New Techniques that Improve MACE-style Finite Model Finding

Koen Claessen and Niklas Sörensson
{koen,nik}@cs.chalmers.se

Chalmers University of Technology and Göteborg University

Abstract. We describe a new method for finding finite models of unsorted first-order logic clause sets. The method is a MACE-style method, i.e. it "flattens" the first-order clauses, and for increasing model sizes, instantiates the resulting clauses into propositional clauses which are consecutively solved by a SAT-solver. We enhance the standard method by using 4 novel techniques: *term definitions*, which reduce the number of variables in flattened clauses, *incremental SAT*, which enables reuse of search information between consecutive model sizes, *static symmetry reduction*, which reduces the number of isomorphic models by adding extra constraints to the SAT problem, and *sort inference*, which allows the symmetry reduction to be applied at a finer grain. All techniques have been implemented in a new model finder, called Paradox, with very promising results.

1 Introduction

There exist many methods for finding finite models of First Order Logic (FOL) clause sets. The two most successful styles of methods are usually called *MACE-style* methods, named after McCune's tool MACE [7], and the *SEM-style* methods, named after Zhang and Zhang's tool SEM [5].

A MACE-style method transforms the FOL clause set and a domain size into a propositional logic clause set by introducing propositional variables representing the function and predicate tables, and consecutively flattening and instantiating the clauses in the clause set. A propositional logic theorem prover, also called SAT solver, is then used to attack the resulting problem. Apart from MACE itself, Gandalf [13] makes use of a MACE-style method, and there are reports of usages of SATO in a similar way [4].

A SEM-style method performs a search directly on the problem without converting it into a simpler logic. A basic backtracking search backed up by powerful constraint propagation methods, mainly based on exploiting equality, are used to search for interpretations of the function and predicate tables. A principle called *symmetry reduction* is used to avoid searching for isomorphic models multiple times. Apart from SEM itself, the tools FINDER [11], and ICGNS [8] are SEM-style methods, and again Gandalf also makes use of SEM-style methods. SEM-style methods are known to perform well on equational problems, and MACE-style methods are supposed be more all-round.

In this paper, we develop a collection of new techniques for finite model generation for unsorted first-order logic. These techniques improve upon the basic idea behind MACE dramatically. We have implemented the techniques in a new model finder called Paradox. The main novel contributions in the paper are the following.

- The main problem in MACE-style methods is the clause instantiation, which is exponential in the number of first-order variables in a clause. A well-known variable reduction technique is *non-ground clause splitting*, for which however only exponential or incomplete algorithms are known. We have devised a new

polynomial heuristic for clause splitting. Moreover, we have come up with a completely new variable reduction technique, called *term definitions*.
- The search for a model goes through consecutive stages of increasing domain sizes. In current-day algorithms, there is hardly any coupling between the search for models of different sizes. We make use of an *incremental satisfiability checker* in order to reuse information about the failed search of a model of size s in the search of a model of size $s + 1$.
- SEM-style model finders make use of techniques like the *least number heuristic* [5] and the *extended least number heuristic* [2] in order to reduce the symmetries in the search problem. To our knowledge, wTTe are the first to adapt a similar technique in a MACE-style framework. Our contribution here is that when using a SAT-solver, we must apply the symmetry reduction *statically*, i.e. by adding extra constraints, whereas SEM-style methods can apply this *dynamically*, i.e. during the search.
- It is well-known that *sort information* can help the search for models. However, we are working with unsorted problems, and therefore need to create unsorted models. We have developed a *sort inference* algorithm, which automatically finds appropriate sort information for unsorted problems. This sort information can then be used in several ways to reduce the complexity of the model search problem, while still searching for an unsorted model.

The rest of the paper is organized as follows. In Section 2 we introduce some notation. In Section 3 we describe the basic ideas behind MACE-style methods. In Sections 4, 5, 6, and 7, we introduce the four techniques: *clause splitting and term definitions*, *incremental search*, *static symmetry reduction*, and *sort inference*. In Section 8 we discuss some experimental results. Sections 9 and 10 discuss related work and conclusions.

2 Notation

In this section, we introduce some standard notation we use in the rest of the paper. We use the set $\mathcal{N}$ to stand for the set of natural numbers $\{0, 1, 2, \ldots\}$. The set $\mathcal{B}$ is the set of booleans $\{\mathsf{false}, \mathsf{true}\}$.

Terms, literal and clauses A term t is built from function symbols $\mathsf{f}, \mathsf{g}, \mathsf{h}$, constants $\mathsf{a}, \mathsf{b}, \mathsf{c}$, and variables $\mathsf{x}, \mathsf{y}, \mathsf{z}$. Each function symbol f has a single arity $\mathsf{ar}(\mathsf{f}) : \mathcal{N}$ which should be respected when building terms. We will merely regard constants as nullary function symbols.

An atom A is a predicate symbol $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \mathsf{S}$ applied to a number of terms. Each predicate symbol P has a single arity $\mathsf{ar}(\mathsf{P}) : \mathcal{N}$. There exists one special predicate symbol $=$, with arity 2, representing equality, whose atoms are written $t_1 = t_2$.

A literal is a positive or negative occurence of an atom. Negative literals are written using $\neg A$, and negative equalities are written $t_1 \neq t_2$.

A clause C is a finite set of literals, intended to be used disjunctively. We write $FV(C)$ for the set of variables in a clause C.

