Using SAT and SMT Solvers for Finite Model Finding with Sorts

Giles Reger[1] Martin Suda[2]

[1] University of Manchester, Manchester, UK
giles.reger@manchester.ac.uk
[2] Institute for Information Systems, Vienna University of Technology, Austria
msuda@forsyte.tuwien.ac.at

Abstract: We report on a recent technique for finding finite models in multi-sorted first-order logic. The approach extends the MACE-style approach of encoding the check for the existence of a finite model of a certain size as a SAT problem. To deal with multiple sorts the space of possible assignments of sort domain sizes is searched. To efficiently navigate through this space, arithmetical constraints are produced and passed to an SMT solver which produces an answer describing the next domain size assignments to try. The technique is implemented in the Vampire theorem prover.

1 Introduction

One method for establishing satisfiability of a first-order formula is to search for a *finite model*. Such models are useful in a number of applications. There have been various approaches to finding finite models in first-order logic. The technique we are interested in was pioneered by the MACE model finder [4] and extended by the Paradox work [2]. The basic idea is to encode the check for a finite model of a certain size as a SAT problem.

We consider the extension of first-order logic with sorts. In this setting a finite model may need to use different domain sizes for different sorts (see the example below). Our new technique [5] explores the space of possible domain size assignments (one domain size for each sort) by producing *constraints* from each failed attempt and using an SMT solver (in our case Z3 [3]) to guide the search.

2 The Monkey Village Example

There is a much-used simple example for finite model finding with sorts. For a more interesting (similar) example see our paper [5]. The example involves a village of monkeys where each monkey owns at least two bananas. This can be captured by two formulas:

$$\begin{array}{l}(\forall M : \mathit{monkey})(\mathsf{b}_1(M) \neq \mathsf{b}_2(M) \wedge \\ \qquad\qquad \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))) \\ (\forall M_1, M_2 : \mathit{monkey})(\forall B : \mathit{banana}) \\ \qquad\qquad (\mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 = M_2)\end{array}$$

where the predicate owns associates monkeys with bananas and the functions b_1 and b_2 witness the existence of each monkey's minimum two bananas.

The smallest finite model for these formulas has a domain of size 1 for *monkey* and a domain of size 2 for *banana*. The two sorts cannot have the same size as there must be at least twice as many bananas as monkeys.

3 MACE-Style Finite Model Building

The MACE-Style approach produces a grounding of the first-order problem using a given n domain constants and then encodes this as a SAT problem. For this grounding to be sound the problem needs to be put into *flattened clausal* form where function and predicate symbols are only applied to variables, e.g. $\mathsf{owns}(M, \mathsf{b}_1(M))$ becomes $\mathsf{owns}(M, x) \vee \mathsf{b}_1(M) \neq x$. The flattened clauses are then instantiated with all mappings of variables to domain constants. One then needs to also encode information about *functionality* and *totality* of functions. For example, for the unary function b_1 and domain constants d_1 and d_2 for *banana* and c_1 for *monkey* we add the following:

$$\begin{array}{ll}\mathit{Functionality} & \mathsf{b}_1(c_1) \neq d_1 \vee \mathsf{b}_1(c_1) \neq d_2 \\ \mathit{Totality} & \mathsf{b}_1(c_1) = d_1 \vee \mathsf{b}_1(c_1) = d_2\end{array}$$

One can optionally add *symmetry breaking* information by ordering ground terms and making the smallest equal the first domain element, etc. This is necessary for efficiency.

The search for finite models then involves producing and checking SAT encodings for increasing domain sizes.

4 Adding Sorts

The previous encoding can be lifted to the multi-sorted setting by introducing a set of domain constants per sort and instantiating variables by constants from their sort. As previously noted, the number of domain constants per sort may (necessarily) vary. We utilise a simple breadth-first search algorithm of the possible domain size assignments. This search is driven by *constraints* derived from failed SAT proofs as summarised below.

