The IDP system: a model expansion system for an extension of classical logic

Johan Wittocx, Maarten Mariën, and Marc Denecker

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{johan,maartenm,marcd}@cs.kuleuven.be

Abstract. The model expansion (MX) search problem consists of finding models of a given theory T that expand a given finite interpretation. Model expansion in classical first-order logic (FO) has been proposed as the basis for an Answer Set Programming-like declarative programming framework for solving NP problems. In this paper, we present IDP, a system for solving MX problems that integrates technology from ASP and SAT. Its strength lies both in its rich input language and its efficiency. IDP is the first model expansion system that can handle full FO, but its language extends FO with many other primitives such as inductive definitions, aggregates, quantifiers with numerical constraints, order-sorted types, arithmetic, partial functions, etc. We show that this allows for a natural, compact representation of many interesting search problems. Despite the generality of its language, our experiments show that the IDP system belongs to the most efficient ASP and MX systems.

1 Introduction

In [10], Marek and Truszczyński presented the idea to base declarative problem solving frameworks on computing *solutions* of a computational problem *as models* of a logic theory. In the last decade, several such frameworks have been developed. The first and arguably most popular among these is Answer Set Programming (ASP) [10, 17]. In ASP, the theories are normal logic programs, and the models are computed according to the stable model semantics. The ASP paradigm is supported by the existence of several efficient ASP solvers, each of them consisting of two components. The first component, called *grounder*, transforms the given logic program into a propositional one with the same stable models. The second component, called *model generator*, then computes the models of the propositional program.

Two more recent framework proposals use classical first-order logic (FO) as underlying logic [5, 16]. The main reason for this choice is the simplicity, expressivity and well-understood knowledge representation methodology of FO. Also, by staying close to classical logic, it is possible to directly use (extensions of) SAT solvers as model generators for a solver.

The first of these FO based proposals, presented in [5], concerns computing Herbrand models of clausal function-free FO theories, extended with cardinality aggregates and Horn rules. The other one introduces *model expansion* (MX)

for (extensions of) full FO as declarative problem solving paradigm [16]. The MX search problem for a logic $\mathcal{L}$, denoted MX($\mathcal{L}$), is the problem of finding the models of an $\mathcal{L}$-theory T that expand a given finite interpretation I of part of T's vocabulary. Currently, there are two existing MX solvers for FO: MXG [15] and IDP. The former is being developed at Simon Fraser University and can handle function free FO, extended with a restricted form of inductive definitions and aggregates. In this paper, we present the latter system.

The IDP system is an MX solver for a rich extension of full FO. More concretely, the input language of IDP extends FO with an order-sorted type system, inductive definitions, partial functions, arithmetic, existential quantifiers with numerical bounds and aggregates such as cardinality, minimum, maximum and sum. Though in principle none of these extensions increase the class of problems that can be represented in MX(FO) [16], they do often considerably simplify the modelling task and may increase the class of problems that can be solved in practice. For instance, *reachability* in the context of a finite domain can be expressed in FO, but not in a natural manner. On the other hand, it can easily be expressed using an inductive definition, and experiments (as, e.g., the ones in [11]) indicate that solvers able to natively handle such definitions are currently more efficient than pure SAT solvers on problems involving reachability.

The goal of this paper is to give a gentle introduction to the IDP system. In the next section, we describe the different language constructs of the input language and illustrate their use. In Section 3, we briefly describe the two components of the system: the grounder GIDL and the extended SAT solver MINISAT(ID). Section 4 demonstrates the IDP methodology and the combined use of several language constructs in one integrated example. Finally, in Section 5, we present some experiments comparing different model generation and model expansion systems of FO and ASP.

2 Specification Language

In this section, we present the specification language of the IDP system. We discuss the different language constructs that have been added to FO in IDP and illustrate their use.

We assume familiarity with many-sorted first-order logic (FO) (see, e.g., [7]). Here we will use the standard notations for logical connectives. For an overview of the ASCII notation of the connectives in IDP's *concrete* language, we refer to the manual of the system [21]. In this paper (as well as in IDP), variables start with a lowercase letter while predicate, constant and function symbols start with an uppercase letter. Sets and tuples of variables are denoted in bold by $\boldsymbol{x}$, $\boldsymbol{y}$, ..., and tuples of terms by $\boldsymbol{t}$. We use $\varphi[\boldsymbol{x}]$ to indicate that $\boldsymbol{x}$ are the free variables of the formula φ.