Interpretations An interpretation $\mathcal{I}$ consists of a non-empty set D (the domain), plus for each function symbol f a function $\mathcal{I}(\mathsf{f}) : D^{\mathsf{ar}(\mathsf{f})} \to D$, and for each predicate symbol P a function $\mathcal{I}(\mathsf{P}) : D^{\mathsf{ar}(\mathsf{P})} \to \mathcal{B}$, where we require $\mathcal{I}(=)(d_1, d_2) = \mathsf{true}$ exactly when d_1 is the same domain element as d_2, and false otherwise. An interpretation is called *finite* if its domain D is a finite set.

3 MACE-style Model Finding

This section describes shortly what the basic idea behind MACE-style model finding methods is. The description differs slightly from earlier presentations [7].

Finite domains The following observation about models is well-known. Given an interpretation $\mathcal{I}$ with a domain D satisfying a clause set S. Given a set D' and a bijection $\pi : D \leftrightarrow D'$, we construct a new interpretation $\mathcal{I}'$ with domain D' in the following way:

$$\mathcal{I}'(\mathsf{P})(x_1, \ldots, x_m) = \mathcal{I}(\mathsf{P})(\pi^{-1}(x_1), \ldots, \pi^{-1}(x_m))$$
$$\mathcal{I}'(\mathsf{f})(x_1, \ldots, x_n) = \pi(\mathcal{I}(\mathsf{f})(\pi^{-1}(x_1), \ldots, \pi^{-1}(x_n))$$

Now, $\mathcal{I}'$ also satisfies S. We call two models $\mathcal{I}$ and $\mathcal{I}'$ in the above relationship *isomorphic*. The observation implies that in order to find (finite) models, only the size s of the domain matters, and not the actual elements of the domain. Therefore, we arbitrarily choose D to be $\{1, 2, \ldots, s\}$. A special case of the observation, which we will use later, arises when $D' = D$, and π simply corresponds to a permutation of D.

Propositional encoding We are going to encode the model finding problem for a particular set of FOL clauses using propositional variables. For each predicate symbol P, and each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{P})})$ (with each $d_i \in D$), we introduce a propositional variable $\mathsf{P}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{P})})$ representing the case when $\mathcal{I}(\mathsf{P})(d_1, \ldots, d_{\mathsf{ar}(\mathsf{P})})$ is true. Also, for each function symbol f, for each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$, and for each domain element d, we introduce a propositional variable $\mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) = d$ representing the case when $\mathcal{I}(\mathsf{f})(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$ is d. We stress that these propositional variable names are merely syntactic constructs and have no meaning without context.

Flattening In order to create the necessary propositional constraints on the above variables, the first step is to transform the general FOL clauses into clauses only containing *shallow* literals, a process called *flattening*.

Definition 1 (Shallow Literals). *A literal is* shallow *iff it has one the following forms:*

1. $\mathsf{P}(\mathsf{x}_1, \ldots, \mathsf{x}_m)$, *or* $\neg\mathsf{P}(\mathsf{x}_1, \ldots, \mathsf{x}_m)$,
2. $\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n) = \mathsf{y}$, *or* $\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n) \neq \mathsf{y}$,
3. $\mathsf{x} = \mathsf{y}$.

There are two cases when a given literal is not shallow: (1) it contains at least one subterm t which is not a variable, but should be; (2) the literal is of the form $\mathsf{x} \neq \mathsf{y}$. In case (1), we can lift out an offending term t out of any literal occurring in a clause C by applying the following sequence of rewrite steps:

$$\begin{aligned} C[t] &\longrightarrow \mathsf{let}\ \mathsf{x} = t\ \mathsf{in}\ C[\mathsf{x}] && (\mathsf{x}\ \text{not in}\ C) \\ &\longrightarrow \forall \mathsf{x}.\, [\mathsf{x} = t\ \Rightarrow\ C[\mathsf{x}]] \\ &\longrightarrow t \neq \mathsf{x} \vee C[\mathsf{x}] \end{aligned}$$

In the second step in the above we make use of a standard representation of let-definitions in terms of universal quantification and implication. If t occurs more than once in C, we introduce only one variable x for t, and replace all occurences of t by x. In case (2), we apply the following rewrite rule:

$$C[\mathsf{x}, \mathsf{y}] \vee \mathsf{x} \neq \mathsf{y} \quad \longrightarrow \quad C[\mathsf{x}, \mathsf{x}]$$

If we apply the above transformations repeatedly, in the end all literals will be be shallow literals.

Example 1. Take the unit clause $\{ \mathsf{P}(\mathsf{a}, \mathsf{f}(\mathsf{x})) \}$. After flattening, the clause looks as follows:

$$\{ \mathsf{a} \neq \mathsf{y},\ \mathsf{f}(\mathsf{x}) \neq \mathsf{z},\ \mathsf{P}(\mathsf{y}, \mathsf{z}) \}.$$

Instantiating The final step is to generate propositional clauses from the flattened clauses. We generate three sets of propositional clauses:

- **Instances** For each flattened clause C, and for each substitution $\sigma : FV(C) \rightarrow D$, we generate the propositional clause $C\sigma$. (Recall that shallow literals instantiated with domain elements function as propositional literals.) In the result of the substitution, we immediately simplify all literals of the form $d_1 = d_2$ (whose value is known at instantiation time), leading to either removal of the whole clause in question (when d_1 and d_2 are equal), or simply removal of the equality literal (when d_1 and d_2 are not equal).

- **Functional definitions** For each function symbol f, for each $d, d' \in D, d \neq d'$, and for each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$, we introduce the propositional clause $\{ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) \neq d,\ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) \neq d' \}$, representing the fact that a function can not return two different values for the same arguments.

