Finite Model Computation via Answer Set Programming

Martin Gebser and **Orkunt Sabuncu** and **Torsten Schaub***
Universität Potsdam, Potsdam, Germany
{gebser,orkunt,torsten}@cs.uni-potsdam.de

Abstract

We show how Finite Model Computation (FMC) of first-order theories can efficiently and transparently be solved by taking advantage of an extension of Answer Set Programming, called incremental Answer Set Programming (iASP). The idea is to use the incremental parameter in iASP programs to account for the domain size of a model. The FMC problem is then successively addressed for increasing domain sizes until an answer set, representing a finite model of the original first-order theory, is found. We developed a system based on the iASP solver *iClingo* and demonstrate its competitiveness.

1 Introduction

While Finite Model Computation (FMC; [Caferra *et al.*, 2004]) constitutes an established research area in the field of Automated Theorem Proving (ATP; [Robinson and Voronkov, 2001]), Answer Set Programming (ASP; [Baral, 2003]) has become a widely used approach for declarative problem solving, featuring manifold applications in the field of Knowledge Representation and Reasoning. Up to now, however, both FMC and ASP have been studied in separation, presumably due to their distinct hosting research fields. We address this gap and show that FMC can efficiently and transparently be solved by taking advantage of a recent extension of ASP, called incremental Answer Set Programming (iASP; [Gebser *et al.*, 2008]).

Approaches to FMC for first-order theories [Tammet, 2003] fall in two major categories, translational and constraint solving approaches. In translational approaches [McCune, 1994; Claessen and Sörensson, 2003], the FMC problem is divided into multiple satisfiability problems in propositional logic. This division is based on the size of the finite domain. A Satisfiability (SAT; [Biere *et al.*, 2009]) solver searches in turn for a model of the subproblem having a finite domain of fixed size, which is gradually increased until a model is found for the subproblem at hand. In the constraint solving approach [Zhang, 1996], a system computes a model by incrementally casting FMC into a constraint satisfaction problem. While systems based on constraint solving are efficient for problems with many unit equalities, translation-based ones are applicable to a much wider range of problems [Tammet, 2003]. Natural application areas of FMC include verification, where it can serve to identify counterexamples.

In fact, translational approaches to FMC bear a strong resemblance to iASP. The latter was developed for dealing with dynamic problems like model checking and planning. To this end, iASP foresees an integer-valued parameter that is consecutively increased until a problem is found to be satisfiable. Likewise, in translation-based FMC, the size of the interpretations' domain is increased until a model is found. This similarity in methodologies motivates us to encode and solve FMC by means of iASP.

The idea is to use the incremental parameter in iASP to account for the domain size. Separate subproblems considered in translational approaches are obtained by grounding an iASP encoding, where care is taken to avoid redundancies between subproblems. The parameter capturing the domain size is then successively incremented until an answer set is found. In the successful case, an answer set obtained for parameter value i provides a finite model of the input theory with domain size i.

We implemented a system based on the iASP solver *iClingo* [Gebser *et al.*, 2008] and compared its performance to various FMC systems. To this end, we used problems from the FNT (First-order form Non-Theorems) division of CADE's 2009 and 2010 ATP competitions. The results demonstrate the competitiveness of our system. On the benchmark collection used in 2009, *iClingo* solved the same number of problems as *Paradox* [Claessen and Sörensson, 2003] in approximately half of its run time on average. Note that *Paradox* won first places in the FNT division each year from 2007 to 2010.

The paper is organized as follows. The next section introduces basic concepts about the translational approach to FMC and iASP. Section 3 describes our encoding of FMC and how it is generated from a given set of clauses. Information about our system can be found in Section 4. We empirically evaluate our system in Section 5 and conclude in Section 6.

2 Background

We assume the reader to be familiar with the terminology and basic definitions of first-order logic and ASP. In what follows,

*Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

we thus focus on the introduction of concepts needed in the remainder of this paper.

