A \times C \times C \to B$ is not difficult: list the input tuples as $\mathsf{t}_1, \ldots, \mathsf{t}_m$, and instead of $f(\mathsf{a}_i)$ above, use $h(\mathsf{t}_i)$.

It is important to note that it is irrelevant whether the input values are fresh or stale. This fact may be surprising, but we prove it in the next subsection.

4.2 Proof of Correctness

To verify the RDI scheme soundly breaks symmetries, we use Definitions 10, 12, and 14. We consider an arbitrary interpretation I, and demonstrate the existence of a stale-fixing SV symmetry σ (a collection of permutations that only permute fresh values) so that $\sigma \bullet I$ satisfies the generated formulas. The work comes in constructing the SV symmetry.

First, consider the simplest case of an RDI function, $f : A \to B$. We can visualize an interpretation of f as an array of values of $\mathcal{D}(B)$, indexed by $\mathcal{D}(A) = \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$:

$$\begin{array}{cccccc} \mathsf{a}_1 & \mathsf{a}_2 & \mathsf{a}_3 & \ldots & \mathsf{a}_k \\ [\mathsf{y}_1 & \mathsf{y}_2 & \mathsf{y}_3 & \ldots & \mathsf{y}_k] \end{array}$$

The a_i's are distinct, though the y_i's are not required to be distinct.

Consider the result of applying an SV permutation σ (consisting of a permutation on $\mathcal{D}(A)$ called σ_A and a permutation on $\mathcal{D}(B)$ called σ_B) to this array. The permutation σ_A shuffles the columns, while σ_B relabels the array elements (consistently, since the y_i's need not be distinct). The key insight is that, because the input and output sorts are independent, we are separately able to permute them. In our construction of σ, we leave the columns fixed and simply relabel the array elements.

Interpretations of higher-arity functions are similar, except they can be seen as multidimensional arrays (e.g. a matrix for a binary function). We take the same strategy: only relabel the array elements.

For a pure problem, the permutation we use for σ_B is given in the following lemma. It has properties that mirror the structure of the proposed symmetry breaking formulas, which is what enables us to apply it to an interpretation and have these formulas be satisfied.

Lemma 16. *Let $X = \{b_1, \ldots, b_n\}$ be a set of n elements. Let $\mathsf{y}_1, \ldots, \mathsf{y}_r$ be a sequence of elements of X that are not necessarily distinct, where $r \leq n$. There exists a permutation $\pi : X \to X$ such that, for each $i \in \{1, \ldots, r\}$,*

- *$\pi(\mathsf{y}_i) \in \{\mathsf{b}_1, \ldots, \mathsf{b}_i\}$, and*
- *for all $j \in \{2, \ldots, i\}$, if $\pi(\mathsf{y}_i) = \mathsf{b}_j$ then, for some $l \in \{j-1, \ldots, i-1\}$, $\pi(\mathsf{y}_l) = \mathsf{b}_{j-1}$.*

Such a permutation can be constructed as follows. Assign y_1 to the first element of X. Then, iteratively assign the next unassigned y_i (recall they may not be distinct) to the next element of X not yet matched to a y_j. For example, if the sequence of y_i's is $\mathsf{1}, \mathsf{2}, \mathsf{2}, \mathsf{4}, \mathsf{1}, \mathsf{3}$, we assign $\sigma(\mathsf{1}) = b_1$, $\sigma(\mathsf{2}) = b_2, \sigma(\mathsf{4}) = b_3, \sigma(\mathsf{3}) = b_4$.

Constructing the correct permutation for a partially-interpreted problem is more complicated, since we leave the stale values fixed. To do this, we generalize Lemma 16.

Lemma 17. *Let X be a set of size n, with partition (S, F), where $F = \{b_1, \ldots, b_m\}$. Let $\mathsf{y}_1, \ldots, \mathsf{y}_r$ be a sequence of $r \leq m$ elements of X that are not necessarily distinct. There exists a permutation $\pi : X \to X$ such that, π acts as the identity on S, and for each $i \in \{1, \ldots, r\}$,*

- *$\pi(\mathsf{y}_i) \in S \cup \{\mathsf{b}_1, \ldots, \mathsf{b}_i\}$, and*
- *for all $j \in \{2, \ldots, i\}$, if $\pi(\mathsf{y}_i) = \mathsf{b}_j$ then, for some $l \in \{j-1, \ldots, i-1\}$, $\pi(\mathsf{y}_l) = \mathsf{b}_{j-1}$.*

Proof. Iteratively for each $i = 1, \ldots, r$, define $\pi(\mathsf{y}_i)$ as follows:

- If $\pi(\mathsf{y}_i)$ is already defined (due to a repetition in the sequence), keep it as is.
- If $\mathsf{y}_i \in S$ and $\pi(\mathsf{y}_i)$ is undefined, define $\pi(\mathsf{y}_i)$ to be y_i.
- If $\mathsf{y}_i \in F$ and $\pi(\mathsf{y}_i)$ is undefined, define $\pi(\mathsf{y}_i)$ to be the first element of F not yet in the range of π.

It can then be verified by induction that π is a partial injection from X to X such that, for each $i \in \{1, \ldots, r\}$,

- if $\mathsf{y}_i \in S$, then $\pi(\mathsf{y}_i) = \mathsf{y}_i$,
- if $\mathsf{y}_i \in F$, then $\pi(\mathsf{y}_i)$ is one of the i first values of F, and
- for all $j \in \{2, \ldots, i\}$, if $\sigma(\mathsf{y}_i) = \mathsf{b}_j$, then $\sigma(\mathsf{y}_l) = \mathsf{b}_{j-1}$ for some $l < i$ (also $l \geq j-1$ by the above properties).

Completing π to a permutation so that it remains the identity on S, we obtain a permutation with the desired properties. □

We now prove the correctness of the RDI scheme for symmetry breaking.

Theorem 18 (RDI Scheme)**.** *Let $P = (\Sigma, \Gamma, \mathcal{D})$ be a partially-interpreted MSFMF problem with RDI functional symbol $f : A_1 \times \cdots \times A_k \to B$. Let $\mathsf{t}_1, \ldots, \mathsf{t}_m$ be the tuples of $\mathcal{D}(A_1) \times \cdots \times \mathcal{D}(A_k)$ and let $\mathsf{b}_1, \ldots, \mathsf{b}_n$ be the elements of* $\mathrm{Fresh}(B)$*. Let $r = \min\{m, n\}$. Define $\Phi = \{\phi_i : 1 \leq i \leq r\} \cup \{\phi'_i : 1 \leq i \leq r\}$, where:*

$$\phi_i := f(\mathsf{t}_i) \in \mathrm{Stale}(B) \cup \{\mathsf{b}_1, \ldots, \mathsf{b}_i\}$$
$$\phi'_i := \bigwedge_{2 \leq j \leq i} f(\mathsf{t}_i) = \mathsf{b}_j \implies \mathsf{b}_{j-1} \in \{f(\mathsf{t}_{j-1}), \ldots, f(\mathsf{t}_{i-1})\}$$

Then Φ is a symmetry breaking set.

Proof. Let I be an interpretation of P. We prove there exists an SV symmetry σ such that $\sigma \bullet I$ satisfies Φ.

For each sort θ that is not B, choose σ_θ to be the identity map on $\mathcal{D}(\theta)$. Now, let $X = \mathcal{D}(B)$, $(S, F) = (\mathrm{Stale}(B), \mathrm{Fresh}(B))$, and let $\mathsf{y}_i = f^I(\mathsf{t}_i)$ for each i. Define σ_B to be the permutation π defined as in Lemma 17. Since σ only permutes values of $\mathrm{Fresh}(B)$, it is stale-fixing, and hence, an SV symmetry by Proposition 11.

To show that $\sigma \bullet I$ satisfies Φ, we prove that, for each $i \in \{1, \ldots, r\}$, $\sigma \bullet I$ satisfies ϕ_i and ϕ'_i. Since σ acts as the identity on the input values of f, we have that for any $j \in \{1, \ldots, m\}$,

$$\begin{aligned} f^{\sigma \bullet I}(\mathsf{t}_j) &= \sigma_B(f^I(\sigma_{A_1}^{-1}(\mathsf{t}_{j,1}), \ldots, \sigma_{A_k}^{-1}(\mathsf{t}_{j,k})) \\ &= \sigma_B(f^I(\mathsf{t}_j)) \\ &= \sigma_B(\mathsf{y}_j) \end{aligned}$$

It then follows from Lemma 17 that

$$f^{\sigma \bullet I}(\mathsf{t}_i) = \sigma_B(\mathsf{y}_i) \in \mathrm{Stale}(B) \cup \{\mathsf{b}_1, \ldots, \mathsf{b}_i\},$$

and so ϕ_i is satisfied by $\sigma \bullet I$. Furthermore, for all $2 \leq j \leq i$, if $\sigma_B(\mathsf{y}_i) = \mathsf{b}_j$ then $\mathsf{b}_{j-1} \in \{\sigma_B(\mathsf{y}_{j-1}), \ldots, \sigma_B(\mathsf{y}_{i-1})\}$. Therefore, for all $2 \leq j \leq i$, if $f^{\sigma \bullet I}(\mathsf{t}_i) = \mathsf{b}_j$ then $\mathsf{b}_{j-1} \in$

$\{f^{\sigma \bullet I}(\mathsf{t}_{j-1}), \ldots, f^{\sigma \bullet I}(\mathsf{t}_{i-1})\}$. Thus, ϕ_i' is satisfied by $\sigma \bullet I$. □

Because of this proof, we know that these formulas constitute a symmetry breaking set. Thus, we can safely add RDI symmetry breaking formulas to a partially-interpreted MSFMF problem, reducing the search space for solutions. By Theorem 13, the problem with the added symmetry breaking formulas is equisatisfiable to the original problem. Also note that Theorem 18 does not require the input values of the function be fresh.