- **Totality definitions** For each function symbol f, and for each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$, we introduce the propositional clause $\{ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) = 1,\ \ldots,\ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) = s \}$, representing the fact that a function must return at least one value for each argument.

If we can find a propositional model that satisfies all of the above clauses, we have found a finite model satisfying the original set of FOL clauses. We use a SAT solver to find the actual propositional model. The FOL model can be easily built by using the propositional encoding described earlier in this section.

4 Reducing Variables in Clauses

The number of instances of each clause that is needed, is exponential in the number variables the clause contains. In general, if a clause contains k variables, the number of instances that will be needed for domain size s are s^k. Moreover, this property is made worse by the fact that term flattening introduces many auxiliary variables (something that a SEM-style model finder does not do). In this section we describe how to remedy this situation.

Splitting Splitting is a well known method [10, 13, 1] that can be used to replace one clause by several other clauses *each containing fewer variables* than the original clause. The following is an example of *non-ground splitting.*

Example 2. By introducing the completely fresh *split predicate* $\mathsf{S}(\mathsf{x})$, the clause $\{ \mathsf{P}(\mathsf{x}, \mathsf{y}),\ \mathsf{Q}(\mathsf{x}, \mathsf{z}) \}$ can be split into the follwing clauses. Note that we by doing this have reduced the maximum number of variables per clause from three to two.

$$\{ \mathsf{P}(\mathsf{x}, \mathsf{y}),\ \mathsf{S}(\mathsf{x}) \}$$
$$\{ \neg\mathsf{S}(\mathsf{x}),\ \mathsf{Q}(\mathsf{x}, \mathsf{z}) \}$$

The general criterion for splitting that we use look like this.

Definition 2 (Binary Split). *Given a clause $C[\boldsymbol{x}] \cup D[\boldsymbol{y}]$ where C, D are subclauses and $\boldsymbol{x}, \boldsymbol{y}$ are the sets of variables occuring in them. Then C and D constitute a* proper binary split *of the given clause iff there exist at least one variable x in $\boldsymbol{x}$*

such that $\mathsf{x} \notin \boldsymbol{y}$*, and at least one variable* y *in* $\boldsymbol{y}$ *such that* $\mathsf{y} \notin \boldsymbol{x}$*. The resulting two clauses after the split are:*

$$\{\ \mathsf{S}(\boldsymbol{x} \cap \boldsymbol{y})\ \} \cup C[\boldsymbol{x}]$$
$$\{\ \neg\mathsf{S}(\boldsymbol{x} \cap \boldsymbol{y})\ \} \cup D[\boldsymbol{y}]$$

Here, S *must be a fresh predicate symbol not occurring anywhere in the problem.*

The resulting two clauses contain less variables per clause since x is guaranteed not to appear in the second clause, and y not in the first clause. A special case is when $\boldsymbol{x} \cap \boldsymbol{y}$ is empty, in which case S becomes nullary predicate, i.e. a logical constant.

Repeated Binary Splitting In general the resulting clauses of a binary split can possibly be splitted further, thus to get best possible result binary splits can be repeated on both resulting clauses until for as long as it is possible. However, there might be several possible ways to apply a binary split, and a greedy choice could destroy the possibilities for further splitting of the resulting clauses. It might be worthwhile to come up with an optimal (in terms of number of variables) algorithm for repeated binary splitting, but so far we have reached good results using a cheap heuristic.

Existing heuristics for binary splits Gandalf [13] and Eground [10] both incorporate the same heuristic for finding binary splits, which works as follows. Given a clause C, all proper subsets of variables occurring in C are enumerated (small subsets first). For each subset V, it is checked if the clause can be split into two clauses, such that the intersection of variables occurring in both clauses is equal to V, which takes linear time in the length of the clause. If such a subset V is found, the clause is split accordingly. Since there is an exponential amount of such subsets, there is an upper limit (an arbitrary constant) on the amount of subsets that is tried. Beyond the limit, the algorithm gives up. The problem here is that for clauses containing many variables, we have to be lucky and find the right subset before we pass the upper limit.

Our heuristic In contrast, the heuristic we use is polynomial and it will always find a split if there is one, though it might not always turn out to be the best split. First, given a clause C, we say that two variables are *connected* in C if there is some literal in C in which they both occur. Note that if all variables are connected to each other, then a proper split is impossible, but otherwise it is. The heuristic now finds the variable x which is connected to the least amount of other variables in C. As soon as we find x, we take all literals containing x on one side of the split, and all other literals on the other side. One advantage of this method is that we know that the side of the split containing x cannot be split any further, so we only have to continue splitting the other side. Our heuristic seems to work well in practice, and works even for clauses with many variables.

Term definitions In cases where literals contain deep ground terms we can avoid introducing auxiliary variables, by introducing fresh constants as names for the terms, and substituting the terms for their names.

Example 3. Flattening the clause $\{\ \mathsf{P}(\mathsf{f}(\mathsf{a},\mathsf{b}),\mathsf{f}(\mathsf{b},\mathsf{a}))\ \}$ yields the clause:

$$\{\ \mathsf{a} \neq \mathsf{x},\ \mathsf{b} \neq \mathsf{y},\ \mathsf{f}(\mathsf{x},\mathsf{y}) \neq \mathsf{z},\ \mathsf{f}(\mathsf{y},\mathsf{x}) \neq \mathsf{w},\ \mathsf{P}(\mathsf{z},\mathsf{w})\ \}$$