In our approach, we translate first-order theories into sets of flat clauses. A clause is *flat* if (i) all its predicates and functions have only variables as arguments, (ii) all occurrences of constants and functions are within equality predicates, and (iii) each equality predicate has at least one variable as an argument. Any first-order clause can be transformed into an equisatisfiable flat clause via *flattening* [McCune, 1994], done by repeatedly applying the rewrite rule $C[t] \rightsquigarrow (C[X] \vee (X \neq t))$, where t is a term offending flatness and X is a fresh variable. For instance, the clause $(f(X) = g(Y))$ can be turned into the flat clause $(Z = g(Y)) \vee (Z \neq f(X))$. In the translational approach to FMC, flattening is used to bring the input into a form that is easy to instantiate using domain elements.

As regards ASP, we rely on the language supported by the grounder *gringo* [Gebser *et al.*, 2011], providing normal and choice rules as well as cardinality and integrity constraints. (For the use of ASP languages in declarative problem solving, the reader may refer to [Baral, 2003; Gebser *et al.*, 2011].) As usual, rules with variables are regarded as representatives for all respective ground instances. Beyond that, our approach makes use of iASP [Gebser *et al.*, 2008] that allows for dealing with incrementally growing domains. In iASP, a parameterized domain description is a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter k ranging over positive integers. Hence, we sometimes denote P and Q by $P[k]$ and $Q[k]$. The base program B describes static knowledge, independent of parameter k. The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k. Our goal is then to decide whether the program

$$R[i] = B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$$

has an answer set for some (minimum) integer $i \geq 1$, where $P[k/j]$ and $Q[k/i]$ refer to the programs obtained from P and Q by replacing each occurrence of parameter k with j or i, respectively. In what follows, we refer to rules in B, $P[k]$, and $Q[k]$ as being *static*, *cumulative*, and *volatile*, respectively.

3 Approach

In this section, we present our encoding of FMC in iASP. The first task, associating terms with domain elements, is dealt with in Section 3.1, and Section 3.2 describes the evaluation of (flat) clauses within iASP programs. In Section 3.3, we explain how a model of a first-order theory is then read off from an answer set. Section 3.4 addresses symmetry breaking in FMC. Due to space limitations, we omit some encoding details, which can be found in [Gebser *et al.*, 2010].

Throughout this section, we illustrate our approach on the following running example:

$$\begin{array}{l} p(a) \\ (\forall X)\ \neg q(X, X) \\ (\forall X)\ (p(X) \rightarrow (\exists Y)\ q(X, Y)). \end{array} \quad (1)$$

The first preprocessing step, *clausification* of the theory, yields the following:

$$\begin{array}{l} p(a) \\ \neg q(X, X) \\ \neg p(X) \vee q(X, sko(X)). \end{array}$$

The second step, *flattening*, transforms these clauses into the following ones:

$$\begin{array}{l} p(X) \vee (X \neq a) \\ \neg q(X, X) \\ \neg p(X) \vee q(X, Y) \vee (Y \neq sko(X)). \end{array} \quad (2)$$

Such flat clauses form the basis for our iASP encoding. Before we present it, note that the theory in (2) has a model I over domain $\{1, 2\}$ given by:

$$\begin{array}{ll} a^I & = 1 \\ sko^I & = \{1 \mapsto 2, 2 \mapsto 2\} \\ p^I & = \{1\} \\ q^I & = \{(1, 2)\}. \end{array} \quad (3)$$

In view of equisatisfiability, the model I satisfies also the original theory in (1), even if sko^I is dropped.