For an interpretation I (also called a structure), the value of a term t in I is denoted by t^I, the truth value of a formula φ in I by φ^I. By $I[\boldsymbol{x}/\boldsymbol{d}]$ we denote the interpretation that assigns the domain elements $\boldsymbol{d}$ to the variables $\boldsymbol{x}$

and corresponds to I on all other symbols. The restriction of a structure I over vocabulary Σ to a subvocabulary $\sigma \subseteq \Sigma$ is denoted by $I|_\sigma$.

The first two extensions of FO in the IDP language are the quantifiers with a cardinality constraint: $\exists_{=n}$ and $\exists_{<n}$. The meaning of formulas $\exists_{=n}x\ \varphi$ and $\exists_{<n}x\ \varphi$ is respectively that "there exist exactly n", respectively and " strictly less than n" objects x satisfying φ. They will be frequently used in the examples below.

2.1 Model Expansion

The IDP system solves the computational task of model expansion for FO and its extensions as presented here. We first formally define this task:

Definition 1. *Let $\mathcal{L}$ be a logic and T an $\mathcal{L}$-theory over a vocabulary Σ, and let $\sigma \subset \Sigma$. Then the* model expansion problem for logic $\mathcal{L}$ with input $\langle T, \sigma\rangle$, *denoted $MX_{\langle T,\sigma\rangle}$, is the problem of computing for an input σ-interpretation I, a Σ-model M of T such that $M|_\sigma = I$. M is called a T*-expansion *of I.*

The vocabulary σ is called the *input vocabulary*, $\Sigma \setminus \sigma$ the *expansion vocabulary* and I the *input structure*.

Mitchell and Ternovska [16] proved that model expansion for FO *captures NP*, in the following sense:

- for any T and σ, the decision problem of $\mathrm{MX}_{\langle T,\sigma\rangle}$, i.e., the problem of deciding whether an σ-structure has a T-expansion, is in NP;
- vice versa, for any NP decision problem X on the class of finite σ-structures, there is a theory T in a vocabulary that extends σ such that a finite σ-structure I belongs to X iff T has a model expanding I. In the latter case, we say that $\mathrm{MX}_{\langle T,\sigma\rangle}$ *expresses* X.

Observe that if $\sigma = \Sigma$, then $\mathrm{MX}_{\langle T,\sigma\rangle}$ reduces to model checking, while if $\sigma = \emptyset$, the problem is that of deciding the existence of a model of T with a given finite size. We illustrate MX for FO in the following examples.

Example 1 (Graph colouring). The *graph colouring* problem takes as input a graph and a set of colours, which we represent here by $\sigma =$ (sorts: $\{Vtx, Colour\}$, vocabulary: $\{Edge(Vtx, Vtx)\}$). A solution is any function from vertices to colours that maps neighbouring vertices to different colours. We represent this function by the function symbol $Colouring(Vtx) : Colour$. The IDP theory that models this problem consists of one formula:

$$\forall v_1 \forall v_2\ Edge(v_1, v_2) \supset Colouring(v_1) \neq Colouring(v_2).$$

Example 2 (SAT). We now encode the SAT problem for CNF formulas, thus demonstrating that every problem in NP can be reduced to an MX problem in polynomial time. The input vocabulary of our encoding is

$$\sigma = (\text{sorts: } \{Atom, Clause\},\ \text{vocabulary: } \{PosIn(Atom, Clause), NegIn(Atom, Clause)\}),$$

where the two sorts represent respectively the atoms and clauses of the given formula, and the two predicate symbols represent the positive, respectively negative occurrences of atoms in clauses. We search an assignment represented by the unary predicate $A(Atom)$, such that

$$\forall c\ \exists a\ (PosIn(a,c) \wedge A(a)) \vee (NegIn(a,c) \wedge \neg A(a)),$$

i.e., each clause contains a true literal.

Interestingly, the grounding produced by the grounder of the IDP system (see Section 3.1) is exactly the CNF formula represented by the input interpretation.

2.2 Inductive Definitions

A first extension of FO is the logic FO(ID) [3, 4], which extends FO with inductive definitions. Inductive definitions have many applications in real problems, e.g., in problems involving reachability. In the context of finite structures, inductive definitions can in principle be encoded in FO (e.g., by encoding the fixpoint construction) but the process is tedious and leads to large theories.