This clause, which cannot be splitted (all variables are connected to each other), contains 4 variables. However, if we first transform the original clause by introducing

fresh names for its ground terms, we obtain the following satisfiability-equivalent set of clauses:

$$\{ \mathsf{t}_1 = \mathsf{f}(\mathsf{a}, \mathsf{b}) \}$$
$$\{ \mathsf{t}_2 = \mathsf{f}(\mathsf{b}, \mathsf{a}) \}$$
$$\{ \mathsf{P}(\mathsf{t}_1, \mathsf{t}_2) \}$$

If we now flatten these, we get the following three clauses:

$$\{ \mathsf{a} \neq \mathsf{x}, \mathsf{b} \neq \mathsf{y}, \mathsf{f}(\mathsf{x}, \mathsf{y}) \neq \mathsf{z}, \mathsf{t}_1 = \mathsf{z} \}$$
$$\{ \mathsf{a} \neq \mathsf{x}, \mathsf{b} \neq \mathsf{y}, \mathsf{f}(\mathsf{y}, \mathsf{x}) \neq \mathsf{z}, \mathsf{t}_2 = \mathsf{z} \}$$
$$\{ \mathsf{t}_1 \neq \mathsf{x}, \mathsf{t}_2 \neq \mathsf{y}, \mathsf{P}(\mathsf{x}, \mathsf{y}) \}$$

These clauses each contain 3 variables; a significant improvement.

In the general case, a clause $C[t]$ is translated into the clauses $\{ \mathsf{a} = t \}$ and $C[\mathsf{a}]$, where t is a non-constant ground term and a is a fresh constant, not occurring anywhere else in the problem. In the clause $C[\mathsf{a}]$ only one variable needs to be introduced for the constant a, in constrast to one for all subterms of t. Note also that if the term t occurs in several different clauses in the problem, then there only has to be one definition $\{ \mathsf{a} = t \}$, and the fresh constant a can be reused by all clauses containing t.

5 Incremental Search

The most popular basic algorithm for SAT solving, the DPLL procedure [3], is a backtracking procedure based on unit propagation. Modern versions of the algorithm usually also include several improvements, such as heuristics for variable selection, backjumping, and conflict learning.

In our context, *conflict learning* is of particular interest. It allows the procedure to learn from earlier mistakes. Concretely, for each contradiction that occurs during the search, the reason for the conflict is analysed, resulting in a *learned clause* that may avoid similar situations in future parts of the search. In this way, a set of conflict clauses is gathered during the search, representing information about the search problem. A conflict clause is always logically implied by the original problem, and thus holds without any assumption.

As part of our tool Paradox, we have implemented a Chaff-style [9] version of the DPLL algorithm[1], extended with the possibility to *incrementally* solve a sequence of problems. The idea is that we want to benefit from the similarity of the sequence of SAT instances, generated by our propositional encoding for each domain size. This is done by keeping the learned clauses generated by the search for one instance, also for the next.

Here is a formalization of the kind of sequences of SAT problems that our incremental SAT solver can deal with.

Definition 3. *Given a sequence of sets of propositional clauses δ_i, and a sequence of sets of propositional unit clauses α_i. Then the sequence φ_i, defined as follows, is an* incremental satisfiability problem.

$$\varphi_1 = \alpha_1 \cup \delta_1$$
$$\varphi_2 = \alpha_2 \cup \delta_1 \cup \delta_2$$
$$\varphi_3 = \alpha_3 \cup \delta_1 \cup \delta_2 \cup \delta_3$$
$$\ldots$$

[1] The SAT solver, called Satnik, is also a stand alone tool in itself

That is, to move from one instance to the next, we have to keep all the general clauses δ_i, but can retract and replace the unit clauses α_i. The incremental SAT algorithm we use represents the α_i as assumptions, and not as constraints. Therefore, we can keep *all* learned clauses from one instance and reuse them in the next instance. This is because every learned clause generated by the conflict analysis algorithm is implied by the subset $(\bigcup_{j=1}^{i} \delta_j)$ of the problem instance.

Model Generation as Incremental Satisfiability In Section 3 we described how to encode the problem of finding a model of a specific size into propositional logic. It is easy to see that the encodings for different sizes have much in common, but in order to specify it as an incremental satisfiability problem we need to be precise about what the difference is.

Given the SAT instance for a specific size s we want to create the instance for the size $s + 1$. Then for the *instances* and *function definitions*, we can keep all previous clauses, and we only have to add the new clauses that mention the new domain element $s + 1$. For the *totality definitions* however, we need to take away the clauses and replace them with less restrictive clauses.

In order to fit this in the incremental framework mentioned above, we introduce a special propositional variable $\mathsf{d_s}$ for each domain size s. This variable should be interpreted as "the current domain size is s". Instead of adding a totality clause as it is, we add a *conditional* variant of it, by adding $\neg\mathsf{d_s}$ as a literal to each totality clause. When solving the problem for domains of size s, we simply take $\mathsf{d_s}$ being true as an assumption unit clause α_s. This immediately implies the unconditional versions of the totality clauses. Then, if we find a model, we are done. Otherwise, $\mathsf{d_s}$ was apparently a too strong assumption, and we thus retract the assumption α_s, and add $\neg\mathsf{d_s}$ as a top-level unit clause. By doing this, we have effectively "deactivated" the totality clauses for size s by satisfying them, and are ready to add clauses for the next domain size $s + 1$.

