Finding Finite Models in Multi-Sorted First Order Logic[⋆]

Giles Reger[1], Martin Suda[1], and Andrei Voronkov[1,2,3]

[1] University of Manchester, Manchester, UK
[2] Chalmers University of Technology, Gothenburg, Sweden
[3] EasyChair

Abstract. This work extends the existing MACE-style finite model finding approach to multi-sorted first order logic. This existing approach iteratively assumes increasing domain sizes and encodes the related ground problem as a SAT problem. When moving to the multi-sorted setting each sort may have a different domain size, leading to an explosion in the search space. This paper focusses on methods to tame that search space. The key approach adds additional information to the SAT encoding to suggest which domains should be grown. Evaluation of an implementation of techniques in the Vampire theorem prover shows that they dramatically reduce the search space and that this is an effective approach to find finite models in multi-sorted first order logic.

1 Introduction

There have been a number of approaches looking at finding finite models for First Order Logic (FOL), however there has not been much work on finding such models for Multi-Sorted FOL where symbols are given *sorts*. We consider a model finding method, pioneered by MACE [12], that encodes the search as a SAT problem. We show how this method can be modified to deal directly with multi-sorted input, rather than translating the problem to the unsorted setting, which is the most common current method.

There are two main motivations for this work. Firstly, many problems are more naturally expressed in multi-sorted FOL than in unsorted FOL (although their theoretical expressive power is equivalent). Therefore, it is useful to be able to reason in this setting and translations from multi-sorted FOL to unsorted FOL often make this reasoning harder. Secondly, MACE-style model finders can use sort information to make the SAT encoding smaller. However, as we discuss below, finding finite models of multi-sorted formulas also presents significant challenges.

The MACE-style approach, later extended in the Paradox [5] work, involves selecting a domain size for the finite model, grounding the first-order problem with this domain and translating the resulting formulas into a SAT problem, which, if satisfied, gives a finite model of the selected size. Search for a finite model then involves considering iteratively larger domain sizes. In the multi-sorted setting it is necessary to consider the size of each sort separately. This can be demonstrated by the following example, which is an extension of the much used *Monkey Village* example [2, 4].

[⋆] The work of all three authors was supported by EPSRC grant EP/K032674/1. Andrei Voronkov is also supported by the Wallenberg Foundation.

Example 1 (Organised Monkey Village). Imagine a village of monkeys where each monkey owns at least two bananas. As the monkeys are well-organised, each tree contains exactly three monkeys. Monkeys are also very friendly, so they pair up to make sure they will always have a partner. We can represent this problem as follows:

$$
\begin{array}{l}
(\forall M : \mathit{monkey})(\mathsf{owns}(M,\mathsf{b}_1(M)) \wedge \mathsf{owns}(M,\mathsf{b}_2(M)) \wedge \mathsf{b}_1(M) \neq \mathsf{b}_2(M)) \\
(\forall M_1, M_2 : \mathit{monkey})(\forall B : \mathit{banana})(\mathsf{owns}(M_1,B) \wedge \mathsf{owns}(M_2,B) \rightarrow M_1 = M_2) \\
(\forall T : \mathit{tree})(\exists M_1,M_2,M_3 : \mathit{monkey})((\bigwedge_{i=1}^{3} \mathsf{sits}(M_i) = T) \wedge \mathsf{distinct}(M_1,M_2,M_3)) \\
(\forall M_1,M_2,M_3,M_4 : \mathit{monkey})(\forall T : \mathit{tree})((\bigwedge_{i=1}^{4} \mathsf{sits}(M_i) = T) \Rightarrow \neg\mathsf{distinct}(M_1,M_2,M_3,M_4)) \\
(\forall M : \mathit{monkey})(\mathsf{partner}(M) \neq M \wedge \mathsf{partner}(\mathsf{partner}(M)) = M)
\end{array}
$$

where the predicates owns associates monkeys with bananas, the functions b_1 and b_2 witness the existence of each monkey's minimum two bananas, the function sits maps monkeys to the tree that they sit in, the function partner associates a monkey with its partner, and the (meta-)predicate distinct is true if all of its arguments are distinct.

This problem requires the domain of *monkey* to be exactly three times larger than the domain of *tree*, the domain of *banana* to be at least twice as large as the domain of *monkey*, and the domain of *monkey* to be even. The main model finding effort then becomes searching for an assignment of domain sizes that satisfies this problem. Here, the smallest such assignment is $|\mathit{tree}| = 2$, $|\mathit{monkey}| = 6$, and $|\mathit{banana}| = 12$. In the general case it is necessary to try all combinations of domain sizes. In the worst case this will mean trying a number of assignments exponential in the number of sorts.

The techniques introduced in this paper tackle this issue by introducing a number of ways to constrain this search space. The main contributions can be summarised as:

1. We show how the MACE-style approach (described in Sect. 3) can be extended to the multi-sorted setting via a novel extension of the SAT encoding (Sect. 5). This encoding uses information from the SAT solver to guide the search through the space of domain size assignments.
2. We use *monotonic sorts* introduced in [4] in a new way for the multi-sorted case to reduce the search space further (Sect. 5.5).
3. We utilise the saturation-based approach for first-order logic to detect further constraints on the search space introduced by injective and surjective functions (Sect. 6).
4. We present an alternative to (1), a complementary search strategy, utilising a different SAT encoding, that only needs to expand (and never shrink) the sizes of sort domains (Sect. 8).

These ideas have been realised within the VAMPIRE theorem prover [11] and evaluated on problems taken from the TPTP and SMT-LIB benchmark suites. Our experimental evaluation (Sect. 9) shows that these techniques can be used to (i) improve finite model finding in the unsorted setting, (ii) effectively and efficiently find finite models in the multi-sorted setting, and (iii) detect cases where no finite models exist.