5 Scheme for Range-Domain Dependent Functions

A range-domain dependent (RDD) function, such as $g : A \times A \times B \times B \times C \to A$, has one of its input sorts as its output sort. An SV symmetry cannot independently permute the input and output sort values like in Theorem 18, so a different symmetry breaking scheme is needed.

In the case of single-sorted functions like $h : A \times A \to A$, we can use the standard CSF scheme. However, it does not apply to RDD functions with multiple sorts, like $g : A \times B \to A$. Additionally, the CSF scheme has two cases depending on whether or not constants were constrained beforehand. In this section we develop a more general, iterative symmetry breaking scheme that handles any RDD function, and we describe it in a way that neatly consolidates the two cases of the CSF scheme.

5.1 Unifying CSF Cases with an Iterative Scheme

Consider a function $f : A \to A$ in a partially-interpreted problem (some values may be stale), and a particular value $\hat{\mathsf{a}} \in \mathcal{D}(A)$ (which may or may not be fresh) and consider how we might constrain $f(\hat{\mathsf{a}})$. Recall from the CSF scheme that, because of the domain-range dependence, the choice that $f(\hat{\mathsf{a}}) = \hat{\mathsf{a}}$ may not be equivalent to the choice that $f(\hat{\mathsf{a}}) = \mathsf{a}$ for a different $\mathsf{a} \in \mathcal{D}(A)$. In general, there are three cases for the values of $f(\hat{\mathsf{a}})$:

1) $f(\hat{\mathsf{a}})$ has value $\hat{\mathsf{a}}$ itself,
2) $f(\hat{\mathsf{a}})$ has value different from $\hat{\mathsf{a}}$ and is stale, or
3) $f(\hat{\mathsf{a}})$ has value different from $\hat{\mathsf{a}}$ and is fresh.

The last case presents opportunity for symmetry breaking: any choice of value in $\mathrm{Fresh}(A) \setminus \{\hat{\mathsf{a}}\}$ is equivalent for $f(\hat{\mathsf{a}})$. Combining the above, we may add the following formula, where a^* is an arbitrary representative of $\mathrm{Fresh}(A) \setminus \{\hat{\mathsf{a}}\}$:

$$f(\hat{\mathsf{a}}) \in \{\hat{\mathsf{a}}\} \cup [\mathrm{Stale}(A) \setminus \{\hat{\mathsf{a}}\}] \cup \{\mathsf{a}^*\}$$

which is more compactly written as:

$$f(\hat{\mathsf{a}}) \in \mathrm{Stale}(A) \cup \{\hat{\mathsf{a}}, \mathsf{a}^*\}$$

We add this formula to the problem, update the sets of stale and fresh values, and iteratively repeat this process for another choice of $\hat{\mathsf{a}}$. This gives us Algorithm 2.

The algorithm has some flexibility in the values $\hat{\mathsf{a}}$ and a^* chosen for each iteration. As a concrete example, suppose initially $\mathcal{D}(A) = \{1,2,3,4,5,6\}$ and $\mathrm{Fresh}(A) = \{3,4,5,6\}$. One possible execution, always choosing a stale value for $\hat{\mathsf{a}}$, is:

Algorithm 2: RDD Symmetry Breaking Scheme

input : A partially-interpreted MSFMF problem P' with function $f : A \to A$
output: A symmetry breaking chain Φ
$\Phi \leftarrow$ the empty sequence
$(S, F) \leftarrow (\mathrm{Stale}(A), \mathrm{Fresh}(A))$
while $F \neq \emptyset$ **do**
  $\hat{\mathsf{a}} \leftarrow$ a value in $\mathcal{D}(A)$
  $\mathsf{a}^* \leftarrow$ a value in $F \setminus \{\hat{\mathsf{a}}\}$
  $\phi \leftarrow$ the formula $f(\hat{\mathsf{a}}) \in S \cup \{\hat{\mathsf{a}}, \mathsf{a}^*\}$
  $\Phi \leftarrow \Phi$ appended with ϕ
  $(S, F) \leftarrow (S \cup \{\hat{\mathsf{a}}, \mathsf{a}^*\}, F \setminus \{\hat{\mathsf{a}}, \mathsf{a}^*\})$
end
return Φ

Iter	F	$\hat{\mathsf{a}}$	a^*	Formula
1	$\{3,4,5,6\}$	1	3	$f(1) \in \{1,2,3\}$
2	$\{4,5,6\}$	2	4	$f(2) \in \{1,2,3,4\}$
3	$\{5,6\}$	3	5	$f(3) \in \{1,2,3,4,5\}$
4	$\{6\}$	4	6	$f(4) \in \{1,2,3,4,5,6\}$

Another possible execution with different values of $\hat{\mathsf{a}}$ is:

Iter	F	$\hat{\mathsf{a}}$	a^*	Formula
1	$\{3,4,5,6\}$	3	4	$f(3) \in \{1,2,3,4\}$
2	$\{5,6\}$	2	5	$f(2) \in \{1,2,3,4,5\}$
3	$\{6\}$	4	6	$f(4) \in \{1,2,3,4,5,6\}$

Note that $\hat{\mathsf{a}}$ should be chosen to be different from each previous $\hat{\mathsf{a}}$, otherwise the formula generated will be redundant (but not incorrect). Both forms of the CSF scheme are generated by selecting $\hat{a}, a^*$ according to the following rules: choose $\hat{\mathsf{a}}$ to be the least value not yet selected by an iteration, and select a^* to be as small as possible. So our RDD scheme unifies the two cases of CSF.

As with Claessen and Sörenson's scheme, for a higher-arity single-sorted function $h : A \times \cdots \times A \to A$, we simply use $h(\hat{\mathsf{a}}, \ldots, \hat{\mathsf{a}})$ instead of $f(\hat{\mathsf{a}})$. Intuitively, we can pretend we are creating a new function $h' : A \to A$ by collapsing h so that $h'(\mathsf{a}) = h(\mathsf{a}, \ldots, \mathsf{a})$ for all $\mathsf{a} \in \mathcal{D}(A)$ and applying the unary scheme to h'.

5.2 Generalizing to Multiple Sorts

Suppose now we wish to apply a similar technique for a function $g : A \times A \times B \times B \times C \to A$. The most straightforward approach is to extend the "collapsing" method above. We fix particular values $\mathsf{b}, \mathsf{b}' \in \mathcal{D}(B), \mathsf{c} \in \mathcal{D}(C)$, held constant throughout the iterations, and use the same iterative procedure with $h(\hat{\mathsf{a}}, \hat{\mathsf{a}}, \mathsf{b}, \mathsf{b}', \mathsf{c})$ instead of $f(\hat{\mathsf{a}})$.

For instance, suppose $\mathcal{D}(A) = \{1,2,3,4\}$, and initially $\mathrm{Fresh}(A) = \{2,3,4\}$. Then, fixing arbitrary $\mathsf{b}, \mathsf{b}' \in \mathcal{D}(B)$ and $\mathsf{c} \in D(C)$, an example algorithm execution is:

Iter	F	$\hat{\mathsf{a}}$	a^*	Formula
1	$\{2,3,4\}$	1	2	$h(1,1,\mathsf{b},\mathsf{b}',\mathsf{c}) \in \{1,2\}$
2	$\{3,4\}$	2	4	$h(2,2,\mathsf{b},\mathsf{b}',\mathsf{c}) \in \{1,2,4\}$
3	$\{3\}$	4	3	$h(4,4,\mathsf{b},\mathsf{b}',\mathsf{c}) \in \{1,2,3,4\}$

We will show that it does not matter whether $\mathsf{b}, \mathsf{b}', \mathsf{c}$ are fresh, it only matters what values of output sort A are fresh.

5.3 Proof of Correctness

For the proof, we show that each step produces a symmetry breaking formula; then by induction the algorithm produces a symmetry breaking chain (Definition 14). Attempting a proof that handles RDD functional symbols with arbitrary sort configurations gives rise to cumbersome notation, so instead we only prove the correctness of this scheme for the specific functional symbol $g : A \times A \times B \times B \times C \to A$. This symbol is sufficiently complex, with multiple and repeated sorts, to demonstrate how to generalize to all RDD functional symbols.

Theorem 19 (RDD Scheme, Single-Step)**.** *Let $P = (\Sigma, \Gamma, \mathcal{D})$ be a partially-interpreted MSFMF problem with functional symbol $g : A \times A \times B \times B \times C \to A$. Let $\mathsf{b}, \mathsf{b}' \in \mathcal{D}(B), \mathsf{c} \in \mathcal{D}(C)$. Let $\hat{\mathsf{a}} \in \mathcal{D}(A)$ and let $\mathsf{a}^* \in \mathrm{Fresh}(A) \setminus \{\hat{\mathsf{a}}\}$ ($\hat{\mathsf{a}}$ may or may not be fresh). The formula*

$$\phi := g(\hat{\mathsf{a}}, \hat{\mathsf{a}}, \mathsf{b}, \mathsf{b}', \mathsf{c}) \in \mathrm{Stale}(A) \cup \{\hat{\mathsf{a}}, \mathsf{a}^*\} \tag{1}$$

is a symmetry breaking formula.