3.1 Interpreting Terms

In order to determine a model, we need to associate the (non-variable) terms in the input with domain elements. To this end, every constant c is represented by a fact $cons(c)$., belonging to the *static* part of our iASP program. For instance, the constant a found in (2) gives rise to the following fact:

$$cons(a). \quad (4)$$

Our iASP encoding uses the predicate $assign(T, D)$ to represent that a term T is mapped to a domain element D. Here and in the following, we write k to refer to the incremental parameter in an iASP program. Unless stated otherwise, all rules provided in the sequel are *cumulative* by default. For constants, the following (choice) rule then allows for mapping them to the kth domain element:

$$\{assign(T, k)\} \leftarrow cons(T). \quad (5)$$

The set notation in the head indicates that (unlike with strict rules such as the fact in (4)) one may freely choose whether to make $assign(T, k)$ true, provided that the body $cons(T)$ holds for a respective instance of T. Also note that, by using k in $assign(T, k)$, it is guaranteed that instances of the rule are particular to each incremental step.

Unlike with constants, the argument tuples of functions grow when k increases. To deal with this, we first declare auxiliary facts to represent available domain elements:

$$dom(k). \qquad arg(k, k). \quad (6)$$

Predicates dom and arg are then used to qualify the arguments of a function in the input, such as sko in (2). Its instances are in turn derived via the following rule:

$$func(sko(X)) \leftarrow dom(X), 1\{arg(X, k)\}. \quad (7)$$

The cardinality constraint $1\{arg(X, k)\}$ stipulates at least one of the arguments of sko, which is only X in this case, to be k. As in (5), though using a different methodology, this makes sure that the (relevant) instances of (7) are particular

to a value of k. However, note that rules of the above form need to be provided separately for each function in the input, given that the arities of functions matter.

To represent new mappings via a function when k increases, the previous methodology can easily be extended to requiring some argument or alternatively the function value to be k. For instance, the rule encoding mappings via unary function sko is as follows:

$$\{assign(sko(X), Y)\} \leftarrow dom(X), dom(Y), \\ 1\{arg(X, k), arg(Y, k)\}. \quad (8)$$

Observe that the cardinality constraint $1\{arg(X, k), arg(Y, k)\}$ necessitates at least one of argument X or value Y of function sko to be k, which in the same fashion as before makes the (relevant) instances of the rule particular to each incremental step.

To see how the previous rules are handled in iASP computations, we below show the instances of (6) and (8) generated in and accumulated over three incremental steps:

Step 1: $dom(1).\quad arg(1, 1).$
$\{assign(sko(1), 1)\}.$

Step 2: $dom(2).\quad arg(2, 2).$
$\{assign(sko(1), 2)\}.$
$\{assign(sko(2), 1)\}.$
$\{assign(sko(2), 2)\}.$

Step 3: $dom(3).\quad arg(3, 3).$
$\{assign(sko(1), 3)\}.$
$\{assign(sko(2), 3)\}.$
$\{assign(sko(3), 1)\}.$
$\{assign(sko(3), 2)\}.$
$\{assign(sko(3), 3)\}.$

Given that the body of (8) only relies on facts (over predicates dom and arg), its ground instances can be evaluated and then be reduced: if a ground body holds, the corresponding (choice) head is generated in a step; otherwise, the ground rule is trivially satisfied and needs not be considered any further. Hence, all rules shown above have an empty body after grounding. Notice, for example, that rule $\{assign(sko(1), 1)\}.$ is generated in the first step, while it is not among the new ground rules in the second and third step.

Finally, a mapping of terms to domain elements must be unique and total. To this end, translation-based FMC approaches add uniqueness and totality axioms for each term to an instantiated theory. In iASP, such requirements can be encoded as follows:

$$\leftarrow assign(T, D), assign(T, k), D < k. \quad (9)$$
$$\leftarrow cons(T), \{assign(T, D) : dom(D)\}0. \quad (10)$$
$$\leftarrow func(T), \{assign(T, D) : dom(D)\}0. \quad (11)$$

While the integrity constraint in (9) forces the mapping of each term to be unique, the ones in (10) and (11) stipulate each term to be mapped to some domain element. However, since the domain grows over incremental steps and new facts are added for predicate dom, ground instances of (10) and (11) are only valid in the step where they are generated. Hence, the integrity constraints in (10) and (11) belong to the *volatile* part of our iASP program.