2 Preliminaries

Multi-Sorted First Order Logic. We consider a multi-sorted first order logic with equality. Term is either a variable, a constant, or a function symbol applied to terms. Literal is either a propositional symbol, a predicate applied to terms, an equality of two terms, or a negation of either. Function and predicate symbols are *sorted* i.e. their arguments (and the return value in the case of functions) have a unique *sort* drawn from a finite set of sorts S. We only consider well-sorted literals. There is an equality symbol per sort and equalities can only be between terms of the same sort. Formulas may use the standard notions of quantification and boolean connectives, but in this work we assume all formulas are *clausified* using standard techniques. A *clause* is a disjunction of literals where all variables are universally quantified (existentially quantified variables can be replaced by skolem functions during clausification).

SAT Solvers. The technique we present later will make use of a black-box SAT solver and we assume the reader is familiar with their general properties. We assume that a SAT solver supports *solving under assumptions* [7, 8]. This means the SAT solver can be asked to search for a model of a set of clauses N additionally satisfying a conjunction of assumption literals A and is able, in case the answer is UNSAT, to provide a subset $A_0 \subseteq A$ of those assumptions which were sufficient for the unsatisfiability proof.

3 MACE-Style Finite Model Finding in an Unsorted Setting

We describe the finite model finding procedure in a single sorted setting. This is a variation of the approach taken by Paradox [5]. The general idea is to create, for each integer $n \geq 1$, a SAT problem that is satisfiable if the problem has a finite model of size n. To find a finite model we therefore iterate the approach for domain sizes $n = 1, 2, 3, \ldots$.

3.1 *DC*-Models

Let S be a set of clauses. Let us fix an integer $n \geq 1$. Let $DC = \{c_1, \ldots, c_n\}$ be a set of distinct constants not occurring in S, we will call the elements of DC *domain constants*. We extend the language by adding the domain constants and say that an interpretation is a *DC-interpretation*, if (i) the domain of this interpretation is DC and (ii) every domain constant c_i is interpreted in it by itself. Every model of S that is also a DC-interpretation will be called a *DC-model* of S. It is not hard to argue that, if S has a model of size n, then it also has a DC-model. We say that S is n-satisfiable if it has a model of size n.

Let C be a clause. A *DC-instance* of C is a ground clause obtained by replacing every variable in C by a constant in DC. For example, if $p(x) \vee x = y$ is a clause and $n \geq 2$, then $p(c_1) \vee c_1 = c_2$ and $p(c_1) \vee c_1 = c_1$ are DC-instances, while $p(c_1) \vee c_2 = c_3$ is not a DC-instance. A clause with k different variables had exactly n^k DC-instances.

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

Let us denote by S^* the set of all DC-instances of the clauses in S. Consider an example. Let S consist of three clauses

$$p(b), \qquad f(a) \neq b, \qquad f(f(x)) = x.$$

The smallest model of S has a domain of size two. Take $n = 2$, then $DC = \{c_1, c_2\}$. By the above theorem, S has a model of size two if an only if S^* has a DC-model. The set S^* consists of four ground clauses:

$$p(b), \qquad f(a) \neq b, \qquad f(f(c_1)) = c_1, \qquad f(f(c_2)) = c_2.$$

Note that DC-models are somehow similar to Herbrand models used in logic programming and resolution theorem proving, except that they are built using (domain) constants instead of all ground terms and DC-instances instead of ground instances.

Theorem 1 is not directly applicable to encode the existence of models of size n as a SAT problem, because DC-instances can contain complex terms. We will now introduce a special kind of ground atom which contains no complex subexpressions. We call a *principal term* any term of the form $f(d_1,\ldots,d_m)$, where $m \geq 0$, f is a function symbol, which is not a domain constant, and $d_1,\ldots,d_m$ are domain constants. In our example there are four principal terms: $a, b, f(c_1), f(c_2)$. A ground atom is called *principal* if it either has the form $p(d_1,\ldots,d_m)$ where $m \geq 0$, p is a predicate symbol different from equality and $d_1,\ldots,d_m$ are domain constants or $p = d$, where p is a principal term and d a domain constant. We call a *principal literal* a principal atom or its negation.

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

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

3.2 The SAT Encoding

The main step in the reduction is to transform every non-ground clause C into an equivalent clause C' such that DC-instances of C' consist (almost) only of principal literals. We will explain what "almost" means below. This transformation is known as *flattening*.

Flattening. A literal is called *flat* if it has one of the following forms:

1. $p(x_1,\ldots,x_m)$ or $\neg p(x_1,\ldots,x_m)$, where $m \geq 0$ and p is a predicate symbol;
2. $f(x_1,\ldots,x_m) = y$ or $f(x_1,\ldots,x_m) \neq y$, where $m \geq 0$ and f is a function symbol, which is not a domain constant.
3. an equality between variables $x = y$.

Every DC-instance of a flat literal is either a principal literal (for the first two cases), or an equality $c_i = c_j$ between domain constants.

To flatten clauses in S, we first get rid of all inequalities between variables, replacing every clause of the form $x \neq y \vee C[x]$ by the equivalent clause $C[y]$. Then we repeatedly replace every clause $C[t]$, where t is not a variable and t occurs as an argument to a predicate or a function symbol, by the equivalent clause $t \neq x \vee C[x]$, where x is a fresh variable.