Proof. Let I be an interpretation of P. We prove there exists an SV symmetry σ such that $\sigma \bullet I$ satisfies ϕ.

Let $\mathsf{z} = g^I(\hat{\mathsf{a}}, \hat{\mathsf{a}}, \mathsf{b}, \mathsf{b}', \mathsf{c}) \in \mathcal{D}(A)$. There are two cases.

Case 1: Suppose $\mathsf{z} \in \mathrm{Stale}(A) \cup \{\hat{\mathsf{a}}\}$. This means that $g^I(\hat{\mathsf{a}}, \hat{\mathsf{a}}, \mathsf{b}, \mathsf{b}', \mathsf{c}) \in \mathrm{Stale}(A) \cup \{\hat{\mathsf{a}}\}$, so it follows that I already satisfies ϕ. We simply take σ to be the identity permutation for each sort, which is clearly an SV symmetry. Then, $\sigma \bullet I = I$ and so $\sigma \bullet I$ satisfies ϕ.

Case 2: Suppose $\mathsf{z} \in \mathrm{Fresh}(A)$, and $\mathsf{z} \neq \hat{\mathsf{a}}$. Define σ_θ to be the identity on each sort $\theta \neq A$. Define σ_A to be the permutation that swaps a^* and z, but leaves all other values fixed. Since σ is the identity on all values except some in $\mathrm{Fresh}(A)$, it is stale-fixing and hence an SV symmetry by Proposition 11. Note that since $\mathsf{a}^* \neq \hat{\mathsf{a}}$ and $\mathsf{z} \neq \hat{\mathsf{a}}$, σ_A acts as the identity on $\hat{\mathsf{a}}$. Then we have that

$$\begin{aligned}
&g^{\sigma \bullet I}(\hat{\mathsf{a}}, \hat{\mathsf{a}}, \mathsf{b}, \mathsf{b}', \mathsf{c}) \\
&= \sigma_A(g^I(\sigma_A^{-1}(\hat{\mathsf{a}}), \sigma_A^{-1}(\hat{\mathsf{a}}), \sigma_B^{-1}(\mathsf{b}), \sigma_B^{-1}(\mathsf{b}'), \sigma_C^{-1}(\mathsf{c}))) \\
&= \sigma_A(g^I(\hat{\mathsf{a}}, \hat{\mathsf{a}}, \mathsf{b}, \mathsf{b}', \mathsf{c})) \\
&= \sigma_A(\mathsf{z}) \\
&= \mathsf{a}^*
\end{aligned}$$

Therefore $\sigma \bullet I$ satisfies ϕ. □

This theorem did not require that $\mathsf{b}, \mathsf{b}', \mathsf{c}$ are fresh, confirming that it only matters which values of the output sort A are fresh. Also, while for the discussion about "collapsing" we hold $\mathsf{b}, \mathsf{b}', \mathsf{c}$ constant over the iterations, the theorem does not use this restriction. Instead, it allows $\mathsf{b}, \mathsf{b}', \mathsf{c}$ to be chosen at each step, meaning they can be chosen differently for each iteration.

6 Scheme for Relations

The schemes presented in Sections 4 and 5 cover every function in MSFOL. However, they do not cover relations like $Q : A \to \mathrm{Bool}$. Unlike function applications, applications of relations are *formulas*, not *terms*. That is, an application $Q(x)$ of a relation Q does not output an object, but True or False. Symmetry breaking schemes using functions rely on the interchangeability of their output values. However, True and False are not interchangeable, so a different approach is needed. In this section, we provide such a scheme.

6.1 Symmetry Breaking Scheme

First consider a pure MSFMF problem, with relational symbol $Q : A \to \mathrm{Bool}$ where $\mathcal{D}(A) = \{\mathsf{a}_1, \ldots, \mathsf{a}_m\}$. An interpretation I assigns Q a unary relation $Q^I \subseteq \mathcal{D}(A)$, which is simply a set of values. Because values are interchangeable, any choice for Q^I of size k is equivalent to any other choice for Q^I of size k: if there is no solution where $Q^I = \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$, then there is no solution where $Q^I = Z$ for any other k-subset $Z \subseteq \mathcal{D}(A)$. Thus, we need only try assigning Q^I to be $\emptyset, \{\mathsf{a}_1\}, \{\mathsf{a}_1, \mathsf{a}_2\}, \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\}, \ldots, \{\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_m\}$ to determine whether a solution exists. This restriction is encoded by the following formulas, which we call **ladder implications**:

$$\begin{aligned}
Q(\mathsf{a}_2) &\implies Q(\mathsf{a}_1) \\
Q(\mathsf{a}_3) &\implies Q(\mathsf{a}_2) \\
&\ldots \\
Q(\mathsf{a}_m) &\implies Q(\mathsf{a}_{m-1})
\end{aligned}$$

For partially-interpreted problems, we only add these ladder implications for the fresh values. In particular, if $\mathrm{Fresh}(A) = \{\mathsf{a}'_1, \ldots, \mathsf{a}'_r\}$, the ladder formulas to add are:

$$\begin{aligned}
Q(\mathsf{a}'_2) &\implies Q(\mathsf{a}'_1) \\
Q(\mathsf{a}'_3) &\implies Q(\mathsf{a}'_2) \\
&\ldots \\
Q(\mathsf{a}'_r) &\implies Q(\mathsf{a}'_{r-1})
\end{aligned}$$

This approach can be generalized to higher-arity relations. If we have a relation $R : A \times A \times B \times B \times C \to \mathrm{Bool}$, we fix particular particular values $\mathsf{b}, \mathsf{b}' \in \mathcal{D}(B), \mathsf{c} \in \mathcal{D}(C)$, and add the following formulas:

$$\begin{aligned}
R(\mathsf{a}'_2, \mathsf{a}'_2, \mathsf{b}, \mathsf{b}', \mathsf{c}) &\implies R(\mathsf{a}'_1, \mathsf{a}'_1, \mathsf{b}, \mathsf{b}', \mathsf{c}) \\
R(\mathsf{a}'_3, \mathsf{a}'_3, \mathsf{b}, \mathsf{b}', \mathsf{c}) &\implies R(\mathsf{a}'_2, \mathsf{a}'_2, \mathsf{b}, \mathsf{b}', \mathsf{c}) \\
&\ldots \\
R(\mathsf{a}'_r, \mathsf{a}'_r, \mathsf{b}, \mathsf{b}', \mathsf{c}) &\implies R(\mathsf{a}'_{r-1}, \mathsf{a}'_{r-1}, \mathsf{b}, \mathsf{b}', \mathsf{c})
\end{aligned}$$

As we will show, it does not matter whether $\mathsf{b}, \mathsf{b}', \mathsf{c}$ are fresh.

6.2 Proof of Correctness

Now we prove the correctness of this ladder scheme. As in the previous section, we only prove correctness using a specific relational symbol, $R : A \times A \times B \times B \times C \to \mathrm{Bool}$. This example is sufficiently complex to show how to generalize to any relation.

Theorem 20 (Ladder Scheme)**.** *Let $P = (\Sigma, \Gamma, \mathcal{D})$ be a partially-interpreted MSFMF problem with relational symbol $R : A \times A \times B \times B \times C \to \mathrm{Bool}$. Let $\mathsf{a}_1, \ldots, \mathsf{a}_m$ be the elements of $\mathrm{Fresh}(A)$, let $\mathsf{b}, \mathsf{b}' \in \mathcal{D}(B)$, and let $\mathsf{c} \in \mathcal{D}(C)$. The set $\Phi = \{\phi_i : 2 \leq i \leq m\}$ is a symmetry breaking set, where:*

$$\phi_i := R(\mathsf{a}_i, \mathsf{a}_i, \mathsf{b}, \mathsf{b}', \mathsf{c}) \implies R(\mathsf{a}_{i-1}, \mathsf{a}_{i-1}, \mathsf{b}, \mathsf{b}', \mathsf{c})$$

Proof. For any interpretation J of P, define R^{J*} to be the set of all $\mathsf{a} \in \mathcal{D}(A)$ such that $(\mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{b}', \mathsf{c}) \in R^J$ (essentially,

we are collapsing R to a unary relation on A by removing the repetition of A and folding in the constants of sorts B and C). Note that an interpretation J satisfies Φ if and only if $R^{J*} \cap \mathrm{Fresh}(A)$ is one of the following sets: $\emptyset, \{\mathsf{a}_1\}, \{\mathsf{a}_1, \mathsf{a}_2\}, \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\}, \ldots, \{\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_m\}$.

To show that Φ is a symmetry breaking set, we will use Definitions 12 and 14. Let I be any interpretation of P. We will construct an SV symmetry σ where $\sigma \bullet I$ satisfies Φ.

Write $R^{I*} \cap \mathrm{Fresh}(A)$ as $\{\mathsf{y}_1, \ldots, \mathsf{y}_k\}$. Note that $k \leq m$. Define a permutation $\pi : \mathrm{Fresh}(A) \to \mathrm{Fresh}(A)$ such that $\pi(\mathsf{y}_j) = \mathsf{a}_j$ for each $j = 1, \ldots, k$ (if $m > k$ then the rest of the permutation may be defined arbitrarily). Now define σ_A to be the union of the identity permutation on $\mathrm{Stale}(A)$ and the permutation π on $\mathrm{Fresh}(A)$. Finally, define σ_S to be the identity on $\mathcal{D}(S)$ for any sort $S \neq A$. The resulting σ is stale-fixing, and hence is an SV symmetry by Proposition 11.