3.2 Interpreting Clauses

To evaluate an input theory, we also need to interpret its predicates. The following rules allow for interpreting the predicates p and q in (2):

$$\{p(X)\} \leftarrow dom(X), 1\{arg(X, k)\}. \\ \{q(X, Y)\} \leftarrow dom(X), dom(Y), \\ 1\{arg(X, k), arg(Y, k)\}. \quad (12)$$

As discussed above, requiring the cardinality constraints in bodies to hold guarantees that (relevant) instances are particular to each incremental step. Also note that, unlike constants and functions, we do not reify predicates, as assigning a truth value can be expressed more naturally without it.

Following [Simons *et al.*, 2002], the basic idea of encoding a (flat) clause is to represent it by an integrity constraint containing the complements of the literals in the clause. However, clauses may contain equality literals, of the form $(X = t)$ or $(X \neq t)$ for some non-variable term t. The complements of such literals are given by $not\ assign(t, X)$ and $assign(t, X)$, respectively.

For our running example, the clauses in (2) give rise to the following integrity constraints:

$$\leftarrow not\ p(X), assign(a, X), dom(X), 1\{arg(X, k)\}. \\ \leftarrow q(X, X), dom(X), 1\{arg(X, k)\}. \\ \leftarrow p(X), not\ q(X, Y), assign(sko(X), Y), \\ dom(X), dom(Y), 1\{arg(X, k), arg(Y, k)\}. \quad (13)$$

Note that we use the same technique as before to separate the (relevant) instances obtained at each incremental step.

3.3 Extracting Models

The rules that represent the mapping of terms to domain elements (described in Section 3.1) along with those representing satisfiability of flat clauses (described in Section 3.2) constitute our iASP program for FMC. To compute an answer set, the incremental parameter k is increased by one at each step. This corresponds to the addition of a new domain element. If an answer set is found at step i, it means that the input theory has a model over a domain of size i.

For the iASP program encoding the theory in (2), composed of the rules in (4–13), the following answer set is obtained in the second incremental step:

$$\left\{ \begin{array}{l} dom(1), dom(2), arg(1, 1), arg(2, 2), \\ cons(a), assign(a, 1), \\ func(sko(1)), assign(sko(1), 2), \\ func(sko(2)), assign(sko(2), 2), \\ p(1), q(1, 2) \end{array} \right\}$$

The corresponding model over domain $\{1, 2\}$ is the one shown in (3).

3.4 Breaking Symmetries

In view of the fact that interpretations obtained by permuting domain elements are isomorphic, an input theory can have many symmetric models. For example, an alternative model to the one in (3) can easily be obtained by swapping domain

elements 1 and 2. Such symmetries tend to degrade the performance of FMC systems. Hence, systems based on the constraint solving approach, such as *Sem* and *Falcon*, apply variants of a dynamic symmetry breaking technique called least number heuristic [Zhang, 1996]. Translation-based systems, such as *Paradox* and *FM-Darwin* [Baumgartner *et al.*, 2009], statically break symmetries by narrowing how terms can be mapped to domain elements.

Our approach to symmetry breaking is also a static one that aims at reducing the possibilities of mapping constants to domain elements. To this end, we use the technique described in [Claessen and Sörensson, 2003; Baumgartner *et al.*, 2009], fixing an order of the constants in the input by uniquely assigning a rank in $[1, n]$, where n is the total number of constants, to each of them. Given such a ranking in terms of facts over predicate $order$, we can replace the rule in (5) with:

$$\{assign(T, k)\} \leftarrow cons(T), order(T, O), k \leq O.$$

For instance, if the order among three constants, c_1, c_2, and c_3, is given by facts $order(c_i, i)$. for $i \in \{1, 2, 3\}$, the following instances of the above rule are generated in and accumulated over three incremental steps:

Step 1:	Step 2:	Step 3:
$\{assign(c_1, 1)\}$.		
$\{assign(c_2, 1)\}$.	$\{assign(c_2, 2)\}$.	
$\{assign(c_3, 1)\}$.	$\{assign(c_3, 2)\}$.	$\{assign(c_3, 2)\}$.