Our example clauses can be flattened as follows:

$$p(y) \vee b \neq y, \qquad f(y_1) = y_2 \vee a \neq y_1 \vee b \neq y_1, \qquad f(y) = x \vee f(x) \neq y$$

DC-Instances. We can now produce the DC-instances of each flattened clause $C[x_1,\ldots,x_k]$. For our running example (with $n = 2$) this produces the following ten DC-instances:

$$\begin{array}{lll}
p(c_1) \vee b \neq c_1 & f(c_1) = c_1 \vee a \neq c_1 \vee b \neq c_1 & f(c_1) = c_1 \vee f(c_1) \neq c_1 \\
p(c_2) \vee b \neq c_2 & f(c_1) = c_2 \vee a \neq c_1 \vee b \neq c_2 & f(c_1) = c_2 \vee f(c_2) \neq c_1 \\
 & f(c_2) = c_2 \vee a \neq c_2 \vee b \neq c_2 & f(c_2) = c_2 \vee f(c_2) \neq c_2 \\
 & f(c_2) = c_1 \vee a \neq c_2 \vee b \neq c_1 & f(c_2) = c_1 \vee f(c_1) \neq c_2
\end{array}$$

Note that all literals are principal. If we treat principal atoms as propositional variables, the two leftmost clauses can be satisfied by making $b \neq c_1$ and $b \neq c_2$ both true, but this violates the assumption that b should equal one of the domain constants. Additionally, the two rightmost topmost clauses can be satisfied by making $f(c_1) = c_1$ and $f(c_1) = c_2$ true but this violates the assumption that f is a function. We would like to prevent both situations. To do this we introduce additional definitions.

Functionality Definitions. For each principal term p and distinct domain constants d_1, d_2 we produce the following clause

$$p \neq d_1 \vee p \neq d_2,$$

These clauses are tautologies in first-order logic and guarantee that all function symbols are interpreted as (partial) functions.

For our running example we introduce four new definitions:

$$a \neq c_1 \vee a \neq c_2, \quad b \neq c_1 \vee b \neq c_2, \quad f(c_1) \neq c_1 \vee f(c_1) \neq c_2, \quad f(c_2) \neq c_1 \vee f(c_2) \neq c_2$$

Totality Definitions. For each principal term p we produce the following clause

$$p = c_1 \vee \ldots \vee p = c_n$$

These clauses are tautologies in first-order logic and guarantee, together with functionality axioms, that all function symbols are interpreted as total functions.

For our running example we introduce four new definitions:

$$a = c_1 \vee a = c_2, \quad b = c_1 \vee b = c_2, \quad f(c_1) = c_1 \vee f(c_1) = c_2, \quad f(c_2) = c_1 \vee f(c_2) = c_2$$

The resulting SAT clauses have a model, meaning that the original clauses have a finite model with a domain of size 2, which can be extracted from the SAT encoding.

Equalities Between Variables. Flattening can result in equalities between variables, that is, clauses of the form $C \vee x = y$. DC-instances of such clauses can have, in addition to principal literals, equalities between domain constants $d_1 = d_2$, which are not principal literals. Since we only want to deal with principal literals, we will get rid of such equalities in an obvious way: delete clauses containing tautologies $d = d$ and delete from clauses literals $d_1 = d_2$, where d_1 are distinct d_2 domain constants.

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

Theorem 3. *let S be a set of flat clauses and S' be the set of clauses obtained from S^* by removing equalities between domain constants as described above and adding all functionality and totality definitions. Then (i) all literals in S' are principal and (ii) S is n-satisfiable if and only if S' is propositionally satisfiable.*

Incrementality. In [5] the authors describe a method for incremental finite model finding i.e. keeping the contents of the SAT solver when increasing n. However, in previous experiments we discovered that the technique of *variable and clause elimination* [6] (VCE) is useful at reducing the size of the SAT problem. As this is not compatible with incremental solving, our general approach is non-incremental.

3.3 Reducing the Number of Variables

The number of instances produced is exponential in the number of variables in a flattened clause. We describe two approaches that aim to reduce this number.

Definition Introduction. This reduces the size of clauses produced by flattening. Complex ground subterms are removed from clauses by introducing definitions. For example, a clause $p(f(a,b),g(f(a,b)))$ becomes $p(e_1,e_2)$ and we introduce the definition clauses $e_1 = f(a,b)$ and $e_2 = g(e_1)$, where e_1, e_2 are new constants. One can also introduce definitions for non-ground subterms.

Clause Splitting. Clauses with k variables are split into subclauses having less than k variables each. New predicate symbols applied to the shared variables are then added to join the subclauses. For example, the clause $p(x,y) \vee q(y,z)$ with three variables is replaced by the two clauses $p(x,y) \vee s(y)$ and $\neg s(y) \vee q(y,z)$ where s is a new predicate symbol. These new clauses have two variables each. For large domain sizes splitting can drastically reduce the size of the resulting propositional problem. This was first used for finite model finding by Gandalf [17] and later in Eground [14] for EPR problems.

3.4 Symmetry Breaking

The SAT problem produced above can contain many symmetries. For example, every permutation of DC applied to a DC-model will give a DC-model, and there are $n!$ such permutations. This means that the SAT solver will be checking the same model multiple times. We can (partially) break these symmetries as follows. Firstly, if the input contains constants $a_1, \ldots, a_l$ we can add the clauses

$$a_i \neq c_m \vee a_1 = c_{m-1} \vee \ldots \vee a_{i-1} = c_{m-1}$$

for $1 < i \leq l$ and $1 < m \leq n$, where we have arbitrarily ordered the constants and captured the constraint that if the i-th constant is equal to a domain element then some earlier constant must be equal to the next smallest domain element. Secondly, we can tell the SAT solver about this order on constants by adding the clauses

$$a_i = c_1 \vee \ldots \vee a_i = c_i$$

for $i \leq \mathsf{min}(m,n)$, which captures the constraint that the i-th constant must be equal to one of the first i-th domain elements. If $1 < m < n$ then we can also use principal terms other than constants in the last clause, but not in the first.