It remains to show that $\sigma \bullet I$ satisfies Φ. Note that since σ leaves values of all sorts besides A fixed, $R^{(\sigma \bullet I)*}$ is obtained simply by applying σ_A to each element of R^{I*}.

Now, since σ_A maps fresh values to fresh values and stale values to stale values, $R^{(\sigma \bullet I)*} \cap \mathrm{Fresh}(A)$ is obtained by applying π to each element of $R^{I*} \cap \mathrm{Fresh}(A)$. Hence, $R^{(\sigma \bullet I)*} \cap \mathrm{Fresh}(A) = \{\pi(\mathsf{y}_1), \ldots, \pi(\mathsf{y}_k)\} = \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$. Since this is one of the sets $\emptyset, \{\mathsf{a}_1\}, \{\mathsf{a}_1, \mathsf{a}_2\}, \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\}, \ldots, \{\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_m\}$, it follows that $\sigma \bullet I$ satisfies Φ. □

Note that the theorem does not require the values of sorts B and C to be fresh.

6.3 Applying the Scheme Multiple Times

This technique can be pushed further by applying it multiple times to the same relation, changing which sort is focused on. For example, *after* adding the above formulas, there may still be fresh values of set B (though b, b' are stale). If these fresh values are $\{\mathsf{b}_1, \mathsf{b}_2, \ldots, \mathsf{b}_l\}$, we can fix $\mathsf{a}, \mathsf{a}' \in \mathcal{D}(A), \mathsf{c}' \in \mathcal{D}(C)$ ($\mathsf{a}, \mathsf{a}', \mathsf{c}$ not necessarily fresh) to add ladder formulas for B:

$$
\begin{aligned}
&R(\mathsf{a}, \mathsf{a}', \mathsf{b}_2, \mathsf{b}_2, \mathsf{c}') \implies R(\mathsf{a}, \mathsf{a}', \mathsf{b}_1, \mathsf{b}_1, \mathsf{c}') \\
&R(\mathsf{a}, \mathsf{a}', \mathsf{b}_3, \mathsf{b}_3, \mathsf{c}') \implies R(\mathsf{a}, \mathsf{a}', \mathsf{b}_2, \mathsf{b}_2, \mathsf{c}') \\
&\ldots \\
&R(\mathsf{a}, \mathsf{a}', \mathsf{b}_l, \mathsf{b}_l, \mathsf{c}') \implies R(\mathsf{a}, \mathsf{a}', \mathsf{b}_{l-1}, \mathsf{b}_{l-1}, \mathsf{c}')
\end{aligned}
$$

We could then similarly apply a third round of symmetry breaking by adding ladder formulas for C, using the remaining values of $\mathcal{D}(C)$ that are still fresh.

7 Symmetry Detection Using Sort Inference

Though Claessen and Sörenson [16] used unsorted FOL, they were interested in the potential performance benefits of inferring sorts and increasing the symmetry breaking that could be performed. In this section, first, we present an explanation of sort inference, borrowing terminology from type systems in programming languages [32]. Next we demonstrate how the symmetry breaking schemes introduced in this paper increase the benefit of sort inference beyond Claessen and Sörenson's work. Finally, we present a new symmetry framework that clarifies the formal relationship between sort inference and the existence of symmetries.

7.1 Sort Inference

Consider the problem of finding a Latin square. The goal is to fill an $n \times n$ grid of cells with the numbers $\mathsf{1}, \ldots, \mathsf{n}$ so that no row or column repeats a number. One MSFMF formulation uses a single sort N, with domain $\mathcal{D}(N) = \{\mathsf{1}, \ldots, \mathsf{n}\}$, and a function $f : N \times N \to N$ mapping coordinates to numbers. The constraints are encoded as:

$$
\begin{aligned}
&\forall r : N.\ \forall c_1, c_2 : N.\ f(r, c_1) = f(r, c_2) \implies c_1 = c_2 \\
&\forall c : N.\ \forall r_1, r_2 : N.\ f(r_1, c) = f(r_2, c) \implies r_1 = r_2
\end{aligned}
$$

We call this single-sorted MSFMF problem $\mathrm{Latin}_{n,S}$.

An alternate formulation uses three sorts R, C, and E, each having the same domain $\mathcal{D}'(R) = \mathcal{D}'(C) = \mathcal{D}'(E) = \{\mathsf{1}, \ldots, \mathsf{n}\}$. There is a function $f : R \times C \to E$, again mapping coordinates to numbers. The formulas are as follows:

$$
\begin{aligned}
&\forall r : R.\ \forall c_1, c_2 : C.\ f(r, c_1) = f(r, c_2) \implies c_1 = c_2 \\
&\forall c : C.\ \forall r_1, r_2 : R.\ f(r_1, c) = f(r_2, c) \implies r_1 = r_2
\end{aligned}
$$

We call this problem $\mathrm{Latin}_{n,M}$.

The two MSFMF problems are different, but both model the same structure. We formalize their relationship through sort substitution.

Definition 21 (Sort Substitution)**.** Let Θ, Θ' be sets of sorts. A **sort substitution** is a mapping $\eta : \Theta \to \Theta'$.

Note that η need not be injective, and if it is, we say the substitution is **trivial** (since it simply defines a renaming of sorts). Otherwise, η can map multiple sorts in Θ to the same sort in Θ'. We let $\eta^{-1}(S')$ be the set of all S such that $\eta(S) = S'$.

We define the application of sort substitution $\eta : \Theta \to \Theta'$ to a signature $\Sigma = (\Theta, \mathcal{F}, \mathcal{R})$ in the natural way, replacing Θ with its image under η and applying η to the sorts inside the functional and relational symbols. For a Σ-formula γ, we define $\eta(\gamma)$ as the $\eta(\Sigma)$-formula obtained from γ by applying η to each sort of the quantified variables. We apply η to a set Γ of Σ-formulas by applying it pointwise.

We also define the application of η to a domain assignment $\mathcal{D}$ for Θ, provided that $\mathcal{D}$ satisfies the property that for all $S_1, S_2 \in \Theta$ where $\eta(S_1) = \eta(S_2)$, we have $\mathcal{D}(S_1) = \mathcal{D}(S_2)$. That is, for all $S' \in \Theta'$, the domain assignment gives the same set of values for all sorts $S \in \eta^{-1}(S')$, which we will call $\mathcal{D}(\eta^{-1}(S'))$. In such a case, we say η and $\mathcal{D}$ **agree**, and define $\eta(D)$ to be the domain assignment for Θ' that assigns each $S' \in \Theta'$ the set of values $\mathcal{D}(\eta^{-1}(S'))$.

Then, for a MSFMF problem $P = (\Sigma, \Gamma, \mathcal{D})$, we define $\eta(P) = (\eta(\Sigma), \eta(\Gamma), \eta(D))$. For the Latin squares problem, if we consider the sort substitution $\eta : \{R, C, E\} \to \{N\}$ defined by $\eta(R) = \eta(C) = \eta(E) = N$, we see that $\eta(\mathrm{Latin}_{n,M}) = \mathrm{Latin}_{n,S}$.

Definition 22 (More Generally Sorted)**.** Let $P_{\mathrm{Gen}} = (\Sigma, \Gamma, \mathcal{D})$ and $P = (\Sigma', \Gamma', \mathcal{D}')$ be MSFMF problems. If there exists a sort substitution η (that agrees with $\mathcal{D}$) such that $\eta(P_{\mathrm{Gen}}) = P$, we say that P_{Gen} is **more generally sorted** than, or **generates**, P, and write $P_{\mathrm{Gen}} \sqsubseteq P$.

This relationship defines a partial order on MSFMF problems (provided we consider problems related by a trivial sort substitution to be "equal" in this ordering). The process of starting with P and finding a P_{Gen} with a maximal number of sorts which generates P is **sort inference**. It is analogous to type inference in programming languages, and standard type inference algorithms (see [32]) are easily adapted to sort inference. Note that in Definition 22, the problem P might already have multiple sorts (the problem P_{Gen} will have at least as many sorts).

7.2 Sort Inference and Symmetry Breaking

We can observe that the possible interpretations of the functional symbol f are the same in both $\text{Latin}_{n,M}$ and $\text{Latin}_{n,S}$, since they are both functions from $\{1,\ldots,n\}\times\{1,\ldots,n\}$ to $\{1,\ldots,n\}$.

This holds more generally. Let $P_{\text{Gen}} = (\Sigma,\Gamma,\mathcal{D})$ and $P = (\Sigma',\Gamma',\mathcal{D}')$, and suppose $P_{\text{Gen}} \sqsubseteq P$, through a sort substitution η (which agrees with $\mathcal{D}$). The following property holds: if $\eta(S) = S'$, then $\mathcal{D}(S) = (\eta(D))(S')$. This means that a functional symbol $f : A_1 \times \cdots \times A_n \to B$ of P_{Gen} and the corresponding $f : \eta(A_1)\times\cdots\times\eta(A_n) \to \eta(B)$ in P will have the same set of interpretations, namely the set of functions from $\mathcal{D}(A_1) \times \cdots \mathcal{D}(A_n)$ to $\mathcal{D}(B)$. This is similarly true for corresponding relational symbols. There is thus a natural correspondence between interpretations of P_{Gen} and interpretations of P, where we say an interpretation I_{Gen} of the former and an interpretation I of the latter correspond if they give corresponding functional and relational symbols the same interpretation.