That is, while all three constants can be mapped to the first domain element, c_1 cannot be mapped to the second one, and only c_3 can be mapped to the third one. Additional rules restricting the admissible mappings of constants to a "canonical form" [Claessen and Sörensson, 2003] are provided in a forthcoming journal paper extending [Gebser *et al.*, 2010].

Finally, we note that our iASP encoding of the theory in (2) yields 10 answer sets in the second incremental step. If we apply the described symmetry breaking, it disallows mapping the single constant a to the second domain element, which prunes 5 of the 10 models. Although our simple technique can in general not break all symmetries related to the mapping of terms because it does not incorporate functions, the experiments in Section 5 demonstrate that it may nonetheless lead to significant performance gains. For the special case of unary functions, an extension [Claessen and Sörensson, 2003] is implemented in *Paradox*; with *FM-Darwin*, it has not turned out to be more effective than symmetry breaking for only constants [Baumgartner *et al.*, 2009].

4 System

We use *FM-Darwin* to read an input in TPTP format, a format for first-order theories widely used within the ATP community, to clausify it if needed, and to flatten the clauses at hand. Additionally, *FM-Darwin* applies some input optimizations, such as renaming deep ground subterms and splitting, to avoid the generation of flat clauses with many variables [Baumgartner *et al.*, 2009].

Given flat clauses, we further apply the transformations described in Section 3.1 and 3.2 to generate an iASP program. To this end, we implemented a compiler called *fmc2iasp*[1], written in Python. It outputs the rules that are specific to an input theory, while the theory-independent rules in (5), (6), and (9–11) are provided separately. This allows us to test encoding variants, for instance, altering symmetry breaking, without modifying *fmc2iasp*. Finally, we use *iClingo* to incrementally ground the obtained iASP program and to search for answer sets representing finite models of the input theory.

[1] `http://potassco.sourceforge.net/`

5 Experiments

Our experiments consider the following systems: *iClingo* (2.0.5), *Clingo* (2.0.5), *Paradox* (3.0), *FM-Darwin* (1.4.5), and *Mace4* (2009-11A). While *Paradox* and *FM-Darwin* are based on the translational approach to FMC, *Mace4* applies the constraint solving approach. For *iClingo* and *Clingo*, we used command line switch `--heuristic=VSIDS`, as it improved search performance.[2] All experiments were performed on a 3.4GHz Intel Xeon machine running Linux, imposing 300 seconds as time and 2GB as memory limit.

FMC instances stem from the FNT division of CADE's 2009 and 2010 ATP competitions. The instances in this division are satisfiable and suitable for evaluating FMC systems, among which *Paradox* won the first place in both years' competitions. The considered problem domains are: computing theory (COM), common-sense reasoning (CSR), geography (GEG), geometry (GEO), graph theory (GRA), groups (GRP), homological algebra (HAL), knowledge representation (KRS), lattices (LAT), logic calculi (LCL), medicine (MED), management (MGT), miscellaneous (MSC), natural language processing (NLP), number theory (NUM), planning (PLA), processes (PRO), rings in algebra (RNG), software verification (SWV), syntactic (SYN).[3]

Table 1 and 2 show benchmark results for each of the problem domains. Column # displays how many instances of a problem domain belong to the test suite. For each system and problem domain, average run time in seconds is taken over the solved instances; their number is given in parentheses.[4] A dash in an entry means that a system could not solve any instance of the corresponding problem domain within the run time and memory limits. For each system, the last row shows its average run time over all solved instances and provides their number in parentheses. The evaluation criteria in CADE competitions are first number of solved instances and then average run time as tie breaker.