3.5 Determining Unsatisfiability

If it is possible to detect the *maximum* domain size then it is possible to show there is no model for a formula if all domain sizes up to, and including, this maximum size have been explored. There are two straightforward ways to detect maximum domain sizes. Firstly, we can look for axioms such as $(\forall x)(x = a \vee x = b)$ and $(\forall x)(\forall y)(\forall z)(x = y \vee x = z \vee z = y)$. Both indicate that the problem has a maximum domain size of 2. Secondly, we can look for so-called EPR problems that only use constant function symbols, in this case, the domain size is bounded by the number of constants.

4 Previous Work in the Multi-Sorted Setting

We review previous work related to finite model finding for multi-sorted FOL.

Translating Sorts Away. One approach to dealing with multi-sorted FOL is to translate the sorts away. We discuss two well-known translations, see [2] for further discussions of such translations.

Sort Predicates. One can *guard* the use of sorted variables by a *sort predicate* that indicates whether a variable is of that sort. This predicate can be set to false in a model for all constants not of the appropriate sort. For example, the last formula in the Organised Monkey Village problem can be rewritten using the sort predicate isMonkey.

$$(\forall M)(\mathsf{isMonkey}(M) \rightarrow \mathsf{partner}(M) \neq M \wedge \mathsf{partner}(\mathsf{partner}(M)) = M)$$

One also needs to add additional axioms that say that sorts are non-empty and that functions return the expected sort. For the *monkey* sort we need to add

$$(\exists M)(\mathsf{isMonkey}(M)) \qquad (\forall M)(\mathsf{isMonkey}(\mathsf{partner}(M))).$$

Sort Functions or Tags. One can *tag* all values of a sort using a *sort function* for that sort. The idea is that in a model the function can map all constants (of any sort) to a constant of the given sort. For example, the last formula from the Organised Monkey Village problem can be rewritten using $\mathsf{f_m}$ as a sort function for *monkey*:

$$(\forall M)(\mathsf{f_m}(\mathsf{partner}(\mathsf{f_m}(M))) \neq \mathsf{f_m}(M) \wedge \mathsf{f_m}(\mathsf{partner}(\mathsf{f_m}(\mathsf{partner}(\mathsf{f_m}(M))))) = \mathsf{f_m}(M))$$

The authors of [2] suggest conditions that allow certain sort predicates and functions to be omitted. However, their arguments relate to resolution proofs and do not apply here.

Sorting it Out with Monotonicity. In [4] Claessen et al. introduce a monotonicity analysis and show how it can help translate multi-sorted formulas to unsorted ones by only applying the above translations to non-monotonic sorts. A sort τ is monotonic for a multi-sorted FOL formula ϕ if for any model of ϕ one can add an element to the domain of τ to produce another model of ϕ. For example, in the Organised Monkey Village example the *banana* sort is monotonic as we can add more bananas once we have enough. However, *monkey* and *tree* are not monotonic as increasing either requires more trees, monkeys and bananas.

In [4] they observe that if there is no positive equality between elements of a sort then a new domain constant can be added and made to behave like an existing domain constant and there is no way to detect this i.e. positive equalities are required to bound a sort. They refine this notion further by noting that a positive equality can be guarded by a predicate, if that predicate can be forced to be true for all new domain elements. They introduce a calculus and associated SAT encoding capturing these ideas that can be used to detect monotonic sorts, which we use in our work.

Using a Theory of Sort Cardinalities. In the single-sorted setting there is a family of techniques called SEM-style after the SEM model finder [18] based on constraint satisfaction methods. There exists a technique in this direction for the multi-sorted setting implemented in the CVC4 SMT solver [13]. The idea behind this approach is to introduce a theory of sort cardinality constraints and to incorporate this theory into the standard SMT solver structure. Briefly, this approach introduces cardinality constraints (upper bounds) for sorts and searches for a set of constraints that is consistent with the axioms. To check a cardinality constraint k for sort s, a congruence relation is built for s-terms and an attempt made to merge congruence classes so that there are at most k. Cardinality constraints are then increased if found to be inconsistent. Quantified formulas are then instantiated with representative constants from the equivalence classes.

5 A Framework for the Multi-Sorted Setting

In this section we introduce our framework able to build models of multi-sorted formulas directly, in contrast to translating the sorts away. The key challenge is dealing with a large and growing search space of domain sizes.

5.1 Using Sorts in the SAT Encoding.

The SAT encoding in Section 3 can be updated to become sort-aware. First, instead of the domain size n we use finite domain sizes n_s for every sort s. Second, instead of considering $DC = \{c_1, \ldots, c_n\}$, we consider domains for each sort $DC_s = \{c_1, \ldots, c_{n_s}\}$. We can now define n as a function (called *domain size assignment*) mapping each sort s to n_s and likewise, define DC as the function mapping each sort s to DC_s. After that we can speak about DC-models and n-satisfiability.

All the definitions for the one-sorted case are modified to respect sorts. For example, in a DC-instance of a clause a variable of a sort s can only be replaced by a domain constant in DC_s. For example, for the Organised Monkey Village problem (see page 2) we could consider the domain size assignment n such that $n_{tree} = 1$, $n_{monkey} = 2$ and $n_{banana} = 2$. The first formula in this description can be split into three clauses, the first of which would be flattened as $\mathsf{owns}(m,x) \vee \mathsf{b}_1(m) \neq x$, which would have the two principal instances $\mathsf{owns}(c_1,c_1) \vee \mathsf{b}_1(c_1) \neq c_1$ and $\mathsf{owns}(c_1,c_2) \vee \mathsf{b}_1(c_1) \neq c_2$. We can use c_1 for both monkeys and bananas here as monkeys and bananas are never compared. For this reason we can also break symmetries on a per-sort basis.

Once we have updated the SAT encoding, finite model finding can then proceed as before where we construct the SAT problem for the current domain size assignment, check for satisfiability, and then either return a model or repeat the process with an updated domain size assignment. The problem then becomes how to generate the next domain size assignment to try.

