An Incremental Answer Set Programming Based System for Finite Model Computation

Martin Gebser[1], Orkunt Sabuncu[1], and Torsten Schaub[1,2]

[1] Universität Potsdam
{gebser,orkunt,torsten}@cs.uni-potsdam.de
[2] Simon Fraser University, Canada, and Griffith University, Australia

Abstract. We address the problem of Finite Model Computation (FMC) of first-order theories and show that FMC can efficiently and transparently be solved by taking advantage of a recent extension of Answer Set Programming (ASP), 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 implemented a system based on the iASP solver *iClingo* and demonstrate its competitiveness by showing that it slightly outperforms the winner of the FNT division of CADE's Automated Theorem Proving (ATP) competition.

1 Introduction

While Finite Model Computation (FMC;[1]) constitutes an established research area in the field of Automated Theorem Proving (ATP;[2]), Answer Set Programming (ASP;[3]) 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;[4]).

Approaches to FMC for first-order theories [5,6] fall in two major categories, translational and constraint solving approaches. In translational approaches [7,8], 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;[9]) 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 [10,11], 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 [6].

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,

T. Janhunen and I. Niemelä (Eds.): JELIA 2010, LNAI 6341, pp. 169–181, 2010.

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* [4] and compared its performance to various FMC systems. To this end, we used the problems from the FNT division of last year's CADE ATP competition. The results demonstrate the efficiency of our system. *iClingo* solved the same number of problems as *Paradox* [8] in approximately half of its run time on average. Note that *Paradox* won first places in the FNT division in 2007, 2008, and 2009.

The paper is organized as follows. The next section introduces basic concepts about the translational approach to FMC and about iASP. Section 3 describes our incremental 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. An input first-order theory along with logic programs, as used by our FMC system based on iASP, are provided in appendixes.

2 Background

We assume the reader to be familiar with the terminology and basic definitions of first-order logic and ASP. In what follows, we thus focus on the introduction of concepts needed in the remainder of this paper.

In our method, 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* [7,8,6], 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 grounders *lparse* [12] and *gringo* [13], providing normal and choice rules as well as cardinality and integrity constraints. As usual, rules with variables are regarded as representatives for all respective ground instances. Beyond that, our approach makes use of iASP [4] 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. In view of this, 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

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

has an answer set for some (minimum) integer $i \geq 1$. 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. Based on this, 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 presents an encoding optimization by means of symmetry breaking. Finally, we show the soundness and completeness of our approach in Section 3.5.

Throughout this section, we illustrate our approach on a running example. Assume that the following first-order theory is given as starting point:

$$\begin{array}{l} p(a) \\ (\forall X)\ \neg q(X, X) \\ (\forall X)\ (p(X) \rightarrow (\exists Y)\ q(X, Y)). \end{array} \tag{2}$$

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:

$$\begin{array}{l} p(X) \vee (X \neq a) \\ \neg q(X, X) \\ \neg p(X) \vee q(X, Y) \vee (Y \neq sko(X)). \end{array} \tag{3}$$

Such flat clauses form the basis for our iASP encoding. Before we present it, note that the theory in (3) has a model I over domain $\{1, 2\}$ given by:

$$\begin{array}{rl} a^I & = 1 \\ sko^I & = \{1 \mapsto 2, 2 \mapsto 2\} \\ p^I & = \{1\} \\ q^I & = \{(1, 2)\}. \end{array} \tag{4}$$

Importantly, I is also a model of the original theory in (2), 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 (3) gives rise to the following fact:

$$cons(a). \tag{5}$$

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 variable 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:

$$\{assign(T, k)\} \leftarrow cons(T). \tag{6}$$

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 (non-zero arity) functions grow when k increases. To deal with this, we first declare auxiliary facts to represent available domain elements:

$$dom(k). \qquad arg(k, k). \tag{7}$$

Predicates dom and arg are then used to qualify the arguments of an n-ary function f in the following rule:

$$\begin{aligned} func(f(X_1, \ldots, X_n)) \leftarrow\ & dom(X_1), \ldots, dom(X_n), \\ & 1\{arg(X_1, k), \ldots, arg(X_n, k)\}. \end{aligned} \tag{8}$$