In Table 1, we see that *Mace4* and *FM-Darwin* solved 50 and 82 instances, respectively, out of the 99 instances in total. *Paradox*, the winner of the FNT division in CADE's 2009 ATP competition, solved 92 instances in 6.05 seconds on average. While the version of our system not using symmetry breaking (described in Section 3.4), denoted by *iClingo (2)*, solved two instances less, the one with symmetry breaking, denoted by *iClingo (1)*, also solved 92 instances. As it spent only 2.29 seconds on average, according to the CADE criteria, our system slightly outperformed *Paradox*. For assess-

[2] Note that *Minisat*, used internally by *Paradox*, also applies VSIDS as decision heuristic [Eén and Sörensson, 2004].

[3] `http://www.cs.miami.edu/~tptp/`

[4] Run time results of our system include the time for preprocessing by *FM-Darwin*.

Benchmark	#	*iClingo (1)*	*iClingo (2)*	*Clingo*	*Paradox*	*FM-Darwin*	*Mace4*
CSR	1	2.87 (1)	2.30 (1)	6.26 (1)	—	20.56 (1)	—
GEG	1	—	—	—	230.36 (1)	—	—
GEO	12	0.08 (12)	0.09 (12)	0.11 (12)	0.08 (12)	0.09 (12)	0.04 (12)
GRA	2	3.44 (1)	—	12.78 (1)	0.49 (1)	—	—
GRP	1	4.25 (1)	216.96 (1)	6.31 (1)	0.63 (1)	—	0.28 (1)
HAL	2	2.52 (2)	2.46 (2)	2.94 (2)	0.67 (2)	11.84 (1)	—
KRS	6	0.14 (6)	0.16 (6)	0.27 (6)	0.11 (6)	30.87 (6)	0.03 (4)
LAT	5	0.10 (5)	0.11 (5)	0.13 (5)	0.12 (5)	0.08 (5)	0.04 (5)
LCL	17	8.62 (17)	9.50 (17)	10.86 (17)	3.70 (17)	1.65 (17)	5.10 (8)
MGT	4	0.08 (4)	0.09 (4)	0.10 (4)	0.06 (4)	0.12 (4)	1.09 (4)
MSC	3	4.70 (2)	0.23 (1)	12.58 (2)	122.56 (2)	0.19 (1)	—
NLP	9	1.66 (9)	2.03 (9)	3.17 (9)	0.24 (9)	0.26 (8)	22.07 (1)
NUM	1	0.19 (1)	0.24 (1)	0.28 (1)	0.27 (1)	0.11 (1)	201.51 (1)
PRO	9	1.09 (9)	9.03 (9)	2.02 (9)	0.34 (9)	0.77 (9)	31.53 (7)
SWV	8	0.15 (4)	0.14 (4)	0.20 (4)	0.13 (4)	44.84 (5)	0.04 (2)
SYN	18	0.59 (18)	0.57 (18)	0.72 (18)	0.40 (18)	3.84 (12)	0.68 (5)
Total	99	2.29 (92)	5.55 (90)	3.32 (92)	6.05 (92)	6.43 (82)	9.88 (50)

Table 1: Benchmark results for problems in the FNT division of CADE's 2009 ATP competition.