5.2 A Search Strategy

We will view the search space of domain size assignments as an infinite directed graph whose nodes are domain size assignments and the children of an assignment are all nodes that have exactly one domain size that is one larger. Thus, the number of children of every node is the number of sorts. A child of a node n having a larger domain size than n for a sort s is called the *s-child* of n. The *s-descendant* relation is the transitive closure of the s-child relation.

A *search strategy* will explore this graph node by node in such a way that a node is always visited before its children. For each node n that we visit, we can either check n-satisfiability or *ignore* this node. To decide whether a node can be ignored, we will maintain a set of *constraints*. A constraint is a predicate on domain size assignments and nodes that do not satisfy the current set of constraints will be ignored.

We will work with a queue Q of nodes and a set $\mathscr{C}$ of *constraints*. Initially, Q consists of a single node assigning 1 to all sorts and $\mathscr{C}$ is empty. We then repeat the following steps:

1. If Q is empty, return "unsatisfiable".
2. Remove the node q from the front of Q. Do nothing if q was visited before at this step. Otherwise, continue with the following steps.

3. If q satisfies all constraints in $\mathscr{C}$, perform finite model finding for q, terminating if a model is found. In variations of this algorithm considered later, we can add some constraints to $\mathscr{C}$ at this step: these constraints will be obtained by analyzing the proof of q-unsatisfiability.
4. Add to Q all children of q.

We will now introduce an important notion helping us to prevent exploring large parts of the search space. A constraint is said to have a *s-beam* property at a node n, if all s-descendants of n violate this constraint. For example, the constraint $n_s < 3$ has the s-beam property at any node q having $q_s = 2$. We can generalize this notion to more than one sort.

With this notion we can improve step 3 of the algorithm as follows:

3′ If there is a constraint in $\mathscr{C}$ having an s-beam property at the s-child n of q, if n violates this constraint, add to Q all children of q apart from n.

For example, if we have the constraint $n_s < 3$ and $q_s = 2$, this constraint will prevent us from considering the s-child n of q having $n_s = 3$.

As a small refinement, we introduce a heuristic for deciding which node in the queue to consider next, rather than processing them in the first-in-first-out order. The idea is to estimate how difficult a domain size assignment is to check and to prioritise exploration of the easier parts of the search space. Under this variation, Q is a priority queue ordered by the size (in the number of clauses) of the corresponding SAT encoding. This setup is complete, as long as this size grows strictly from a child to its parent.

5.3 Encoding the Search Problem

We now show how an extension of the SAT encoding can be used to produce constraints and therefore indicate areas of the search space that should be avoided.

For each sort s we introduce the new propositional variables too_small_s and too_large_s. The meaning is hopefully obvious. The first should be true if the domain size for s is too small, and the second should be true if it is too large.

We update the encoding as follows. The totality definition for each principal term p becomes

$$p = c_1 \vee \ldots \vee p = c_{n_s} \vee too\_small_s$$

i.e. either the principal term equals one of the domain constants or the domain is too small. DC-Instances can be similarly updated. Let C be a DC-Instance and let $\mathsf{sorts}(C)$ be the set of sorts of variables occurring in C. We replace C by

$$C \vee \bigvee_{s \in \mathsf{sorts}(C)} too\_large_s$$

i.e. either the DC-Instance holds or the domain is too large.

We then attempt to solve the updated SAT problem under the assumptions

$$A = \bigwedge_{s \in S} (\neg(too\_small_s) \wedge \neg(too\_large_s))$$

i.e. we assume that we are using the correct domain sizes. This effectively turns the new set of constraints into those used before the addition of the extra assumptions.

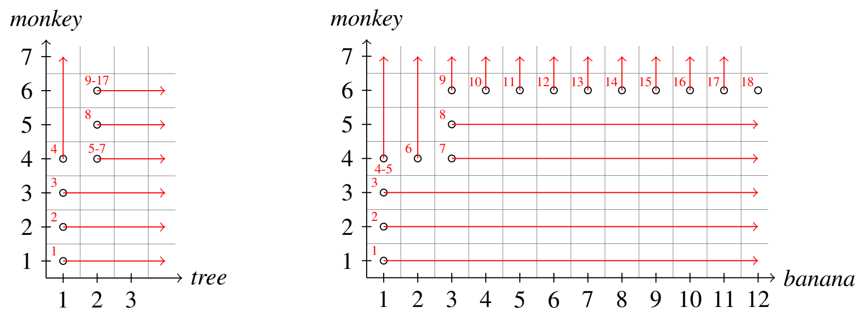

Fig. 1. Finding a finite model for the Organised Monkey Village problem.

However, as previously explained, solving under assumptions returns a subset $A_0 \subseteq A$ of the assumptions that were sufficient to establish unsatisfiability. This means that P and A_0 are inconsistent where P is the encoding of the problem. Furthermore, we can note that there is a subset of P' of P that does not use the assumptions not in A_0 such that P' and A_0 are inconsistent.

We make the following observation. The unsatisfiability proof using A_0 can be replayed for any node that includes P' in its SAT encoding. When is this the case? Exactly when it does not change the size of the domains for any of the sorts mentioned in A_0. This can then suggest which sorts need to be updated. Indeed, $\neg A_0$ can be used directly as a constraint.

For each sort there are three possible cases to consider:

1. Neither $\neg \mathit{too\_large}_s$ nor $\neg \mathit{too\_small}_s$ appear in A_0 i.e. the lack of model is independent of the current size of s
2. The conjunction $\neg \mathit{too\_large}_s \wedge \neg \mathit{too\_small}_s$ appears in A_0 i.e. the domain of s is either too large or too small. This means that to avoid repeating the above unsatisfiability we can change the size of s
3. Only $\neg \mathit{too\_large}_s$ appears in A_0 i.e. the domain of s is too large (recall this is an assumption that was inconsistent). This means that we did not need the totality definitions to establish unsatisfiability.