The cardinality constraint $1\{arg(X_1, k), \ldots, arg(X_n, k)\}$ stipulates at least one of the arguments $X_1, \ldots, X_n$ of f to be k. As in (6), though using a different methodology, this makes sure that the (relevant) instances 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. For the unary function sko in (3), applying the described scheme leads to the following rule:

$$func(sko(X)) \leftarrow dom(X), 1\{arg(X, k)\}. \tag{9}$$

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. The following (choice) rule encodes mappings via an n-ary function f:

$$\begin{aligned} \{assign(f(X_1, \ldots, X_n), Y)\} \leftarrow\ & dom(X_1), \ldots, dom(X_n), dom(Y), \\ & 1\{arg(X_1, k), \ldots, arg(X_n, k), arg(Y, k)\}. \end{aligned} \tag{10}$$

For instance, the rule encoding mappings via unary function sko is as follows:

$$\{assign(sko(X), Y)\} \leftarrow dom(X), dom(Y), 1\{arg(X, k), arg(Y, k)\}. \tag{11}$$

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 (7) and (11) generated in and accumulated over three incremental steps:

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

Given that the body of (11) 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:

$$\leftarrow assign(T,D), assign(T,k), D < k. \tag{12}$$

$$\leftarrow cons(T), \{assign(T,D) : dom(D)\}0. \tag{13}$$

$$\leftarrow func(T), \{assign(T,D) : dom(D)\}0. \tag{14}$$

While the integrity constraint in (12) forces the mapping of each term to be unique, the ones in (13) and (14) 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 (13) and (14) are only valid in the step where they are generated. Hence, the integrity constraints in (13) and (14) 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. To this end, we include a rule of the following form for every n-ary predicate p in our iASP program:

$$\begin{aligned}\{p(X_1,\ldots,X_n)\} \leftarrow\ & dom(X_1),\ldots,dom(X_n),\\ & 1\{arg(X_1,k),\ldots,arg(X_n,k)\}.\end{aligned} \tag{15}$$

As discussed above, requiring $1\{arg(X_1,k),\ldots,arg(X_n,k)\}$ to hold guarantees that (relevant) instances are particular to each incremental step. The only exception to this

is $n = 0$ (a predicate p of arity zero), in which case the rule $\{p\}.$ belongs to the *static* part of our program. Also note that, unlike constants and functions, we do not reify predicates, as assigning a truth value can be expressed more naturally without it. For example, the following rules allow for interpreting the predicates p and q in (3):

$$\begin{aligned} \{p(X)\} &\leftarrow dom(X), 1\{arg(X,k)\}. \\ \{q(X,Y)\} &\leftarrow dom(X), dom(Y), 1\{arg(X,k), arg(Y,k)\}. \end{aligned} \tag{16}$$

Following [14], 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 $(X = Y)$ or $(X \neq Y)$, where at least one of the terms X and Y is a variable, and so we also need to consider complements of such literals. W.l.o.g., we below assume that the left-hand side of every equality literal is a variable, while the right-hand side is either a variable or a non-variable term. In view of this convention, we define the encoding $\overline{L}$ of the complement of a (classical or equality) literal L as follows:

$$\overline{L} = \begin{cases} not\ p(X_1,\ldots,X_n) & \text{if } L = p(X_1,\ldots,X_n) \\ p(X_1,\ldots,X_n) & \text{if } L = \neg p(X_1,\ldots,X_n) \\ not\ assign(t,X) & \text{if } L = (X = t) \text{ for some non-variable term } t \\ assign(t,X) & \text{if } L = (X \neq t) \text{ for some non-variable term } t \\ X \neq Y & \text{if } L = (X = Y) \text{ for some variable } Y \\ X = Y & \text{if } L = (X \neq Y) \text{ for some variable } Y. \end{cases}$$

Observe that the first two cases refer to the interpretation of a predicate p, the third and the fourth to the mapping of non-variable terms to domain elements, and the last two to built-in comparison operators of grounders like *lparse* and *gringo*.