A *definition* Δ is a finite set of rules of the form

$$\forall \boldsymbol{x}\ (P(\boldsymbol{t}) \leftarrow \varphi[\boldsymbol{y}]),$$

where P is a predicate symbol, φ an FO formula, $\boldsymbol{y} \subseteq \boldsymbol{x}$ and $\boldsymbol{t}$ a tuple of terms such that its free variables are among $\boldsymbol{x}$. $P(\boldsymbol{t})$ is called the *head* of the rule, $\varphi[\boldsymbol{y}]$ the *body*. The connective $\leftarrow$ is called *definitional implication* and is to be distinguished from material implication $\supset$. The aim of a definition is to define its set of *defined predicates* in terms of other symbols, called the *open symbols* of Δ. The defined predicates are those that appear in the head of a rule. Formally, a structure satisfies a definition Δ if its interpretation of the defined symbols is given by the *well-founded model* [20] of Δ computed in terms of the interpretation of the open symbols. As argued in, e.g., [4], the well-founded semantics is used because it correctly formalizes the semantics of all common types of inductive definitions in mathematics.

Example 3 (Transitive closure). Definition Δ_1 defines relation T to be the transitive closure of relation R.

$$\Delta_1 = \left\{ \begin{array}{l} T(x,y) \leftarrow R(x,y), \\ T(x,y) \leftarrow \exists z\ (T(x,z) \wedge T(z,y)) \end{array} \right\}$$

An FO(ID) theory $\mathcal{T}$ is a finite set of FO sentences and definitions. Thus, an FO(ID) theory has the appearance of an FO theory augmented with a collection of *logic programs*. This entails that FO(ID)'s definitions cannot only be used to represent mathematical concepts, but also for common sense knowledge such as (local forms of) CWA, inheritance, exceptions, defaults, causality, etc..

Example 4 (Transitive opening). A *transitive opening* of a binary relation T is a minimal relation R with transitive closure T. We express the problem to find

a transitive opening as an MX(FO(ID)) problem with input vocabulary $\sigma =$ (sorts: $\{Vtx\}$, vocabulary: $\{T(Vtx, Vtx)\}$). The requirement that the expansion predicate R's transitive closure is T is expressed by Δ_1 in Example 3. In logic, minimality of a relation is normally expressed by a second order axiom which cannot be expressed in IDP. However, in this case we can express the minimality of R as follows. We introduce a predicate $TE(Vtx, Vtx, Vtx, Vtx)$ such that, for each u and v, the binary relation $TE(\cdot, \cdot, u, v)$ denotes the transitive closure of $R \setminus \{(u, v)\}$.

$$\Delta_2 = \left\{ \begin{array}{l} TE(x, y, u, v) \leftarrow R(x, y) \land \neg(x = u \land y = v) \\ TE(x, y, u, v) \leftarrow \forall z \ (TE(x, z, u, v) \land TE(z, y, u, v)) \end{array} \right\}$$

The minimality of R is then expressed by the formula

$$\Psi = \forall x, y \ (R(x, y) \supset \neg TE(x, y, x, y)).$$

It expresses that each $(x, y) \in R$ necessarily belongs to R, i.e., (x, y) does not belong to the transitive closure of $R \setminus \{(x, y)\}$ nor to any of its subrelations. It is easy to prove that the MX(FO(ID)) problem $\mathrm{MX}_{\langle\{\Delta_1, \Delta_2, \Psi\}, \sigma\rangle}$ correctly models the transitive opening problem, i.e. that the expansion models M of an input interpretation I have a one to one correspondence to transitive openings of T^I.

2.3 Partial Functions

In standard FO, function symbols represent total functions. Many real applications contain functions that are partial on the whole domain, yet total on a well-known subset of the universe. Examples are *spouse* for married people, the arithmetic functions $\div$ and *mod*, etc.

In general, arbitrary use of partial function symbols creates an ambiguity problem. E.g., consider the formula $P(F(\boldsymbol{t})))$, where F is a partial function symbol. This formula can be interpreted in two different ways, as illustrated by the following non-ambiguous rewritings of it:

$$\exists y \ (F(\boldsymbol{t}) = y \land P(y)) \tag{1}$$

$$\forall y \ (F(\boldsymbol{t}) = y \supset P(y)) \tag{2}$$

Here, the atoms $F(\boldsymbol{t}) = y$ should be interpreted as $G_F(\boldsymbol{t}, y)$, where G_F denotes the graph of F. When F is total, both rewritings are equivalent, but this is not the case when F is partial. Indeed, for an interpretation I such that $\boldsymbol{t}^I$ is not in the domain of F^I, $I \not\models (1)$, but $I \models (2)$.

In IDP we interpret positive occurrences of atoms $P(F(\boldsymbol{t})))$ by (2) and negative occurrences by (1). This leads to a cautious interpretation of sentences with such statements, i.e., an interpretation that minimizes the truth value of a sentence. Our observation is that this cautious interpretation is usually the intended one. In cases where it is not, one can easily achieve the correct behaviour by explicitly rewriting the occurrence in the desired non-ambiguous form (1) or (2).