Each of these case allows us to produce some constraints, including those having s-beam properties. For example, the last case can give constraints like $n_s < 3$ use as an example above. We leave the full description of how the constraints are generated to a full version of this paper.

5.4 An Example.

Let us consider the Organised Monkey Village example (page 2). Running the initial search strategy on this problem requires checking 2,661 different domain sort assignments. Using the encoding described above means that only 18 assignments are tried. The search carried out by our approach is illustrated in Fig. 1. We give two projections of the 3-dimensional search space and the arrows show the parts of the search space ruled out by s-beam constraints. On each step the constraints rule out all but one neighbour, meaning that we take a direct path to the solution through the search space.

5.5 Using Monotonicity.

In our framework we can use the notion of monotonicity (see Sect. 4) in two ways.

- *Collapsing Monotonic Sorts.* All monotonic sorts can be collapsed into a single sort as this sort can grow to the size of the largest monotonic sort. It is never safe to collapse a monotonic sort into a non-monotonic one as the monotonic sort may depend on the non-monotonic one. For example, whilst *banana* is a monotonic sort it must always be twice as large as the non-monotonic sort *monkey*.
- *Refining the Search Encoding.* If a sort s is monotonic then if no model exists for domain size d_s then no model can exist where n_s is smaller. This allows us to generalise some constraints produced in the earlier search encoding. Whenever we have $\neg too\_large_s \wedge \neg too\_small_s$ in A_0 we can reduce this to $\neg too\_small_s$.

6 Detecting Constraints Between Sorts

In this section we discuss how properties of functions between sorts can be used to further constrain the domain size search space. Consider the following set of formulas

$$\begin{array}{ll} \mathsf{distinct}(a_1,a_2,a_3,a_4,a_5) & (\forall x:\tau_1)(f(x) \neq b) \\ (\forall x,y:\tau_1)(f(x)=f(y) \rightarrow x=y) & (\forall x,y:\tau_2)(g(x)=g(y) \rightarrow x=y) \end{array}$$

where a_i are constants of sort τ_1, b is a constant of sort τ_2, $f:\tau_1 \rightarrow \tau_2$, and $g:\tau_2 \rightarrow \tau_3$. The previous approach would try increasing the size of each sort by 1, discovering that one sort must grow at each step. However, we can see from the bottom two formulas that f and g are injective and therefore that $|\tau_1| \leq |\tau_2| \leq |\tau_3|$, furthermore, the second formula tells us that f is non-surjective and therefore that $|\tau_1| < |\tau_2|$. Using these constraints we can immediately discount 9 of the 15 domain size assignments considered without them.

To find constraints between the sizes of sorts τ_1 and τ_2 we look for four cases:

1. If a function $f:\tau_1 \rightarrow \tau_2$ is injective, then $|\tau_1| \leq |\tau_2|$.
2. If a function $f:\tau_1 \rightarrow \tau_2$ is injective and non-surjective, then $|\tau_1| < |\tau_2|$.
3. If a function $f:\tau_1 \rightarrow \tau_2$ is surjective, then $|\tau_1| \geq |\tau_2|$.
4. If a function $f:\tau_1 \rightarrow \tau_2$ is surjective and non-injective, then $|\tau_1| > |\tau_2|$.

These constraints can be added to the set of constraints used in the search described in Section 5. In rare cases we may learn that $|\tau_1| \geq |\tau_2|$ and $|\tau_1| \leq |\tau_2|$ i.e. that the sorts are of equal size.

Our method for detecting bounds was inspired by Infinox [3], a method for showing no finite model can exist for unsorted FOL formulas if there is a strict bound within a sort. We detect bounds by attempting to prove properties of functions.

Unary functions. For a unary function $f:\tau_1 \rightarrow \tau_2$ occurring in the problem we can simply make a claim such as

$$(\forall x:\tau_1)(\forall y:\tau_2)(f(x)=f(y) \rightarrow x=y) \wedge (\exists y:\tau_2)(\forall x:\tau_1)(f(x) \neq y)$$

for each case (this is case (2) above), and then ask whether this claim follows from the axioms of the input problem. We discuss how we do this below.

Non-unary functions. We can make similar claims for every function of arity $n \geq 2$ occurring in the problem. For example, for the binary function $g : \tau_1 \times \tau_2 \rightarrow \tau_3$ our above claim for case (2) is modified to

$$(\exists z : \tau_2)((\forall x : \tau_1)(\forall y : \tau_3)(g(z,x) = g(z,y) \rightarrow x = y) \wedge (\exists y : \tau_3)(\forall x : \tau_1)(g(z,x) \neq y))$$

i.e. we existentially quantify over one of the arguments of g. This existential quantification must cover both parts of the claim to ensure the same function is used.

Using Saturation. We can generate a set of claims in the manner described above and check, for each claim C whether $A \models C$, where A are the input axioms, using standard techniques. However, this could become very expensive. Instead we search for all claims that hold in one process. We do this by labelling a claim C as $C \rightarrow l$ where l is a fresh propositional symbol. We then use saturation-based techniques to derive consequences of these labelled claims and A and detect all unit clauses consisting of labels only[4]. Such unit clauses tell us which claims follow from A. A similar technique is used in the consequence elimination mode of Vampire [9].

7 Getting More Sorts

Previously we have seen how sort information can be used to reduce the size of the SAT encoding by only growing the domain sizes of sorts that need to be grown. In this section we describe the technique first described in [4] for inferring new sorts and how these new sorts can be useful.

Inferring Subsorts. Consider the Organised Monkey Village example. The *monkey* sort can be split into three separate subsorts as the problem can be split into three parts (assigning bananas to monkeys, assigning monkeys to trees and assigning monkeys to their partners) where the signatures do not overlap. Abstractly, we can use different monkeys in these different places as they do not interact - later we will see why this is useful. To infer such subsorts we can use the standard union-find method on positions in the signature.