Example 4. Assume that we have two constants a and b and a current domain size of 2. Then the conditional totality definitions look like this:

$$\{\ \mathsf{a} = 1,\ \mathsf{a} = 2,\ \neg\mathsf{d_2}\ \}$$
$$\{\ \mathsf{b} = 1,\ \mathsf{b} = 2,\ \neg\mathsf{d_2}\ \}$$

The problem for size 2 is now solved by assuming $\mathsf{d_2}$. We can get to the problem for size 3 by adding the following clauses:

$$\{\ \neg\mathsf{d_2}\ \}$$
$$\{\ \mathsf{a} = 1,\ \mathsf{a} = 2,\ \mathsf{a} = 3,\ \neg\mathsf{d_3}\ \}$$
$$\{\ \mathsf{b} = 1,\ \mathsf{b} = 2,\ \mathsf{b} = 3,\ \neg\mathsf{d_3}\ \}$$

Assuming $\mathsf{d_3}$ now gives the right totality clauses for size 3.

Effectiveness In our preliminary experiments, the method of incrementally solving the model generation problem for increasing domain sizes decreases the overall time spent in the SAT solver in many cases by at least a factor of 2. The implementation of the SAT solver removes clauses that are trivially satisfied because of the presence of other unit clauses. This mechanism takes care of removing the reduntant totality clauses of previous sizes.

There are also some questions left to investigate, particularly which of the learned clauses should be kept between instances. In general it slows the SAT solver down too much to store all of them, apart from the fact that it is also too space consuming. Currently we simply use our SAT solver's basic heuristic for learned clause

removal, which is designed to work well for solving single problems. It is likely that one could design other heuristics that would take into account the fact that a clause that seems to be uninteresting in the current part of the current search problem, in fact could be useful in the next problem instance.

6 Static Symmetry Reduction

The way we have expressed the model finding problem in SAT implies that for each model, all of its isomorphic valuations (i.e. the valuations we get by permuting the domain elements) are also models. This makes the SAT problem unneccessarily difficult. SEM-style methods use *symmetry reduction techniques* such as the *least number heuristic* (LNH) [5] and the *extended least number heuristic* (XLNH) [2] in order to restrict the search space to particular permutations of models. This is done while the search for a model is going on, i.e. *dynamically*. In order not to have to change the inner workings of the SAT-solver, we adapt some of the ideas behind these symmetry reduction techniques, but implement them by adding extra constraints, which remove symmetries *statically*.

Constant symmetries Let us start with the simple case, and only look at the values of the constants occurring in the problem. If we order all constants occurring in the problem in some arbitrary way, we get a sequence $\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_k$. Suppose that we are searching for a model of size s. We now require that the model we are looking for has a certain *canonical form*. This canonical form corresponds to a certain permutation of the domain elements, namely such that $\mathsf{a}_1 = 1$, and for all $i > 1$, $\mathsf{a}_i = d$ only when there is a $j < i$ such that $\mathsf{a}_j = d - 1$. So, when picking a domain element for a_i, we can either pick an element that we have already seen, or a new element, which must be the *least* element in D which we have not used yet. It is easy to see that every interpretation has an isomorphic permutation where this is the case.

Adding this extra restriction implies that $\mathcal{I}(a_i) \leq i$, which immediately gives rise to the following extra clauses:

$$\{\ \mathsf{a}_1 = 1\ \}$$
$$\{\ \mathsf{a}_2 = 1,\ \mathsf{a}_2 = 2\ \}$$
$$\{\ \mathsf{a}_3 = 1,\ \mathsf{a}_3 = 2,\ \mathsf{a}_3 = 3\ \}$$
$$\ldots$$

These clauses actually subsume the totality clauses for their constants. Also, for any $1 < i \in D$, and for $1 < d \leq i \in D$, we add the following clause, which directly formulates the canonicity requirement:

$$\{\ \mathsf{a}_i \neq d,\ \mathsf{a}_1 = d - 1,\ \mathsf{a}_2 = d - 1,\ \mathsf{a}_3 = d - 1,\ \ldots,\ \mathsf{a}_{i-1} = d - 1\ \}$$

That is, a_i can only get the value d if some previous constant already has used the value $d - 1$.

This process can be adapted for an incremental search in the following way: for each new model size, we add only those symmetry-reducing clauses that contain the new domain element, and none of the greater elements. We never take away any of the generated symmetry reduction clauses in the incremental search.

Function symmetries If the problem only contains function symbols of arity 0, then the above clauses are enough to remove *all* symmetries. However, when there are function symbols of higher arity, this is no longer easy to do statically. We can however remove some of the symmetries by, in addition to the above clauses,

adding the following clauses for a function symbol f of arity 1, when we are looking for models of sizes s greater than k (the number of constants). We require that $k > 0$ (we just introduce an arbitrary constant when $k = 0$).

$$\{\ \mathsf{f}(1) = 1,\ \mathsf{f}(1) = 2,\ \ldots,\ \mathsf{f}(1) = k+1\ \}$$
$$\{\ \mathsf{f}(2) = 1,\ \mathsf{f}(2) = 2,\ \ldots,\ \mathsf{f}(2) = k+1,\ \mathsf{f}(2) = k+2\ \}$$
$$\{\ \mathsf{f}(3) = 1,\ \mathsf{f}(3) = 2,\ \ldots,\ \mathsf{f}(3) = k+1,\ \mathsf{f}(3) = k+2,\ \mathsf{f}(3) = k+3\ \}$$
$$\ldots$$

The rationale here is again that, in order to pick a value for a particular $\mathsf{f}(d)$, we can simply use a element that we have already seen, or the least element that has not been used yet.

Note that we add only *one* such clause for each size increase beyond k. We have not investigated how to decide which function symbol to pick. In our current implementation, we simply pick an arbitrary function symbol. In principle it is possible to use a different function symbol in every clause. We can also generalize the above for function symbols g of arity larger than 1, by defining a fresh function f in terms of g by e.g. $\mathsf{f}(\mathsf{x}) = \mathsf{g}(\mathsf{x}, \mathsf{x})$.