With the complements of literals at hand, we can now encode a flat clause containing literals $L_1,\ldots,L_m$ and variables $X_1,\ldots,X_n$ by an integrity constraint as follows:

$$\leftarrow \overline{L}_1,\ldots,\overline{L}_m, dom(X_1),\ldots,dom(X_n), 1\{arg(X_1,k),\ldots,arg(X_n,k)\}. \tag{17}$$

Note that we use the same technique as before to separate the (relevant) instances obtained at each incremental step. For our running example, the clauses in (3) give rise to the following integrity constraints:

$$\begin{aligned} &\leftarrow not\ p(X), assign(a,X), dom(X), 1\{arg(X,k)\}. \\ &\leftarrow q(X,X), dom(X), 1\{arg(X,k)\}. \\ &\leftarrow p(X), not\ q(X,Y), assign(sko(X),Y), \\ &\quad dom(X), dom(Y), 1\{arg(X,k), arg(Y,k)\}. \end{aligned} \tag{18}$$

While the first two integrity constraints each contribute a single instance at an incremental step, $(2 * k) - 1$ instances are obtained for the third one.

Although they are unlikely to occur in first-order theories, *propositional* clauses without variables and equality literals require a slightly different treatment. For a propositional clause containing (classical) literals $L_1,\ldots,L_m$, instead of (17), we include the following simpler integrity constraint in the *static* part of our iASP program:

$$\leftarrow \overline{L}_1,\ldots,\overline{L}_m. \tag{19}$$

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 variable k is increased by one at each step. This corresponds to the addition of a new domain element. If an answer set is found in a step i, it means that the input theory has a model over a domain of size i. In fact, from an answer set A of our iASP program, a model I of the input theory over domain $\{d \mid dom(d) \in A\}$ is extracted as follows:

$$\begin{aligned} c^I &= d \text{ where } cons(c), assign(c,d) \in A, \\ f^I &= \{(d_1,\ldots,d_n) \mapsto d \mid assign(f(d_1,\ldots,d_n),d) \in A\}, \\ p^I &= \{(d_1,\ldots,d_n) \mid p(d_1,\ldots,d_n) \in A\}. \end{aligned}$$

For the iASP program encoding the theory in (3), composed of the rules in (5–7, 9, 11–14, 16, 18), 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 (4).

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 (4) 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 [11]. Translation-based systems, such as *Paradox* and *FM-Darwin*, staticly 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 [8,15], 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 (6) with:

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

For instance, if the set of constants is $\{c_1,c_2,c_3\}$ and the order 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
$\{assign(c_1,1)\}.$		
$\{assign(c_2,1)\}.$	$\{assign(c_2,2)\}.$	
$\{assign(c_3,1)\}.$	$\{assign(c_3,2)\}.$	$\{assign(c_3,3)\}.$

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.

Finally, we note that our iASP encoding of the theory in (3) 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. Unlike with constants, given a priori, additionally incorporating functions into our approach to symmetry breaking would require the extension of predicate $order$ to newly composed functional terms in each incremental step. For the special case of unary functions, such an extension [8] is implemented in *Paradox*; with *FM-Darwin*, it has not turned out to be more effective than symmetry breaking for only constants [15].