Benchmark	#	*iClingo (1)*	*iClingo (2)*	*Clingo*	*Paradox*	*FM-Darwin*	*Mace4*
COM	2	0.21 (2)	0.28 (2)	0.39 (2)	0.23 (2)	0.09 (2)	0.05 (2)
GEO	1	0.06 (1)	0.11 (1)	0.07 (1)	0.05 (1)	0.07 (1)	0.03 (1)
GRA	2	101.08 (1)	—	207.62 (1)	8.21 (2)	15.56 (1)	207.37 (1)
GRP	1	0.08 (1)	0.12 (1)	0.12 (1)	0.03 (1)	0.04 (1)	0.03 (1)
HAL	1	2.48 (1)	2.37 (1)	2.73 (1)	0.33 (1)	—	—
KRS	1	0.09 (1)	0.11 (1)	0.12 (1)	0.05 (1)	0.05 (1)	0.03 (1)
LCL	52	5.43 (42)	5.25 (41)	4.57 (40)	3.81 (49)	6.22 (42)	8.08 (19)
MED	1	0.11 (1)	0.11 (1)	0.15 (1)	0.08 (1)	0.06 (1)	0.02 (1)
MSC	1	—	—	—	—	—	—
NLP	52	41.59 (21)	38.29 (22)	35.49 (22)	0.21 (32)	15.85 (49)	3.71 (2)
NUM	9	0.17 (9)	0.19 (9)	0.24 (9)	0.19 (9)	0.10 (9)	22.61 (8)
PLA	4	70.41 (4)	0.27 (3)	0.52 (3)	0.20 (4)	0.23 (4)	0.26 (4)
RNG	2	0.19 (2)	0.26 (2)	0.29 (2)	0.23 (2)	0.20 (2)	287.29 (1)
SWV	2	—	—	—	—	—	—
SYN	12	7.13 (12)	7.18 (12)	7.96 (12)	5.46 (12)	68.76 (9)	—
Total	143	16.07 (98)	11.98 (96)	13.28 (96)	2.38 (117)	13.73 (122)	20.43 (41)

Table 2: Benchmark results for problems in the FNT division of CADE's 2010 ATP competition, restricted to instances not already used in 2009.

ing the advantages due to incremental grounding and solving, we also ran *Clingo*, performing iterative deepening search by successively grounding and solving our iASP encoding for fixed domains of increasing size. The average run time achieved with *Clingo*, 3.32 seconds, is substantially greater than the one of *iClingo (1)*; the gap becomes more apparent the more domain elements (not shown in Table 1) are needed.

Although there are 200 instances in the FNT division of CADE's 2010 ATP competition, we only show 143 of them in Table 2. (The 57 remaining instances were already used in 2009.) *Mace4* solved 41 of these 143 instances. *Paradox*, the winner of the FNT division also in 2010, solved 117 instances in 2.38 seconds on average. When considering all 200 instances used in 2010, *Paradox* solved 167 of them in 2.55 seconds on average, and *FM-Darwin* also solved 167 instances, but in 12.99 seconds on average. In fact, *FM-Darwin* solved the most instances in Table 2, 122 out of 143. Like with the results shown in Table 1, the version of our system with symmetry breaking, *iClingo (1)*, solved two more instances than the one without symmetry breaking, *iClingo (2)*. Furthermore, the advantages due to incremental grounding and solving are more apparent in Table 2, viewing that *Clingo* solved two instances less than *iClingo (1)*, 96 compared to 98. We however observe that *iClingo* cannot keep step with *Paradox* and *FM-Darwin* on the instances in Table 2; possible reasons are elaborated on next.

A general problem with the translational approach is that flattening may increase the number of variables in a clause, which can deteriorate grounding performance. We can observe this when comparing the results of *FM-Darwin* on the NLP domain in Table 2 with those of *iClingo* and *Paradox*: *FM-Darwin* solved 17 instances more than *Paradox* and 28 more than *iClingo (1)*. Although *FM-Darwin* also pursues a translational approach, it represents subproblems by function-free first-order clauses and uses *Darwin*, not relying on grounding [Baumgartner *et al.*, 2009], to solve

them. An instance of the SWV group (instance SWV484+2) in Table 1 provides an extreme example for the infeasibility of grounding: it includes predicates of arity 34 and is solved only by *FM-Darwin*. Furthermore, in comparison to *iClingo*, sort inference [Claessen and Sörensson, 2003; Baumgartner *et al.*, 2009] promotes *Paradox* and *FM-Darwin* on the instances of NLP in Table 2. This shows that there is still potential to improve our iASP approach to FMC. On the other hand, for the instances of CSR and MSC in Table 1, we speculate that clausification and further preprocessing steps of *Paradox* may be the cause for its deteriorated performance. In fact, given the proximity of the solvers used internally by *iClingo* and *Paradox*, major performance differences are most likely due to discrepancies in grounding.