The resulting SAT problem, even though it is a little bit bigger than without the symmetry reducing clauses, is often dramatically easier to solve, both in cases where there is a model, and in cases where there is no model.

7 Sort Inference

When formalizing a problem in terms of unsorted FOL, there often exist different *concepts* in the problem, which when finding a model, all have to be interpreted using the same domain D. This can be quite unnatural, both when trying to understand a model and when trying to find a model. Examples of these kinds of concepts are points, lines, and planes in geometry problems, and booleans and numbers in system descriptions. A 'typed' version of FOL, Multi-Sorted First Order Logic (MSFOL), requires these concepts, the *sorts*, to be explicit in the formulation of the problem. It is well-known that sort information helps searching for models. In this section, we describe how to use the sort information, and, more interestingly, how to *infer* the sort information such that it can be used for originally unsorted problems as well.

Sorted models In the MSFOL world, apart from predicate symbols and function symbols, there exists *sort* symbols $\mathsf{A}, \mathsf{B}, \mathsf{C}$. Each function symbol f has an associated sort $\mathsf{sort}(\mathsf{f})$ of the form $\mathsf{A}_1 \times \ldots \times \mathsf{A}_{\mathsf{ar}(\mathsf{f})} \to \mathsf{A}$, and each predicate symbol P (except for $=$, which works on all sorts) has an associated sort $\mathsf{sort}(\mathsf{P})$ of the form $\mathsf{A}_1 \times \ldots \times \mathsf{A}_{\mathsf{ar}(\mathsf{P})}$. Moreover, in each clause, each variable x has an associated sort $\mathsf{sort}(\mathsf{x})$ of the form A. These sorts have to be respected in order to build only well-sorted terms and literals.

An MSFOL interpretation $\mathcal{M}$ consists of a domain D_{A} for each sort A, plus for each function symbol f a function $\mathcal{I}(\mathsf{f}) : D_{\mathsf{A}_1} \times \ldots \times D_{\mathsf{A}_{\mathsf{ar}(\mathsf{f})}} \to D_{\mathsf{A}}$ matching the sort of f, and for each predicate symbol P a function $\mathcal{I}(\mathsf{P}) : D_{\mathsf{A}_1} \times \ldots \times D_{\mathsf{A}_{\mathsf{ar}(\mathsf{P})}} \to \mathcal{B}$, matching the sort of P.

We define the notion of satisfiability for MSFOL interpretations in the obvious way: quantification of variables in clauses becomes sort-dependent.

Unsorted vs. sorted Now, since our objective is to find an unsorted model, in order to make use of sorts, we must link unsorted models and sorted models in some way. It is not automatically the case that if we find a sorted model, there is also an

unsorted model of the same problem. However, it is the case that we can turn any unsorted model of a problem into a sorted model of the same problem.

The basic observation is that any unsorted interpretation $\mathcal{I}$ with a domain D can be turned into a sorted interpretation $\mathcal{M}_\mathcal{I}$ by taking $D_\mathsf{A} = D$ for each sort A, and by simply reusing all function and predicate tables from $\mathcal{I}$. Now, for a suitably well-sorted set of clauses S, we have that $\mathcal{I}$ satisfies S iff $\mathcal{M}_\mathcal{I}$ satisfies S. So, the key idea is that when searching for an unsorted model $\mathcal{I}$, we can just as well search for the sorted model $\mathcal{M}_\mathcal{I}$, i.e. search for a sorted model where all sorts have the same domain size.

The advantage of searching for a sorted model becomes clear in the following, which is a more fine-grained version of the symmetry observation for unsorted interpretations. Given an MSFOL interpretation $\mathcal{M}$ containing a domain D_A for a sort A satisfying an MSFOL clause set S. Given a set D' and a bijection $\pi : D_\mathsf{A} \leftrightarrow D'$, we construct a new interpretation $\mathcal{M}'$ by replacing D_A by D' and applying π in the obvious way. Now, $\mathcal{M}'$ also satisfies S.

Sorted symmetry reduction We can now make the following refinement of our earlier symmetry reduction method. Given a valid sort-assignment to each function and predicate symbol, we can simply search for an MSFOL model where the domains for each sort are the same, but we can apply symmetry reduction for each sort separately. That is, for each sort, we create a sequence of the constants of that sort, and we add the extra clauses mentioned in Section 6.

Example 5. Given a problem with three constants $\mathsf{a}, \mathsf{b}, \mathsf{c}$ and two sorts A, B, where $\mathsf{sort}(\mathsf{a}) = \mathsf{sort}(\mathsf{b}) = \mathsf{A}$ and $\mathsf{sort}(\mathsf{c}) = \mathsf{B}$, we get the following symmetry reduction clauses:

$$\{\ \mathsf{a} = 1\ \}$$
$$\{\ \mathsf{b} = 1,\ \mathsf{b} = 2\ \}$$
$$\{\ \mathsf{c} = 1\ \}$$

Sort inference The big question is then: How do we get such a suitable sort-assignment for a flattened unsorted clause set? The algorithm we use is simple. In the beginning, we assume that all function symbols and predicate symbols have completely unrelated sorts. Then, for each variable in each clause, we force the sorts of the occurences of that variable to be the same. Also, we force the sorts on both sides of the $=$ symbol to be the same. This can be implemented by a straight-forward union-find algorithm, so that the whole agorithm runs in linear time. In the end, the hope is to be left with more than one sort.