3.5 Soundness and Completeness

Before stating our theorem, we first define the parameterized domain description formed for a set $\mathcal{T}$ of flat clauses. The signature $\langle \mathcal{F}_0, \mathcal{F}, \mathcal{P}_0, \mathcal{P} \rangle$ of $\mathcal{T}$ is built from a set $\mathcal{F}_0$ of *constants*, a set $\mathcal{F}$ of (non-zero arity) *functions*, a set $\mathcal{P}_0$ of *zero arity predicates*, and a set $\mathcal{P}$ of *non-zero arity predicates*. For $\mathcal{T}$, we then form the parameterized domain description (B, P, Q) in the following way:

$$
\begin{aligned}
B &= \big\{\ cons(c).\mid c \in \mathcal{F}_0\ \big\} \cup \big\{\ \{p\}.\mid p \in \mathcal{P}_0\ \big\} \cup \Pi^{\mathcal{T}_0}\ ,\\
P &= \big\{\ dom(k).\quad arg(k,k).\quad \{assign(T,k)\} \leftarrow cons(T).\\
&\qquad \leftarrow assign(T,D), assign(T,k), D<k.\ \big\} \cup \Pi^{\mathcal{F}} \cup \Pi^{\mathcal{P}} \cup \Pi^{\mathcal{T}}\ ,\ \text{and}\\
Q &= \big\{ \leftarrow cons(T), \{assign(T,D) : dom(D)\}0.\\
&\qquad \leftarrow func(T), \{assign(T,D) : dom(D)\}0.\ \big\}\ ,
\end{aligned}
$$

where $\Pi^{\mathcal{F}}$ contains rules of form (8) and form (10) for each function $f \in \mathcal{F}$, $\Pi^{\mathcal{P}}$ contains a rule of form (15) for each predicate $p \in \mathcal{P}$, $\Pi^{\mathcal{T}}$ contains a rule of form (17) for each non-propositional clause in $\mathcal{T}$, and $\Pi^{\mathcal{T}_0}$ contains a rule of form (19) for each propositional clause in $\mathcal{T}$. With these concepts at hand, we are ready to formulate the soundness and completeness of our approach.

Theorem 1. *Let $\mathcal{T}$ be a set of flat clauses and (B, P, Q) the parameterized domain description for $\mathcal{T}$. Then, the logic program $R[k/i]$, as defined in (1), has an answer set for some positive integer i iff $\mathcal{T}$ has a finite model over a domain of size i.*

Note that the theorem still applies when including symmetry breaking, as described in the previous section, in view of the fact that it may eliminate some isomorphic models, but not all of them.

4 System

We use *FM-Darwin* to read an input in TPTP format, a format for first-order theories widely used within the community of ATP, to clausify it if needed, and to flatten the clauses at hand. Additionally, *FM-Darwin* applies some input optimizations before flattening, such as renaming deep ground subterms to avoid the generation of flat clauses with many variables [15]. For obtaining flat clauses from an input theory specified in a file `tptp_input.p`, *FM-Darwin* is invoked as follows:

```
darwin -fd true -pfdp Exit tptp_input.p
```

Having an input in terms of flat clauses, we can 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 (6), (7), and (12–14) are provided in a separate file. This separation allows us to test encoding variants without changing *fmc2iasp*, for instance, the symmetry breaking described in Section 3.4. Finally, we use *iClingo* to incrementally ground the obtained iASP program and to search for answer sets representing finite models of the input theory. Provided that `fmc.lp` is the file containing theory-independent rules, the following command-line call is used for FMC:

```
darwin -fd true -pfdp Exit tptp_input.p | fmc2iasp.py |
cat fmc.lp - | iclingo
```

5 Experiments

We 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] Our experiments have been 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 (First-order form Non-Theorems) division of the 2009 CADE ATP competition. The instances in this division are satisfiable and suitable for evaluating FMC systems, among which *Paradox* won the first place. The considered problem domains are: common-sense reasoning (CSR), geography (GEG), geometry (GEO), graph theory (GRA), groups (GRP), homological algebra (HAL), knowledge representation (KRS), lattices (LAT), logic calculi (LCL), management (MGT), miscellaneous (MSC), natural language processing (NLP), number theory (NUM), processes (PRO), software verification (SWV), syntactic (SYN).[3]

Table 1 shows 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

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

[2] Note that *Minisat*, used internally by *Paradox*, also applies VSIDS as decision heuristic [16].