If I_{Gen} and I correspond in this way, then I_{Gen} satisfies a Σ-formula γ if and only if I satisfies the formula $\eta(\gamma)$. There are twofold implications of this. First, I_{Gen} satisfies P_{Gen} if and only if I satisfies P. Second, if adding a formula ϕ to P_{Gen} does not change the satisfiability of P_{Gen}, then $\eta(\phi)$ can be added to P without changing satisfiability. In particular, this holds if ϕ is a symmetry breaking formula of P_{Gen}.

Claessen and Sörenson [16], and later Reger et al. [25], used this observation to increase symmetry breaking. Given an initial problem P, they would perform sort inference to find a P_{Gen} which generates P, find symmetry breaking constraints for P_{Gen}, then map those constraints back to P. Because symmetry breaking can be applied to each sort separately, the greater number of sorts obtained from sort inference offers stronger symmetry breaking formulas.

For instance, suppose in P there is a sort A with constants $c_1 : A, c_2 : A$, and $\mathcal{D}(A) = \{1,2\}$. The only useful formula yielded by the CSC scheme is $c_1 = 1$. Now, suppose $P_{\text{Gen}} \sqsubseteq P$, and in P_{Gen} the constants are sorted as $c_1 : B, c_2 : C$. The CSC scheme yields $c_1 = 1$ and $c_2 = 1$, which is stronger (fewer interpretations satisfy the formulas) than the single formula generated for P.

The symmetry breaking schemes introduced in this article derive even more value from sort inference. For instance, consider $\text{Latin}_{3,S}$ and $\text{Latin}_{3,M}$. In $\text{Latin}_{3,S}$, the function f is single-sorted, and so we use the CSF scheme, yielding the formulas:

$$f(1,1) \in \{1,2\}$$
$$f(2,2) \in \{1,2,3\}$$

However, applying sort inference to obtain the $\text{Latin}_{3,M}$ problem, the function f is now RDI. We are able to use our new RDI scheme, obtaining the following formulas (redundant formulas omitted):

$$f(1,1) = 1$$
$$f(2,2) \in \{1,2\}$$
$$f(3,3) \in \{1,2,3\}$$
$$f(3,3) = 3 \implies 2 = f(2,2)$$

These formulas are much stronger. While 13122 interpretations satisfy the CSF formulas, only 3645 satisfy the RDI formulas, a decrease of about 72%. Hence, even for unsorted problems like $\text{Latin}_{n,S}$, after sort inference our new schemes can be used to dramatically increase symmetry breaking over existing techniques.

7.3 Sort Inference Detects More General Symmetries

We noted previously that if $P_{\text{Gen}} \sqsubseteq P$, then the semantics of the two problems are highly linked: corresponding interpretations either both satisfy their respective problems, or both do not satisfy their respective problems. However, there are a couple of oddities about this relationship that suggest a deeper connection with symmetries.

First, the number of SV symmetries in P_{Gen} and P may not be the same. For example, since $\text{Latin}_{n,S}$ has only one sort, it has $n!$ SV symmetries. However, $\text{Latin}_{n,M}$ has $(n!)^3$ SV symmetries, since it has three sorts which may be separately permuted to yield SV symmetries.

The second oddity concerns symmetry breaking formulas. For example, a symmetry breaking formula (from the RDI scheme) for the multi-sorted problem $\text{Latin}_{2,M}$ is:

$$f(1,1) = 1$$

In the previous subsection, we noted that we can add the corresponding constraint $f(1,1) = 1$ to $\text{Latin}_{2,S}$ without changing the satisfiability of the problem because a constraint from a more generally sorted problem can be mapped back to the original problem. Confusingly however, $f(1,1) = 1$ is *not* a symmetry breaking formula for $\text{Latin}_{2,S}$. Consider the (non-satisfying) interpretation for the single-sorted problem $\text{Latin}_{2,S}$ given by the following grid:

2	1
1	1

Since there is a single sort N with two values, there are exactly two SV symmetries: the identity and the transposition swapping 1 and 2. Applying the identity to this interpretation does not cause $f(1,1) = 1$ to be satisfied (as it leaves the interpretation unchanged). However, nor does applying the transposition. Since the same sort is used for the rows, columns, and entries, the SV symmetry *simultaneously* swaps the cell labels, the rows, and the columns of the grid. The resulting interpretation is the following, which does not satisfy $f(1,1) = 1$:

2	2
2	1

Therefore Definition 12 is not satisfied, and $f(1,1) = 1$ is not a symmetry breaking formula for $\mathrm{Latin}_{2,S}$.

It is important to note that this is not a contradiction. All symmetry breaking formulas must preserve satisfiability of the MSFMF problem, but formulas that preserve satisfiability are not always symmetry breaking formulas as defined by Definition 12. However, given that $\mathrm{Latin}_{2,S}$ and $\mathrm{Latin}_{2,M}$ behave the same semantically, it seems strange that the notions of symmetry and symmetry breaking do not correspond between the two problems.

These peculiarities arise from limitations of the definition of SV symmetry (Definition 9): it is for a MSFMF problem P with particular sorts and thus varies with which sorts are in P even for problems that behave the same semantically.

To resolve these issues, we define a more general notion of symmetry tied to the semantic structure. We take inspiration from the literature for **constraint satisfaction problems (CSPs)** [33] [34], by defining a more general group of symmetries, and view the SV symmetries as a subgroup. First, we define the group of permutations (which are not necessarily symmetries).

Definition 23 (Semantic Permutation)**.** For a partially-interpreted MSFMF problem, a **semantic permutation** σ is a collection containing a permutation $\sigma_f : \mathrm{Interps}(f) \to \mathrm{Interps}(f)$ for each functional or relational symbol f.

Applying a semantic permutation σ to an interpretation I produces an interpretation $\sigma \bullet I$, defined in a natural way: for each functional or relational symbol h, we define $h^{\sigma \bullet I}$ to be the result of applying σ_h to h^I .

We define the semantic symmetries to be exactly those semantic permutations that preserve whether an interpretation is a solution.

Definition 24 (Semantic Symmetry)**.** Let P be a partially-interpreted MSFMF problem. A semantic permutation σ is a **semantic symmetry** for P if for every interpretation I of P, $\sigma \bullet I$ satisfies P if and only if I satisfies P.

SV symmetries naturally induce semantic symmetries.

Definition 25 (Semantic Extension)**.** Let σ be an SV permutation of a partially-interpreted MSFMF problem. The **semantic extension** of σ is the semantic permutation σ^* defined as follows. For each functional or relational symbol h, σ^*_h is defined by $\sigma^*_h(H) = H_\sigma$ for $H \in \mathrm{Interps}(h)$.

By Definition 9, all SV symmetries preserve whether an interpretation is a solution, so their semantic extensions are semantic symmetries. If we redefine symmetry breaking formulas (Definition 12) to use semantic symmetries (rather than SV symmetries), then all of the symmetry breaking schemes we proved correct would still hold, since SV symmetries are special cases of semantic symmetries.

As a result of the relationship between their domain assignments, problems such as $\mathrm{Latin}_{n,S}$ and $\mathrm{Latin}_{n,M}$, and more generally any P_{Gen} and P with $P_{\mathrm{Gen}} \sqsubseteq P$, have the same set of interpretations for each functional and relational symbol, as well as corresponding solutions. Therefore they have the same set of semantic symmetries. Thus, for any symmetry breaking formula (based on the modified definition using semantic symmetries) for $\mathrm{Latin}_{n,M}$, its corresponding formula in $\mathrm{Latin}_{n,S}$ is a symmetry breaking formula for $\mathrm{Latin}_{n,S}$.

Semantic symmetries also explain the relationship between sort inference and symmetry. $\mathrm{Latin}_{n,M}$ contains SV symmetries that are semantic symmetries for $\mathrm{Latin}_{n,S}$, but are not SV symmetries for $\mathrm{Latin}_{n,S}$. These "hidden" symmetries make up the difference between the $n!$ SV symmetries of $\mathrm{Latin}_{n,S}$ and the $(n!)^3$ SV symmetries of $\mathrm{Latin}_{n,M}$. Sort inference *detects* these hidden symmetries, allowing for stronger symmetry breaking. In future work, we would like to explore whether there exist more hidden semantic symmetries that we can detect.

8 Empirical Evaluation

In this section, we describe our evaluation of the practical benefits of our new symmetry breaking schemes for many-sorted finite model finding. Our testing platform is an Intel®Xeon®CPU E3-1240 v5 @ 3.50 GHz with Ubuntu 16.04 64-bit with up to 64GB of user memory. The scripts we used in our evaluation are available at : https://github.com/WatForm/fortress-eval-poremba-symmetry.

To evaluate our new schemes, we implemented them in the Fortress finite model finder [18]. Fortress is a finite model finding library for many-sorted first-order logic. It is a MACE-style solver, which reduces the input MSFMF problem to a problem in the logic of equality with uninterpreted functions (EUF), and then invokes the Z3 [35] SMT solver on the EUF problem. While the problem is expressed in EUF, Fortress adds range formulas to guarantee that the solver only searches for solutions of finite size.

The existing version of Fortress uses the CSC and CSF symmetry breaking schemes, applied separately for each sort. We created two modified versions of Fortress, which handle symmetry breaking differently from the existing version as shown in Table 1 and described in the following paragraphs. Fortress with all its versions is available at: https://github.com/WatForm/fortress.