Using Inferred Subsorts. Claessen et al. [4] describe two ways these inferred subsorts can be used:

– *Removing Instances.* If a subsort τ is monotonic and all function symbols with the return sort τ are constants, then we can bound the subsort by the number of constants. It is easy to argue that any ground clauses (instances, totality or functionality) for a domain constant larger than the bound of the subsort can be omitted as they will necessarily be equivalent to an existing clause. This helps reduce the size of the SAT encoding.

– *Symmetry breaking.* For the same reasons that symmetry breaking can occur per sort, symmetry breaking can now occur per inferred subsort. This is safe due to the above observation that values for different subsorts will never be compared.

Making Subsorts Proper Sorts. It is possible to promote an inferred subsort so that it is treated as a proper sort, but only if that subsort is monotonic and only if we do not allow the subsort to grow larger than its parent sort. These constraints are based on the observation that in the intedended model the subsort must have the same size as its

[4] As a technical aside, we must also ensure that the labels are very small in the term ordering used by the saturation-based method. This ensures that the claim is discharged before the label.

parent. Only monotonic subsorts can grow more slowly than their parents as they can be safely extended to be the same size by adding new elements.

To understand whether this approach can be useful consider the FOL formula

$$\mathsf{distinct}(a_1,\dots,a_{50}) \wedge (\forall x)(f(f(f(f(f(f(f(f(f(f(x)))))))))) \neq x)$$

which has an overall finite model size of 50. There are two subsorts, that of the constants a_1 to a_{50} and that of f. The flattening of the definition of f contains many variables but its subsort is monotonic and does not need to grow beyond size 3. Making this subsort a proper sort means that the SAT encoding for the check that finds the model contains 125,236 clauses, considerably fewer than the 1,187,577 clauses required otherwise.

To ensure that a promoted sort does not grow larger than its parent we add constraints to the search strategy in the same way as for sort bounds detected previously.

8 An Alternative Growing Search

The previous search strategy considers each domain size assignment separately (we therefore refer to it as a *pointwise* encoding). However, we can modify the encoding so that it captures the current assignment *and all smaller ones* at the same time. Thus we no longer talk of a domain size but rather of a domain size upper *bound*, as the parameter of the encoding. These bounds never need to shrink and thus grow monotonically for each sort. We call this encoding a *contour* encoding as we can think of it drawing a contour around the explored part and growing this outwards.

This alternative encoding works as follows. For each sort s with domain size bound n_s we introduce n_s propositional variables $bound_s(1)$ to $bound_s(n)$. Then instead of single totality constraint for each function we introduce all totality constraints for domain sizes up to n_s guarded by the appropriate bound i.e.

$$a = c_1 \vee bound_s(1), \quad \dots, \quad a = c_1 \vee \ldots \vee a = c_{n_s} \vee bound_s(n)$$

We will guard principal instances of these clauses with negations of these guards in the following way. For each sort s let s_{max} be the index of the largest domain constant in in this instance used to replace a variable of sort s. Then if s_{max} is defined, i.e. there is at least one such variable, and $s_{max} > 1$ we guard the instance with a literal $\neg bound_s(max-1)$. For example, given a function symbol $f : \tau_1 \to \tau_2$, a constant $b : \tau_2$, and a flattened clause $f(x) \neq y \vee b \neq y$, its principal instance $f(c_3) \neq c_1 \vee b \neq c_1$ would be guarded as $f(c_3) \neq c_1 \vee b \neq c_1 \vee \neg bound_{\tau_1}(2)$.

In this encoding the SAT solver can satisfy the clauses for a domain size smaller than n_s i.e. if it can satisfy a stricter totality constraint then it can effectively ignore some of the instances. As a further variation, if a sort is monotonic then we do not need to consider the possibility that a sort is smaller than its current bound. Therefore, we only need the largest totality constraint and do not need constraints on instances.

In a similar way as before, we solve the problem under the assumptions that the sort sizes are big enough i.e.

$$A = \bigwedge_{s \in S} \neg bound_s(n_s).$$

If this is shown unsatisfiable the subset of assumptions A_0 will suggest the sorts that could be grown; growing a sort not mentioned in A_0 would allow the same proof of

Table 1. Experimental Results for Unsorted problems.

			VAMPIRE		
	CVC4	Paradox	Ignore	Use	Expand
FOF+CNF: sat	1181	1444	1421	1463	**1503**
FOF+CNF: unsat	-	-	1400	1604	**1628**

unsatisfiability to be produced. We can either arbitrarily select a sort to grow, or use the heuristic described earlier to pick the smallest SAT problem of the available ones. This approach is significantly different from the previous approach as now we only consider one next domain size assignment. However, the SAT problems may be considerably harder to solve as the SAT solver is now considering a much larger set of models. In essence, each new SAT problem contains all the previous ones as sub-problems.

Finally, if the SAT problem is satisfiable then the size of a sort is given by the smallest totality constraint that is turned on, therefore by i where $bound_s(i)$ is the smallest constraint that is false.

9 Experimental Evaluation

In this section we evaluate the different techniques for finite model finding in multi-sorted FOL described in this paper and compare our approach to other tools.

Experimental Setup. We consider two sets of problems. From the TPTP [16] library (version 6.3.0) we take unsorted problems in the FOF or CNF format (we also experimented with TF0 problems without arithmetic but the results were uninteresting). From the SMT-LIB library [1] we take problems from the UF (Uninterpreted Functions) logic. Experiments were run on the StarExec cluster [15], whose nodes are equipped with Intel Xeon 2.4GHz processors and 128 GB of memory. Each experiment was run for 60 seconds. On satisfiable problems we compare with version 3.0 of Paradox [5] and version 1.4 of CVC4 [13]; Paradox does not establish unsatisfiability and CVC4 runs more than a finite model finding approach, making a comparison on unsatisfiable problems difficult. We will also include iProver [10] results in the final version of this paper. The techniques described in this paper were implemented in VAMPIRE.