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

Table 1. Benchmark results for problems in the FNT division of the 2009 CADE competition

Benchmark	#	*iClingo (1)*	*iClingo (2)*	*Clingo*	*Paradox*	*FM-Darwin*	*Mace4*
CSR	1	2.28 (1)	2.19 (1)	4.20 (1)	—	20.96 (1)	—
GEG	1	—	—	—	229.12 (1)	—	—
GEO	12	0.07 (12)	0.06 (12)	0.09 (12)	0.08 (12)	0.09 (12)	0.02 (12)
GRA	2	3.48 (1)	—	12.33 (1)	0.50 (1)	—	—
GRP	1	5.58 (1)	215.62 (1)	78.90 (1)	0.65 (1)	—	0.26 (1)
HAL	2	2.35 (2)	2.40 (2)	2.68 (2)	0.68 (2)	11.43 (1)	—
KRS	6	0.13 (6)	0.13 (6)	0.24 (6)	0.20 (6)	30.76 (6)	0.02 (4)
LAT	5	0.09 (5)	0.09 (5)	0.12 (5)	0.10 (5)	0.07 (5)	0.03 (5)
LCL	17	8.71 (17)	9.36 (17)	11.44 (17)	3.72 (17)	1.56 (17)	5.07 (8)
MGT	4	0.06 (4)	0.06 (4)	0.08 (4)	0.07 (4)	0.12 (4)	0.98 (4)
MSC	3	9.31 (2)	0.20 (1)	16.51 (2)	121.03 (2)	0.19 (1)	—
NLP	9	1.60 (9)	1.98 (9)	3.06 (9)	0.25 (9)	0.28 (8)	22.19 (1)
NUM	1	0.19 (1)	0.19 (1)	0.26 (1)	0.26 (1)	0.13 (1)	202.45 (1)
PRO	9	1.09 (9)	8.99 (9)	1.91 (9)	0.37 (9)	0.78 (9)	31.56 (7)
SWV	8	0.13 (4)	0.12 (4)	0.17 (4)	0.16 (4)	45.13 (5)	0.03 (2)
SYN	18	0.56 (18)	0.57 (18)	0.68 (18)	0.40 (18)	3.88 (12)	0.66 (5)
Total	99	2.39 (92)	5.49 (90)	4.24 (92)	6.02 (92)	6.43 (82)	9.88 (50)

number is given in parentheses. 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 the 2009 CADE competition, solved 92 instances in 6.02 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 2.39 seconds on average, according to the CADE criteria, our system slightly outperformed *Paradox*. For assessing 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*, 4.24 seconds, is substantially greater than the one of *iClingo (1)*, and the gap becomes more apparent the more domain elements are needed.

However, 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 observed this clearly for the instance in the GEG domain, where the flat clauses contain about seven variables. While *iClingo* could not ground the resulting iASP program within the given limits, *Paradox* still solved it (in 229.12 seconds). The fact that the underlying first-order theory has many sorts, so that sort inference [8] of *Paradox* helps, shows that there is still potential to improve the translational approach via iASP. On the other hand, for the instances in groups CSR and MSC, we speculate

that clausification and further preprocessing steps of *Paradox* may be the cause for its deteriorated performance.

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 the iASP encoding. The declarative nature of our approach makes it easily modifiable and leaves room for further improvements. Moreover, our approach is rather competitive and has even a slight edge on the hitherto leading system for FMC. Finally, our approach complements the work in [17], where FMC systems were used for computing the answer sets of tight[4] logic programs in order to circumvent grounding.

In [18], a special class of first-order formulas, called Effectively Propositional (EPR) formulas, was addressed via ASP. EPR formulas must not contain function symbols in their clause forms. Although our approach takes more general input than this, it can currently not decide EPR formulas. To this end, we had either to extract a bound on the incremental parameter to make the system halt or to provide an alternative dedicated encoding of EPR formulas. Such extensions are interesting topics for future research.