Fortress+: Our first modified version is called Fortress+. It begins symmetry breaking the same as Fortress, performing the CSC and CSF schemes where possible. After having exhausted possibilities to use CSC and CSF, Fortress+ chooses multi-sorted functions on which to apply the RDI and RDD schemes. Once these schemes are exhausted, Fortress+ then applies the ladder scheme on predicates where possible. At all times, Fortress+ tracks which values are fresh and stale. Importantly, Fortress+ performs symmetry breaking on functions in the same order as Fortress, it just may add additional symmetry breaking formulas. Because SMT solver performance is difficult to predict, the order of functions selected for symmetry breaking may have significant performance implications. Thus, for our evaluation, we use the same order of functions used in the original Fortress and simply add more symmetry breaking formulas.

Fortress+SI: The second modified version, Fortress+SI, is similar to Fortress+, but performs sort inference on the original problem for greater symmetry breaking as discussed in Section 7. To control function orderings for meaningful comparison, Fortress+SI always selects functions in the same order as Fortress+, though it may select more functions

TABLE 1
Versions of Fortress

Version	Order of Use of Symmetry Breaking Schemes
Fortress	CSC, CSF
Fortress+	CSC, CSF, RDI, RDD, ladder symmetry breaking
Fortress+SI	Sort inference then CSC, CSF, RDI, RDD, ladder symmetry breaking

TABLE 2
Comparison Ratios Per-File and Total Times Over All Files

				Per-File Time Ratios				Total (Summed) Times	
Comparison	Type	# Files	# Wins	Geom. Mean	Median	Min	Max	Absolute Times (sec)	Ratio
Fortress+ / Fortress	unsat	100	53/47	82%	100%	7%	282%	42355/48564	87%
Fortress+ / Fortress	sat	50	33/17	79%	79%	8%	551%	20743/24381	85%
Fortress+SI / Fortress+	unsat	73	32/41	97%	102%	6%	932%	33388/32405	103%
Fortress+SI / Fortress+	sat	23	9/14	100%	105%	47%	225%	7803/7685	102%
Fortress+SI / Fortress	unsat	73	31/42	91%	102%	3%	523%	33388/34085	98%
Fortress+SI / Fortress	sat	23	16/7	77%	83%	14%	204%	7803/9298	84%

afterwards if it has not exhausted all symmetry breaking opportunities.

To generate MSFOL finite model finding problems, we start with the unbounded MSFOL problems in the UF (uninterpreted functions, includes quantifiers but only free sorts) SMT-LIB benchmark repository [36]. Since these are unbounded problems, we determine a suitable finite scope for our evaluation as follows. We randomly order the files in the entire benchmark. Next, following the order of the randomly ordered list, we perform a binary search to find a scope between 5 and 30 on which all versions of Fortress did not exceed 20 minutes, but took longer than 3 minutes on Fortress (to eliminate trivially easy problems). For simplicity, all sorts in a problem are assigned the same size. If the binary search failed to find such a scope, the MSFOL problem is discarded. We continue this procedure until we have 50 satisfiable and 100 unsatisfiable MSFMF problems [4]. After selecting the MSFMF problems, we run Fortress, Fortress+, and Fortress+SI on each file three times and take the average running time for each version[5]. Though, we only run Fortress+SI on files where new sorts are detected by sort inference, as otherwise it behaves identically to Fortress+.

Table 2 shows the results of our comparison between all the methods. The last two columns report the absolute total times, summed across all files, and the ratio of these total times between the different Fortress versions. The 'Per-File' columns report the geometric mean, median, minimum, and maximum of the ratios of the running times for individual files. That is, for each individual file, the ratio of the running times between the Fortress versions is taken, and then statistics (mean, median, etc.) are computed of those per-file ratios. Numbers less than 100% indicate the method with more symmetry breaking performed better than the method with less symmetry breaking.

We make two remarks on these statistics. First, for the per-file ratios, we use the geometric mean rather than the usual arithmetic mean since it is more appropriate for time ratios - for example, ratios of 50% (twice as fast) and 200% (twice as slow) average to 100% (same speed) in the geometric mean but not the arithmetic mean. Second, we note that the per-file ratios and total time ratios each indicate important but distinct aspects of performance. For instance, suppose on one file that version A takes 50 seconds and version B takes 100 seconds, while on a second file version A takes 500 seconds and version B takes 1000 seconds. Both files would have equal per-file ratios of 50% and thus have the same impact on the per-file geometric mean, but the latter file impacts the total time ratio significantly more since the absolute times are much larger. So short tests have a larger impact on per-file results than for total time results.

Figures 1, 2, 3, 4, 5, and 6 are scatterplots, showing the individual times for every file for the six comparisons in Table 2. Points below the diagonal indicate that the method with more symmetry breaking is faster.

In comparing Fortress+ with Fortress (Figures 1 and 2), Fortress+ took 87% of the time of Fortress overall on the unsatisfiable files and is faster on 53 of the 100 unsatisfiable files. The geometric mean of the per-file ratios (which we call the "per-file mean" from now on) for unsatisfiable files is 82%, in favour of Fortress+. For the 50 satisfiable files, Fortress+ takes 85% of the total time of Fortress and is faster on 33 of the 50 files. The per-file mean for satisfiable files is 79%, again in favour of Fortress+. Thus demonstrating that our symmetry breaking schemes achieve performance improvements for both unsatisfiable and satisfiable files. Symmetry constraints always reduce the space of possible interpretations, however, they add more formulas (namely the symmetry breaking constraints) to the problem in order to limit the interpretations. For the cases where Fortress+ takes more time than Fortress, the additional formulas in the problem cause the SMT solver to take more time to produce a result. It would be useful to count solutions for

4. The repository contained many more unsatisfiable problems than satisfiable ones.

5. There was very little variation in the running times. These times are the complete running times to produce a UNSAT/SAT result from the input files, including the time to prepare the symmetry breaking constraints, carry out sort inference, transform into EUF, and invoke the SMT solver and await a result. The time is very much dominated by the SMT solving time.

all the satisfiable problems to show the reduction in the number of interpretations considered, however, the number of interpretations is too high to generate this count in a reasonable amount of time.

Next, we compare Fortress+ to Fortress+SI, which adds sort inference prior to symmetry breaking. Sort inference can reduce the number of interpretations considered significantly. For the single-sorted Latin squares problem from Section 7, using a scope of $n = 5$, we counted the number of satisfying solutions. Fortress+ found 33152 satisfying interpretations. With the same scope, Fortress+SI found only 1344 satisfying interpretations, since after sort inference it was able to produce stronger symmetry breaking constraints. However, in practice there may be a point at which too many formulas complicate the SMT solver's performance. One could add sort inference directly to Fortress (rather than to Fortress+) and compare its performance with Fortress, however, the symmetry breaking schemes of Fortress do not depend on sorts as much so it is unclear whether Fortress in its original form would benefit from sort inference.

We compare Fortress+SI to Fortress+ (Figures 3 and 4) for the 73 of the 100 unsatisfiable files and the 23 of the 50 satisfiable files where extra sorts are detected by sort inference. In this case Fortress+ has slightly better total performance on our problems sets, as Fortress+SI takes 103% of the overall time of Fortress+ for unsatisfiable problems and 102% of the overall time for satisfiable problems. The per-file means are 97% for unsatisfiable problems and 100% for satisfiable problems, the former slightly in favour of Fortress+SI. Fortress+ wins on more problems for both unsatisfiable and satisfiable problems. It is interesting that Fortress+SI does not outperform Fortress+ since Fortress+SI reduces the space of possible interpretations. However, Fortress+SI increases the number of symmetry breaking formulas added subject to the order of constants, functions, and predicates chosen. In static symmetry breaking, the formulas are added independently of the underlying solving algorithm. We hypothesize that at some point having more formulas becomes a more important factor in determining the solver time than reducing the space of possible interpretations.

Finally, we compare Fortress+SI to Fortress (Figures 5 and 6) for the 73 unsatisfiable files and the 23 satisfiable files where extra sorts are detected by sort inference. For the satisfiable problems, Fortress+SI has a clear performance improvement, with a per-file mean of 77% and median of 83%, and being faster in the overall total time at 84% the total time of Fortress. For the unsatisfiable problems, the results are less conclusive, with Fortress+SI losing more than it wins, though with a 91% per-file mean and taking 98% the total time of Fortress.

In summary, based on this set of problems, our symmetry breaking schemes reduce the solving time for both unsatisfiable and satisfiable problems. However, it is not clear that the additional symmetry breaking that comes from sort inference is worthwhile to improve performance.

9 Related Work

There are a number of solving tools that are based on finite model finding. Examples of SEM-style solvers, which use an explicit search algorithm, are Mace4 [15], SEM [24], and Falcon [26]. Examples of MACE-style solvers, which reduce the FMF problem to symbolic logic, are Paradox [16], Kodkod [22] (used in the Alloy Analyzer), and Fortress [18]. The SMT solver CVC4 [12], [13] has a hybrid algorithm where finite model finding is integrated into the logic solver. The Vampire theorem prover uses a MACE-style approach for its finite model finding [25].

SEM-style solvers apply symmetry breaking dynamically during search. For example, Zhang's Falcon model finder [26] uses the least number heuristic, tracking the maximal domain value used so far during search and only allowing a single value from the unused values to be considered when assigning values to a cell. The SEM model finder [37] and Mace4 [15] also use the least number heuristic. The FMF in CVC4 relies on the existing symmetry breaking for the EUF decision procedure in the underlying SMT solver [12].