6 Discussion

We presented an efficient yet transparent approach to computing finite models of first-order theories by means of ASP. Our approach takes advantage of an incremental extension of ASP that allows us to consecutively search for models with given domain size by incrementing the corresponding parameter in an iASP encoding. Its declarative nature makes our approach easily modifiable and leaves room for further improvements. Still, our approach is rather competitive and has even a slight edge on the winner of the FNT division of CADE's 2009 ATP competition on the respective benchmark collection.

Related works include [Sabuncu and Alpaslan, 2007], where FMC systems were used for computing the answer sets of tight[5] logic programs in order to circumvent grounding. In [Lierler and Lifschitz, 2008], a special class of first-order formulas, called Effectively Propositional (EPR) formulas, was addressed via ASP; application domains like planning and bounded model checking have been encoded by EPR formulas and were successfully tackled by means of FMC [Pérez and Voronkov, 2008; 2007].

Acknowledgments. This work was supported by the German Science Foundation (DFG) under grants SCHA 550/8-1/2. We are grateful to the anonymous referees of previously submitted workshop, conference, and journal versions as well as of this paper for their helpful comments.

References

[Baral, 2003] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Baumgartner *et al.*, 2009] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 7(1):58–74, 2009.

[Biere *et al.*, 2009] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS, 2009.

[Caferra *et al.*, 2004] R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*. Kluwer, 2004.

[5]Tight programs are free of recursion through positive literals (cf. [Baral, 2003]).

[Claessen and Sörensson, 2003] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. *Proceedings of the Workshop on Model Computation (MODEL'03)*. 2003.

[Eén and Sörensson, 2004] N. Eén and N. Sörensson. An extensible SAT-solver. *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 502–518. Springer, 2004.

[Gebser *et al.*, 2008] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. *Proceedings of the 24th International Conference on Logic Programming (ICLP'08)*, 190–205. Springer, 2008.

[Gebser *et al.*, 2010] M. Gebser, O. Sabuncu, and T. Schaub. An incremental answer set programming based system for finite model computation. *Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA'10)*, 169–181. Springer, 2010.

[Gebser *et al.*, 2011] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, 345–351. Springer, 2011.

[Lierler and Lifschitz, 2008] Y. Lierler and V. Lifschitz. Logic programs vs. first-order formulas in textual inference. Unpublished draft, 2008.

[McCune, 1994] W. McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical Report ANL/MCS-TM-194, Argonne National Laboratory, 1994.

[Pérez and Voronkov, 2007] J. Navarro Pérez and A. Voronkov. Encodings of bounded LTL model checking in effectively propositional logic. *Proceedings of the 21st International Conference on Automated Deduction (CADE'07)*, 346–361. Springer, 2007.

[Pérez and Voronkov, 2008] J. Navarro Pérez and A. Voronkov. Planning with effectively propositional logic. *Volume in Memoriam of Harald Ganzinger*. Springer, To appear.

[Robinson and Voronkov, 2001] J. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001.

[Sabuncu and Alpaslan, 2007] O. Sabuncu and F. Alpaslan. Computing answer sets using model generation theorem provers. *Proceedings of the 4th International Workshop on Answer Set Programming (ASP'07)*, 225–240. 2007.

[Simons *et al.*, 2002] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[Tammet, 2003] T. Tammet. Finite model building: Improvements and comparisons. *Proceedings of the Workshop on Model Computation (MODEL'03)*. 2003.

[Zhang, 1996] J. Zhang. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, 17(1):1–22, 1996.