Acknowledgments. This work was supported by the German Science Foundation (DFG) under grant SCHA 550/8-1.

References

1. Caferra, R., Leitsch, A., Peltier, N.: Automated Model Building. Kluwer Academic, Dordrecht (2004)
2. Bibel, W.: Automated Theorem Proving. Vieweg (1987)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University, Cambridge (2003)
4. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 190–205. Springer, Heidelberg (2008)
5. Zhang, J., Huang, Z.: Reducing symmetries to generate easier SAT instances. Electronic Notes in Theoretical Computer Science 125(3), 149–164 (2005)
6. Tammet, T.: Finite model building: Improvements and comparisons. In: Baumgartner, P., Fermüller, C. (eds.) Proceedings of the Workshop on Model Computation — Principles, Algorithms, Applications, MODEL 2003 (2003)
7. McCune, W.: 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)
8. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: Baumgartner, P., Fermüller, C. (eds.) Proceedings of the Workshop on Model Computation — Principles, Algorithms, Applications, MODEL 2003 (2003)

[4] Tight programs are free of recursion through positive literals (cf. [3]).

9. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. IOS (2009)
10. Zhang, J., Zhang, H.: SEM: A system for enumerating models. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), pp. 298–303. Morgan Kaufmann, San Francisco (1995)
11. Zhang, J.: Constructing finite algebras with FALCON. Journal of Automated Reasoning 17(1), 1–22 (1996)
12. Syrjänen, T.: Lparse 1.0 user's manual, `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
13. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo, `http://potassco.sourceforge.net`
14. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
15. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. Journal of Applied Logic 7(1), 58–74 (2009)
16. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
17. Sabuncu, O., Alpaslan, F.: Computing answer sets using model generation theorem provers. In: Costantini, S., Watson, R. (eds.) Proceedings of the 4th International Workshop on Answer Set Programming (ASP 2007), pp. 225–240 (2007)
18. Lierler, Y., Lifschitz, V.: Logic programs vs. first-order formulas in textual inference, `http://z.cs.utexas.edu/users/ai-lab/publications_recent.php`

A Input Theory

The input theory (2), written in TPTP format, is as follows:

```
fof(1, axiom, p(a)).
fof(2, axiom, ! [X]: (~q(X,X)) ).
fof(3, axiom, ! [X]: (p(X) => (? [Y]: q(X,Y))) ).
```

B Theory-Independent iASP Program

The theory-independent program part with symmetry breaking (cf. Section 3.4), in the input language of *iClingo*, is as follows:

```
#cumulative k.

dom(k).
arg(k,k).

{ assign(T,k) } :- cons(T), order(T,O), k<=O.

:- assign(T,D), assign(T,k), D<k.

#volatile k.

:- cons(T), { assign(T,D):dom(D) } 0.
:- func(T), { assign(T,D):dom(D) } 0.
```

C Theory-Dependent iASP Program

The rules generated by *fmc2iasp* for the flat clauses in (3) are as follows:

```
#cumulative k.

% functions
func(sko(X0)) :- dom(X0), 1 { arg(X0,k) }.
{ assign(sko(X0),Y) } :- dom(X0;Y), 1 { arg(X0;Y,k) }.

% predicates
{ p(X0) } :- dom(X0), 1 { arg(X0,k) }.
{ q(X0,X1) } :- dom(X0;X1), 1 { arg(X0;X1,k) }.

% flat clauses
:- not p(X0), assign(a,X0), dom(X0), 1 { arg(X0,k) }.
:- q(X0,X0), dom(X0), 1 { arg(X0,k) }.
:- p(X0), not q(X0,X1), assign(sko(X0),X1),
   dom(X0;X1), 1 { arg(X0;X1,k) }.

#base.

cons(a).
order(a,1).

#hide.
#show assign/2.
#show q/2.
#show p/1.
```

In order to compute a finite model of (2), we can use this program concatenated with the rules from Appendix B, as it is described in Section 4.