MACE-style model finders, which reduce FMF to satisfiability over simpler logics and invoke an off-the-shelf SAT or SMT solver, use static symmetry breaking. Additional formulas are added in advance of invoking the underlying solver to prevent it from exploring redundant portions of the search space.

Crawford et al. [30] first introduced static symmetry breaking for boolean SAT. In this context, symmetries arise from interchangeable propositional variables, rather than values. An ordering is chosen on the propositional variables of interpretations, which allows each interpretation to be treated as a binary string indexed by variables. For a given symmetry σ which permutes variables, a constraint is generated which requires that the string for a solution must be (weakly) lexicographically smaller than the string obtained by applying σ to the solution. These constraints are then encoded in propositional logic and appended to the SAT problem. In general, the number of symmetries may be exponentially large, so only a subset of the symmetries is selected. This approach would later become known as Lex-Leader.

Lex-Leader can be generalized for enumerating other combinatorial objects, as done for example by Shlyakhter [38]. Symmetries of values become symmetries of bits or propositional variables when combinatorial objects are encoded as binary strings. Shlyakhter showed how to generate optimized symmetry breaking formulas for various objects, including acyclic digraphs, permutations, relations, and functions. These formulas for relations and functions are not sufficient for our purposes with MSFMF however, since they assume all input and output domains are distinct. We have schemes for all kinds of sort configurations, where sorts can be repeated across both the input and output sorts. We also consider cases where some of those sorts may have stale values so that we may combine symmetry breaking schemes. Additionally, our symmetry framework operates at the first-order logic level rather than considering lexical orderings of binary strings.

Claessen and Sörenson introduced static symmetry breaking to the context of FMF for their MACE-style solver, Paradox [16], which reduces FMF to SAT. They used a single-sorted first-order logic, and created symmetry breaking schemes for constants and functions in that context.

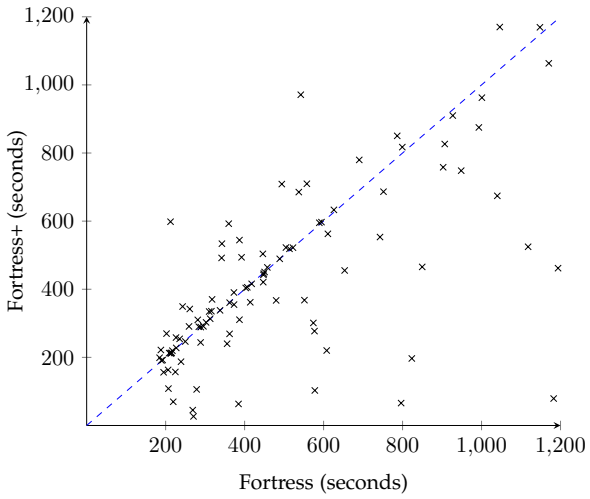

Fig. 1. Performance of Fortress vs. Fortress+ on 100 UNSAT problems

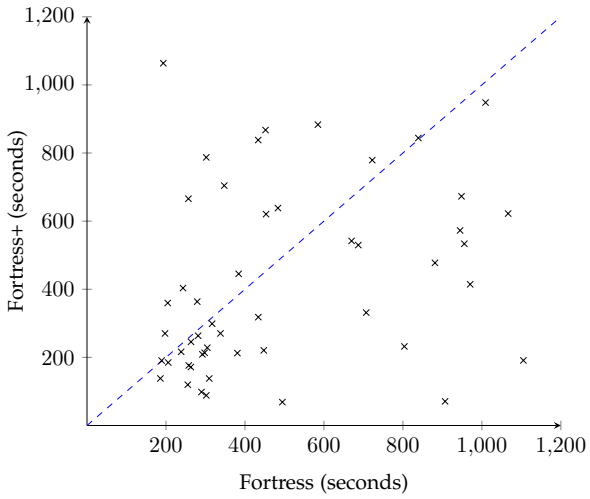

Fig. 2. Performance of Fortress vs. Fortress+ on 50 SAT problems

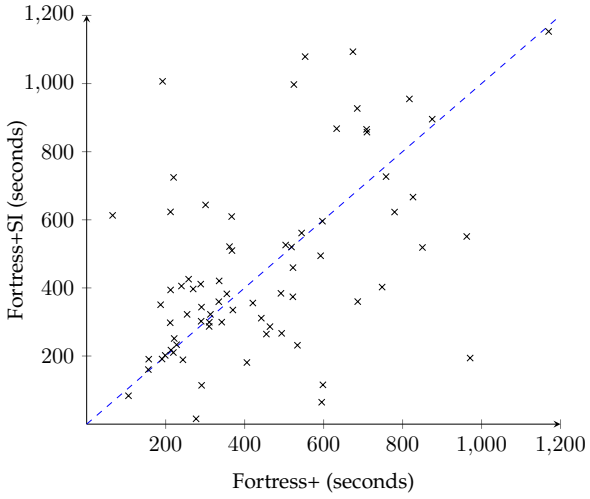

Fig. 3. Performance of Fortress+ vs. Fortress+SI on 73 UNSAT problems

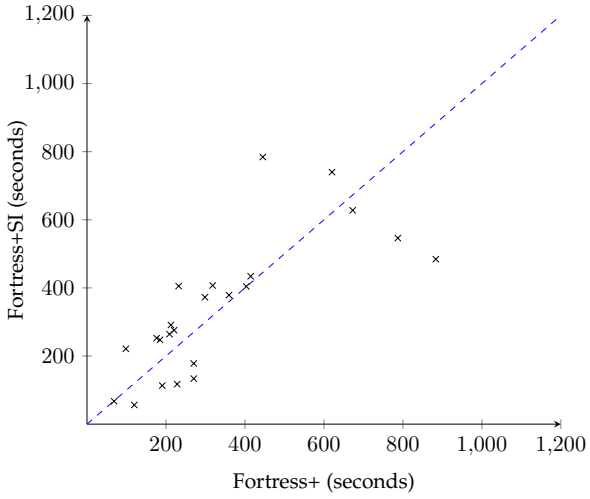

Fig. 4. Performance of Fortress+ vs. Fortress+SI on 23 SAT problems

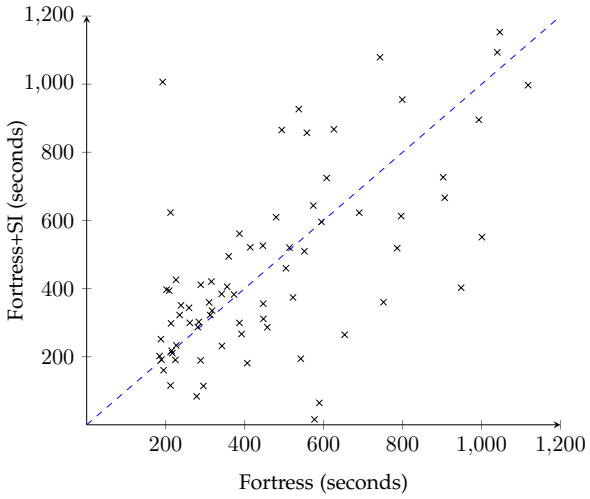

Fig. 5. Performance of Fortress vs. Fortress+SI on 73 UNSAT problems

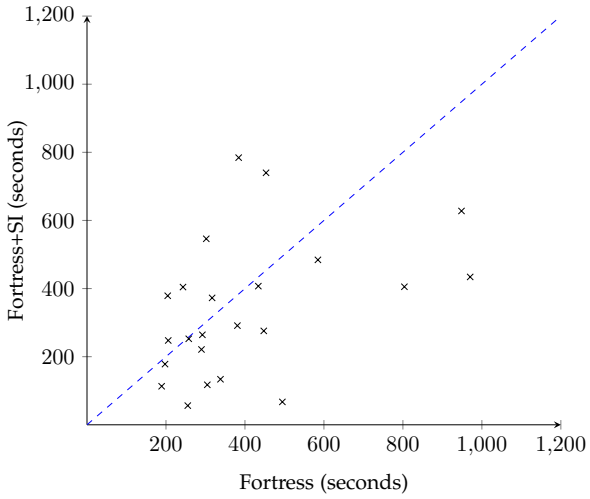

Fig. 6. Performance of Fortress vs. Fortress+SI on 23 SAT problems

These techniques are described earlier in this article. They also applied sort inference to increase the symmetry breaking obtained for their scheme for constants.

Our static symmetry breaking builds on the work of Claessen and Sörenson, but extends it with three new symmetry breaking schemes such that there is some kind of symmetry breaking applicable to every function and predicate in a pure FMF problem. We formalize the idea of partially-interpreted MSFOL to facilitate the combination of symmetry breaking schemes, which handles the problem of partial interpretations implicitly.

Reger et al. [25] use the schemes of Claessen and Sörenson in the Vampire theorem prover. Reger, Riener, and Suda [31] prove the correctness of Paradox's symmetry breaking schemes. Their technique is more general than Paradox's in that constants and functions can be intermixed for symmetry constraints more freely, however, it does not address functions of different sorts. They complete an experimental evaluation to compare heuristics for the order of applying symmetry breaking to constants and functions.