Getting Constraints We update the above encoding so that a failed check for a model can give us some insight into why the check failed. We extend the encoding with two extra labels $|s| > n$ and $|s| < n$ for each sort s and for the concrete value n of the current size of the domain of s. Intuitively these stand for the size of sort s being too small or too big, respectively.

If we cannot satisfy a totality condition then the size of the sort is too small (we need more domain constants) and

Table 1: Experimental Results .

	CVC4	Paradox	iProver	Vampire		CVC4	Vampire
FOF+CNF: sat	1181	1444	1348	**1503**	UF: sat	764	**896**
FOF+CNF: unsat	-	-	1337	**1628**	UF: unsat	-	249

therefore we can extend the totality constraints to

$$\mathsf{b}_1(d_1) \neq d_1 \vee \mathsf{b}_1(d_1) \neq d_2 \vee |monkey| > 2$$

where the current size of $monkey$ is 2. Conversely, if we cannot satisfy a grounded input clause then the size of s may be too large and we can extend the grounding to

$$\mathsf{owns}(c_2, d_1) \vee \mathsf{b}_1(c_2) \neq d_1 \vee |monkey| < 2$$

using the previous clause from our example.

After extending the encoding we solve the SAT problem under the assumptions that we are using the correct sort sizes, i.e. we add the following for each sort s

$$\neg(|s| > 2) \wedge \neg(|s| < 2).$$

The mechanism for solving under assumptions, supported by many SAT solvers, provides a subset of these assumptions sufficient to show unsatisfiability of the SAT problem. This subset is our set of constraints that explains why this check failed.

Using Constraints To use these constraints to guide the search we use an SMT solver to find a model of the constraints. This model will assign a value to each domain size, giving us the next SAT problem to check. To ensure that this search grows appropriately we add the additional constraint that the sum of the sort sizes must initially equal the number of sorts. If no model can be found with this constraint then we add one to this value and try again.

5 More Fun with Sorts

We can do various other things in an attempt to improve this search for the right combination of sort sizes.

Monotonic Sorts A sort s is *monotonic* for a formula φ if adding another domain constant to s in a model of φ produces another model for φ (see [1]). Monotonic sorts can be easily detected and used in a number of ways.

Collapsing Sorts All monotonic sorts can be treated as a single sort. This reduces the size of the search space. However, if one of these sorts needs to be very large then all sorts will need to grow, potentially unnecessarily increasing the size of the SAT encoding.

Expanding Subsorts Alternatively, one can infer *subsorts* by identifying function and predicate positions that are disjoint with respect to variables. If these subsorts are monotonic then they can be treated as real sorts, with the constraint that they do not grow larger than their parent sort.

More Constraints It is possible to detect constraints *between* sorts due to (for example) an injective function from one sort to another. To detect such properties we adapt a standard saturation algorithm to use a single proof attempt to prove as many of these relationships as possible.

6 No Finite Model

There are cases where we can establish upper bounds on the size of a sort e.g. when it only uses constant symbols. These can be treated as additional constraints. If the resultant set of constraints is unsatisfiable without the previous restriction on sort sizes (that they sum to some number) then there is no model and the problem is unsatisfiable.

7 Experiments and Concluding Remarks

Table 1 gives a brief summary of the experimental results reported in [5]. Two sets of experiments are described. The first considers unsorted TPTP problems and applies sort expansion. The second considers SMT-LIB problems from the Uninterpreted Functions logic and applies monotonic sort grouping. The new techniques perform better than the other leading tools for finite model finding.

Future work involves introducing further heuristics for symmetry breaking and search. One option is to explore *incomplete* search strategies that skip parts of the search space; sacrificing finite model completeness for efficiency.

References

[1] Koen Claessen, Ann Lillieström, and Nicholas Smallbone. Sort it out with monotonicity - translating between many-sorted and unsorted first-order logic. In *CADE-23*, pages 207–221, 2011.

[2] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *CADE-19 Workshop: Model Computation - Principles, Algorithms and Applications*, 2003.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[4] William Mccune. A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical report, Argonne National Laboratory,, 1994.

[5] Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In *SAT-19*, 2016.