We have found that this simple sort inference algorithm really finds multiple sorts in about 30% of the (unsorted) problems occurring in the current TPTP benchmark suite [12] over all, and in about 50% of the satisfiable problems particularly.

Sort size reduction There is another way in which we can make use of the inferred sort information in model finding. Under certain conditions, we can restrict the size of the domains for particular sorts, which reduces the complexity of the instantiation procedure.

Suppose that we have a sort A, and k constants $\mathsf{a}_1, \ldots, \mathsf{a}_k$ of sort A, but no function symbols of arity greater than 0 which have A as their result sort. Moreover, assume that we are not using the $=$ symbol positively on terms of sort A anywhere in the problem S, then the following holds. There exists an MSFOL model of S with a domain D_A of size k iff there exists an MSFOL model of S with a domain D_A of size greater than k. In other words, to find an MSFOL model of S, we do not have to instantiate variables of sort A with more than k domain elements. This can considerably reduce instantiation time for problems where sorts are inferred.

The proof looks as follows. ($\Leftarrow$) If we have a model where D_{A} has more than k elements, there must be an element d which is not the value of any constant, so we can safely take it away from the domain, and all function tables and predicate tables will still be still well-defined. It is also still a model, since making a domain smaller only increases the number of clause sets that are satisfied. ($\Rightarrow$) If we have a model of S, then we can always add a new element d' to D_{A} by picking an existing element $d \in D_{\mathsf{A}}$ and making all functions and predicates produce the same results for d' as they do for d. The resulting interpretation is still a model, because every literal evaluates to the same value for d' as it does for d. (However, this is only true for non-equality literals.) For negative equality, making the domain bigger can only increase the number of clause sets that are satisfied. This is not true for the use positive equality literals, which is the reason why it is disallowed in the assumption.

(It is however possible to weaken the restriction on the use of positive equality. In order to be able to restrict the size of the domain of A, it is enough to require that we do not use positive literals of the form $\mathsf{x} = \mathsf{y}$, where x and y are variables of sort A. So, it is okay to use positive literals of the form $t = t'$ and $t = \mathsf{x}$, where t and t' are not variables. In the latter case however, we generally need to consider $k + 1$ elements instead of k.)

EPR problems A special case of sort size reduction is the case where the problem is an *Effectively Propositional* (EPR) problem, i.e. no functions of arity greater than 0 occur in the problem at all. In this case, each sort only contains constants, and the number of constants k in the largest sort is an upper bound on the size of models we need to consider. (This is independent of the use of equality in the problem, since we only need the ($\Leftarrow$) part of the above proof.) When no model of size up to k is found, we know there can be no model of greater size, and therefore the problem must be contradictory. Thus we have a complete method for EPR problems.

8 Experimental Results

We have implemented all the techniques in a new finite model finder called Paradox. Here is a list of promising concrete results we have obtained so far with our model finder:

- On the current TPTP version 2.5.0 [12], and with a time limit of 2 minutes for each problem, we can solve 90% of the satisfiable problems. This is significantly better than last year's CASC winner in the satisfiability category on the same problems with a time limit of 5 minutes.
- Within a time limit of 10 minutes, we have solved 28 problems from the current TPTP which currently have a rating 1.0 (i.e. Paradox is the first to solve those problems), including 15 "open" or "unknown" problems that were solved within seconds.
- With an older version of Paradox (using different term definitions heuristics and SAT solver parameters), in the search for counter models for the combinatory logic question if the fragment {B,M} posseses the fixed point property, we have shown that there are no counter models of sizes smaller than or equal to 10, which took us 9 hours. The previously known bound was 7.

Our preliminary findings are that the techniques described in this paper strictly improve on all known MACE-based methods. Also, they perform almost always better than SEM-based methods on problems that contain more than just unit equalities. SEM-based methods however are superior on most problems that contain lots of unit equalities, such as group theory problems. Interestingly, there are some exceptions, such as combinatory logic problems, where Paradox seems to behave well.

9 Related Work

There are several tools that solve similar problems as we do, or use similar techniques to the ones we use.

Eground [10] is a tool that takes an EPR problem, i.e. a problem that does not contain any function symbols of arity greater than 0, and generates a SAT problem that is satisfiable iff the original problem is. Interestingly, Eground has many of the same problems as a MACE-style model finder. Eground was the first tool to perform non-ground splitting in order to reduce the number of variables in clauses. Also, Eground performs an analysis that computes sets of constants for each variable in a clause for which the variable should be instantiated. The hope is that these sets are smaller than the set of all constants in the problem. The analysis is somewhat similar to sort inference, with three main differences. Our analysis also works for non-EPR problems, and also works for problems containing equality. Eground's analysis makes use of the sign of predicate symbols, which makes it more precise in some cases.

Comparing Eground as an EPR solver with our proposed method of solving EPR problems mentioned at the end of Section 7, it seems that they are complementary. Assuming that no equality is used in the problem, and that the analyses work equally well, and ignoring the symmetry reduction, we can make the following rough observations. Given an EPR problem that is contradictory, Eground's method is probably going to win over ours since it immediately tries the "biggest" case, whereas our method will go through all smaller model sizes first. Given an EPR problem that is satsifiable, it is very likely that it is not needed to go all the way up to the biggest case, and that a smaller model can be found much quicker.

MACE [7] is McCune's first finite model finder. The basic idea behind MACE is described in Section 3. Since it does not perform any variable reduction techniques, there are many problems where MACE cannot deal with largish domain sizes. It has its own built-in SAT solver which is currently not up-to-date with the current state-of-the-art SAT technology.