Kodkod [17] is a MACE-style model finder that reduces to SAT. Kodkod uses a relational first-order logic, where there is no special support for functions but there are rich operations on relations, such as join. Kodkod supports partial instances, where a user can specify a list of values (or value-tuples) that must be present in the interpretation of a relation. In our partially-interpreted formulas, values can appear anywhere in a formula so partial instances are a special case of our partially-interpreted formulas. We use partially-interpreted problems to decompose symmetry breaking into an iterative process to facilitate proofs of correctness and implementation, but unlike Kodkod, we assume the input problem is pure. In Section 2, we noted that the only stale values in our framework are those introduced by our own symmetry breaking formulas, which are highly asymmetric in their use of values, so the only remaining useful symmetries are those arising from permuting fresh values. Thus, concerning ourselves with stale-fixing symmetries is sufficient. Kodkod allows an input problem to include arbitrary partial instances, thus there may be symmetries arising from the interchangeability of some stale values. Kodkod attempts to detect such symmetries, then adds Lex-Leader symmetry breaking formulas.

Fortress [18] introduced a novel method for converting FMF to EUF (rather than SAT) but used the same symmetry breaking schemes as Claessen and Sörenson.

10 Conclusion

Our novel, iterative framework for combining static symmetry breaking schemes through the use of partially-interpreted MSFMF problems makes it possible to soundly combine symmetry breaking schemes and explore more symmetries. We presented and proved the correctness of three static symmetry breaking schemes for many-sorted finite model finding. The range-domain independent (RDI) and range-domain dependent (RDD) schemes extend earlier work by Claessen and Sörensson to functions of multiple sorts. The ladder scheme for relations is entirely novel and exploits symmetries in the set of elements that can satisfy a predicate. Together, these schemes cover all the kinds of functions and predicates that can appear in MSFOL to increase the amount of symmetry breaking possible for a problem. Sorts present opportunities to increase the amount of symmetry breaking. We presented a framework based on semantic symmetries to describe the hidden symmetries exposed by sort inference. Our techniques are implemented in the Fortress model finder and we demonstrated their effectiveness empirically on a large corpus of models. Our techniques may also benefit other finite model finding tools such as Kodkod and Paradox. An interesting direction for future work is to examine heuristics for the order in which to choose constants and functions for symmetry breaking constraints. We are currently investigating linking the Alloy Analyzer with the Fortress solver so that Alloy users can take advantage of our results.

Acknowledgments

We thank Khadija Tariq and Andrew Reynolds for discussions regarding this work. We also thank the editor and the reviewers for their detailed comments. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] D. Jackson, *Software abstractions: logic, language, and analysis*, revised edition ed. Cambridge, Mass: MIT Press, 2012.

[2] F. Nazerian, H. Motameni, and H. Nematzadeh, "Emergency role-based access control (E-RBAC) and analysis of model specifications with Alloy," *Journal of Information Security and Applications*, vol. 45, pp. 131–142, 2019.

[3] P. Zave, "Using Lightweight Modeling to Understand Chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 49–57, 2012.

[4] S. Shah, K. Anastasakis, and B. Bordbar, "From UML to Alloy and back again," in *Models in Software Engineering, Workshops and Symposia at MODELS*, vol. 413, 10 2009, pp. 158–171.

[5] X. Wang and A. Rutle, "Model checking healthcare workflows using Alloy," *Procedia Computer Science*, vol. 37, pp. 481–488, 2014, the 5th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2014)/ The 4th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2014)/ Affiliated Workshops.

[6] S. F. Siegel, "What's wrong with on-the-fly partial order reduction," in *Computer-Aided Verification (CAV)*, ser. Lecture Notes In Computer Science, vol. 11562. Springer, 2019, pp. 478–495.

[7] T. Dyer, A. Altuntas, and J. Baugh, "Bounded verification of sparse matrix computations," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 36–43.

[8] M. Jahanian, J. Chen, and K. Ramakrishnan, "Formal verification of interoperability between future network architectures using Alloy," in *International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, ser. International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ), vol. 12071. Springer, 2020, pp. 44–60.

[9] A. Vakili and N. A. Day, "Temporal logic model checking in Alloy," in *International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, ser. Lecture Notes In Computer Science, vol. 7316. Springer, 2012, pp. 150–163.

[10] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "The Electrum analyzer: Model checking relational first-order temporal specifications," in *Automated Software Engineering*, 2018, pp. 884–887.

[11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. T. Tinelli, "CVC4," in *Computer-Aided Verification (CAV)*. Springer, 2011, pp. 171–177.

[12] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić, "Finite Model Finding in SMT," in *Computer-Aided Verification (CAV)*, vol. 8044. Springer, 2013, pp. 640–655.

[13] A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. Barrett, "Quantifier instantiation techniques for finite model finding in smt," in *Conference on Automated Deduction (CADE)*, ser. Lecture Notes In Computer Science, vol. 7898. Springer, 2013, pp. 377–391.

[14] W. McCune, "A Davis-Putnam program and its application to finite first-order model search: Quasigroup Existence Problem," Argonne National Laboratory, Tech. Rep., 1994.

[15] H. Zhang and J. Zhang, "MACE4 and SEM: A Comparison of Finite Model Generators," in *Automated Reasoning and Mathematics*. Springer, 2013, vol. 7788, pp. 101–130.

[16] K. Claessen and N. Sörensson, "New techniques that improve MACE-style finite model finding," in *Conference on Automated Deduction (CADE)*, 2003, pp. 11–27.

[17] E. Torlak and D. Jackson, "Kodkod: A Relational Model Finder," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[18] A. Vakili and N. A. Day, "Finite Model Finding Using the Logic of Equality with Uninterpreted Functions," in *International Symposium on Formal Methods*. Springer, 2016, vol. 9995, pp. 677–693.

[19] W. McCune, "Mace4 reference manual and guide," *CoRR*, vol. cs.SC/0310055, 2003.

[20] F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Shatter: Efficient symmetry-breaking for boolean satisfiability," in *ACM/IEEE Design Automation Conference*. ACM, 2003, p. 836–839.

[21] W. Wang, M. Usman, A. Almaawi, K. Want, K. S. Meel, and S. Khurshid, "A study of symmetry breaking predicated and model counting," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes In Computer Science, vol. 12078. Springer, 2020, pp. 115–134.

[22] E. Torlak, "A constraint solver for software engineering : finding models and cores of large relational specifications," Thesis, Massachusetts Institute of Technology, 2009.

[23] C. Barrett, P. Fontaine, and A. Stump, "The SMT-LIB Standard: Version 2.6," p. 104, 2017-07-18.

[24] J. Zhang and H. Zhang, "SEM: a system for enumerating models," in *International Joint Conferences on Artificial Intelligence Organization (IJCAI)*, 1995.

[25] 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*. Springer International Publishing, 2016, vol. 9710, pp. 323–341.

[26] J. Zhang, "Constructing finite algebras with FALCON," *Journal of Automated Reasoning*, vol. 17, no. 1, Aug. 1996.

[27] N. Peltier, "A new method for automated finite model building exploiting failures and symmetries," *Journal of Logic and Computation*, vol. 8, no. 4, pp. 511–543, Aug. 1998.

[28] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli, "Computing finite models by reduction to function-free clause logic," *Journal of Applied Logic*, vol. 7, no. 1, pp. 58–74, Mar. 2009.

[29] G. Audemard and B. Benhamou, "Symmetry in finite model of first order logic," in *Workshop on Symmetry and Constraint Satisfaction Problems–Affiliated to CP (SymCon)*, 2001, pp. 01–08.

[30] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy, "Symmetry-breaking Predicates for Search Problems," in *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, ser. KR'96. Morgan Kaufmann Publishers Inc., 1996, pp. 148–159.

[31] G. Reger, M. Riener, and M. Suda, "Symmetry avoidance in MACE-style finite model finding," in *Frontiers of Combining Systems (FroCoS)*, ser. Lecture Notes In Computer Science, vol. 11715, 2019.

[32] B. C. Pierce, *Types and programming languages*. Cambridge, Mass: MIT Press, 2002.

[33] I. P. Gent, K. E. Petrie, and J.-F. Puget, "Symmetry in Constraint Programming," in *Foundations of Artificial Intelligence*. Elsevier, 2006, vol. 2, pp. 329–376.

[34] D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith, "Symmetry Definitions for Constraint Satisfaction Problems," *Constraints*, vol. 11, no. 2-3, pp. 115–137, Jul. 2006.

[35] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[36] "SMT-LIB uf benchmarks," last accessed: 2022-01-11. [Online]. Available: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/UF.git

[37] J. Zhang and H. Zhang, "System description generating models by SEM," in *Conference on Automated Deduction (CADE)*, vol. 1104. Springer, 1996, pp. 308–312.

[38] I. Shlyakhter, "Generating effective symmetry-breaking predicates for search problems," *Discrete Applied Mathematics*, pp. 1539–1548, 2007.

Joseph Poremba is a PhD student at the University of British Columbia. He received his MSc degree at the University of British Columbia in 2022 and his BMath degree at the University of Waterloo in 2020. His undergraduate research, supervised by Prof. Nancy A. Day, focused on finite model finding and symmetry breaking. His graduate research, supervised by Prof. Bruce Shepherd, focuses on combinatorial optimization.

Nancy A. Day is an Associate Professor in the David R. Cheriton School of Computer Science at the University of Waterloo in Canada. She received her PhD and MSc degrees at the University of British Columbia and her BSc at the University of Western Ontario. Her research interests include modelling languages and model-driven engineering, formal specification and automated analysis, and verification methods to ensure the safety of software-intensive systems. She is a member of the IEEE Computer Society.

Amirhossein Vakili is a software engineer focused on the design and implementation of large-scale cloud-based services. He received his PhD from University of Waterloo under the supervision of Prof. Nancy A. Day. His dissertation explored the applications of automated reasoning to model checking of temporal logics.