Adding Sorts to Unsorted Problems. Our first experiment considers the effect of sort inference on unsorted problems. We consider three settings: (i) inferred subsorts are ignored, (ii) inferred subsorts are used to reduce the problem size and break symmetries only, and (iii) inferred subsorts are expanded to proper sorts where possible. Table 1 presents the results. This shows that sort information can be used to solve more problems. For satisfiable problems the best VAMPIRE strategy solves more problems than either CVC4 or Paradox. For both satisfiable and unsatisfiable problems, expanding subsorts into proper sorts and treating the problems as multi-sorted problems helps solve the most problems. We note that 4 problems found unsatisfiable using this approach could not be solved by any other technique in VAMPIRE, this is significant as VAMPIRE is one of the best theorem provers available for such problems.

Removing Sorts from Sorted Problems. Next we consider the translation techniques described in Section 4 applied to multi-sorted problems. Table 2 shows the results of running variations of these translations on the multi-sorted UF problems described

Table 2. Experimental Results for Translations from Multi-Sorted to Unsorted.

	Sort Predicates				Sort Functions			
	Plain	Monotonicity	Subsorts	Both	Plain	Monotonicity	Subsorts	Both
UF: sat	813	810	872	**874**	710	771	834	873
UF: unsat	101	112	221	**232**	67	67	171	171

Table 3. Experimental Results for Multi-Sorted problems.

	CVC4	Pointwise				Contour			Without
		Default	Expand	Collapse	Bounds	Default	Collapse	Bounds	Constraints
UF: sat	764 (8)	795	789	**901 (12)**	810	886 (3)	899 (1)	886 (1)	154
UF: unsat	-	212	215	241	218	270	261	267	66

above. Plain applies the translation to the whole problem, ignoring subsorts in the result. With Monotonicity only non-monotonic sorts are translated and with Subsorts the resulting problem is solved using inferred subsorts. Both adds both variations.

These results show that, for these problems, sort predicates are more useful and that the techniques of monotonicity detection and subsort inference are useful in improving the translation and reasoning with it.

Finding Models of Multi-sorted Problems. Finally, we consider our framework for reasoning with multi-sorted problems directly. Table 3 gives the results for CVC4 and eight variations of the techniques presented in this paper (we use only CVC4 since Paradox does not work on sorted problems). At the top level these are split into the Pointwise and Contour encodings and a version where no constraints were added. Then Expand refers to subsort expansion (Sect 7), Collapse refers to collapsing monotonic sorts together (Sect. 5.5), and Bounds refers to sort bound extraction (Sect. 6). These results can also be compared to Table 2 as the problems are the same.

The three main conclusions from this information are (i) overall the approach taken in this paper is able to solve more problems than the approach taken by CVC4, (ii) collapsing monotonic sorts is very useful, and (iii) including the search problem as part of the SAT encoding is vital. Bracketed numbers show unique problems solved by an approach, this shows that although CVC4 solves fewer problems it does solve some uniquely. The contour encoding was generally more successful, however in UF there are 15 and 19 problems that are only solvable using the pointwise and contour encodings respectively. As a further point, we note that the heuristic introduced on page 9 is useful, without it the default pointwise approach solved 61 fewer problems. Finally, comparing with the results in Table 2, we see that finding models for multi-sorted problems directly performs better than translating the problem to an unsorted one.

10 Conclusions and Further Work

We have introduced a new framework for MACE-style finite model finding for multi-sorted first order logic. This involved two complementary SAT encodings that capture the search for a satisfying domain size assignment and techniques aimed at decreasing the size of this search space. We have demonstrated experimentally that these techniques are effective at improving finite model finding in the unsorted setting and finding finite models for multi-sorted first order formulas. Further work will consider possible extensions to uninterpreted sorts and infinite, but finitely representable, models.

References

1. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
2. J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, 2013.
3. K. Claessen and A. Lillieström. Automated inference of finite unsatisfiability. *J. Autom. Reasoning*, 47(2):111–132, 2011.
4. K. Claessen, A. Lillieström, and N. Smallbone. Sort it out with monotonicity - translating between many-sorted and unsorted first-order logic. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pp. 207–221, 2011.
5. K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *CADE-19 Workshop: Model Computation - Principles, Algorithms and Applications*, 2003.
6. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, pp. 61–75, 2005.
7. N. Eén and N. Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pp. 502–518, 2003.
8. N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.
9. K. Hoder, L. Kovács, and A. Voronkov. Case studies on invariant generation using a saturation theorem prover. In *MICAI 2011 (Part I)*, vol. 7094 of *LNCS*, pp. 1–15, 2011.
10. K. Korovin. iProver An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Automated Reasoning*, vol. 5195 of *Lecture Notes in Computer Science*, pp. 292–298. Springer Berlin Heidelberg, 2008.
11. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35, 2013.
12. W. Mccune. A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical report, Argonne National Laboratory,, 1994.
13. A. Reynolds, C. Tinelli, A. Goel, and S. Krstic. Finite model finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pp. 640–655, 2013.
14. S. Schulz. A comparison of different techniques for grounding near-propositional CNF formulae. In *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, May 14-16, 2002, Pensacola Beach, Florida, USA*, pp. 72–76, 2002.
15. A. Stump, G. Sutcliffe, and C. Tinelli. StarExec, a cross community logic solving service. `https://www.starexec.org`, 2012.
16. G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
17. T. Tammet. Gandalf. *J. Autom. Reasoning*, 18(2):199–204, 1997.
18. J. Zhang and H. Zhang. SEM: a system for enumerating models. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pp. 298–303, 1995.