SEM [5], FINDER [11], and ICGNS [8] are all SEM-based tools. SEM and FINDER are specifically designed for sorted problems, whereas ICGNS only works for unsorted problems. Comparing the symmetry reduction in SEM-based tools with ours, we can say that their symmetry reduction works dynamically, i.e. during the search they will always pick the smallest not-used domain element when a new element is needed. We apply the symmetry reduction statically, which removes the same symmetries when picking values for constants, but for functions, our extra function symmetry clauses do not remove as many symmetries. Still, having a state-of-the-art SAT solver as the underlying search engine seems to be superior in many cases once one is able to instantiate the problem for the desired domain size. It will continually be useful to investigate these complementary method's strengths and weaknesses in order to understand the problem area better.

Gandalf [13] is a general theorem prover that also implements model finding. Gandalf contains lots of different complementary algorithms for particular problem domains, which are, upon receiving a problem, scheduled, together occupying all available time. For satisfiability, Gandalf provides saturation techniques (that help finding cases with infinite models), SEM-style techniques, and one MACE-style method. As far as we know, Gandalf was the first to use splitting techniques in MACE-style model finding. Gandalf was the winner of last year's CASC satisfiability division.

A more general approach to incremental SAT solving than what we use here is used in the tool Satire [6]. In Satire, one can take away and add clauses arbitrarily. This requires extra bookkeeping to implement. Our method requires one to decide on beforehand which clauses are going to be retracted. Fortunately, Satire's extra

generality is not needed in our application, and our simple, more efficient (but more restrictive) approach is enough.

10 Conclusions and Future Work

We have shown that MACE-style methods can be improved upon by incorporating symmetry reduction methods (inspired by well-known related work in the SEM world), by adding them as static constraints to the generated SAT problem. The automatic inference of sorts in order to refine the symmetry reduction turned out to be a very powerful tool in this context. We have also shown that it is good to intimately integrate a SAT solver with the algorithm that uses it, in order to get the most benefit out of it.

However, our work on reducing the number of variables in clauses by using splitting methods and term definitions is, though very promising, only showing the tip of the ice berg of what is left to do in the area. Our splitting heuristic seems to perform well in practice, but it is unsatisfactory that it is based on repeated binary splits. We have not been able to formulate a "most general" clause *hyper*-splitting condition, which all correct splitting algorithms must obey; all previous attempts have been too restrictive. This has made it impossible for us to explore the design space of splitting algorithms in a satisfactory way. A similar situation holds for the term definitions, where it is unclear exactly when term definitions should be introduced. Ultimately we would like to integrate splitting and term definitions in order to get the best of both worlds.

Other future work includes improving the sort inference algorithm which is very simple at the moment. The problem can be thought of as a *flow-analysis* problem from the field of program analysis, and much inspiration can be found there. Also, we would generalize the sort inference to already sorted problems, in order to find more fine-grained sort assignments than the sort-assignment declared in the problem.

Another direction of research is to adapt clausification algorithms in order to perform clausification, flattening, and splitting at the same time. The basic decision a clausification algorithm must make is when to introduce a new name for a subformula. This decision is guided by optimizing certain parameters of the resulting problem, usually the number of resulting clauses or literals. We could adapt such an algorithm to minimize number of variables per clause instead.

Some problems are inherently complex because they for example contain predicate or function symbols with a huge arity. In order to even represent (let alone search for) models of reasonable domain size would require too much memory. One idea we have started to explore is to *strengthen* the original theory by replacing the offending symbols by nested expressions containing function symbols of much lower arity. If we find a model of the strengthened problem, which might not exist anymore but is hopefully easier to do, that model can be translated back into a model of the original problem. (Of course, introducing these huge predicates is ultimately a modelling question; something that the modeller of the original problem should think about.)

Lastly, we have looked at non-standard applications of our model finding techniques in the fields of planning (where a found model represents a plan), finite state system verification (where a found model represents a proof of the correctness of the system), and general FOL theorem proving (where we use finite models to approximate possible infinite models, and the absence of such a finite approximation model beyond a certain precision represents the absence of a model alltogether).

References

1. A. Voronkov A. Riazanov. Splitting without backtracking. Technical Report Preprint CSPP-10, University of Manchester, 2000.
2. Gilles Audemard and Laurent Henocque. The eXtended Least Number Heuristic. *Lecture Notes in Computer Science*, 2083, 2001.
3. D. Loveland D. Putnam, G. Logeman. A machine program for theorem proving. *Communications of the ACM*, 5(7), 1962.
4. H. Zhang J. Zhang. SEM: a system for enumerating models. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1995.
5. H. Zhang J. Zhang. Generating models by SEM. In *Proc. of International Conference on Automated Deduction (CADE'96)*, pages 308–312. Springer-Verlag, 1996.
6. Karem A. Sakallah Jesse Whittemore, Joonyoung Kim. Satire: A new incremental satisfiability engine. In *Design Automation Conference*, pages 542–545, 2001.
7. W. McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994. http://www-unix.mcs.anl.gov/AR/mace/.
8. W. McCune. ICGNS, 2002. http://www-unix.mcs.anl.gov/~mccune/icgns/.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
10. S. Schulz. A comparison of different techniques for grounding near-propositional CNF formulae. In *Proc. 15th International FLAIRS Conference*, pages 72–76, 2001.
11. John Slaney. FINDER 3.0, 1993. http://arp.anu.edu.au/~jks/finder.html.
12. Geoff Sutcliffe and Christian Suttner. TPTP v. 2.5.0, 2003. http://www.tptp.org.
13. T. Tammet. Gandalf, 2002. Webpage momentarily unknown. Available from tammet@staff.ttu.ee.