Example 5. Assuming that *Spouse* is a partial function, the formal meaning of the formula $\forall x\ (Male(x) \supset Female(Spouse(x))$ is

$$\forall x\ (Male(x) \supset \forall y(Spouse(x) = y \supset Female(y)).$$

This means that the spouse of a male is female, provided that this spouse exists. If we had opted for (1), the meaning would be that any male has a female spouse. The sentence with a negative occurrence of a partial function, $\forall x\ (Male(Spouse(x)) \supset Female(x))$, means that a person is female if she has a male spouse. If we had opted for (2), the meaning would have been: if all spouses of a person are male, then (s)he is female. Since the condition is satisfied for a person without spouse, such person is claimed to be female.

2.4 Subsorts

The input language for IDP extends many-sorted FO to order-sorted FO. More precisely, it allows sorts to be a direct subsort of at most one other sort. The corresponding hierarchy of sorts must be a collection of trees. The roots of the trees are *base* sorts. A function that has at least one argument position of a non-base sort is treated as a partial function.

2.5 Arithmetic

In IDP, every vocabulary contains a sort *int* and the arithmetic functions $+$, $-$, $\cdot$, $\div$, $abs(\cdot)$ and *mod*. In every structure over such a vocabulary, *int* is interpreted by the integers $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$, $+$ by addition on $\mathbb{Z}$, $-$ by subtraction, $\cdot$ by multiplication, $\div$ by integer division, *abs* by the absolute value and *mod* by the remainder. Note that $\div$ and *mod* are partial functions on $\mathbb{Z}$ with domain $\mathbb{Z} \setminus \{0\}$. Terms of the form $t_1 + t_2$, $t_1 \cdot t_2$, etc, are of sort *int*.

To ensure finite grounding, the use of *int* is restricted in IDP. In the input and expansion vocabulary declaration, a sort can be declared to be a subsort of *int* and a variable may have sort *int*. On the other hand, predicate or function declarations with sort $(\ldots, int, \ldots)$ are not allowed. Each integer variable x in a formula should belong to some finite subtype of *int* or, it should occur in a subformula $\forall x\ (\varphi \supset \ldots)$, respectively $\exists x\ (\varphi \wedge \ldots)$ where φ is a formula that puts a finite bound on the value of x. I.e., there exists a finite interval $[n, m]$ such that $M[x/d] \not\models \varphi$ for any model M of the theory and domain element $d \notin [n, m]$. We call φ a *bound* for x. One can specify a bound φ for variable x also in other, equivalent forms, as in $\forall x\ (\ldots \vee \neg\varphi \vee \ldots)$ or $\exists x\ (\ldots \wedge \varphi \wedge \ldots)$. The bounds allowed in IDP have a very simple form. E.g., an atom $P(\ldots, x, \ldots)$ is a bound and also formulas of the form $t_1 \leq x \leq t_2$, at least if t_1 can be bounded from below and t_2 from above; i.e., the IDP grounder should be able to find integers $n_1, n_2 \in \mathbb{Z}$ such that $n_1 \leq t_1^M$ and $t_2^M \leq n_2$ in each expansion M.

Example 6 (N-queens). The N-queens problem consists in positioning N chess queens on a $N \times N$ chessboard in such a way that no two queens attack each other, i.e. are on the same row, column, or diagonal.

This encoding uses input vocabulary $\sigma =$ (sorts: $\{int\ Row, int\ Column\}$, vocabulary: $\emptyset$). For the problem instance with parameter N, the domain of both sorts will both be $\{1, \ldots, N\}$. The solution is represented by the function symbol $Q(Row) : Column$, satisfying:

$$\forall c\ \exists_{=1} r\ Q(r) = c, \tag{3}$$

$$\begin{aligned} \forall r_1, r_2, c_1, c_2\ \big(Q(r_1) = c_1 \wedge Q(r_2) = c_2 \wedge r_2 > r_1\big) \supset \\ \big(r_2 - r_1 \neq abs(c_2 - c_1)\big). \end{aligned} \tag{4}$$

Note the use of the quantifier $\exists_{=1}$ in the first axiom.

2.6 Aggregates

Aggregates are functions that have a set as argument. IDP supports the aggregates: cardinality, sum, product, minimum and maximum. Concretely, the following terms have sort *int* and free variables $\boldsymbol{z}$: $card\{\boldsymbol{y} \mid \varphi[\boldsymbol{y}, \boldsymbol{z}]\}$, $sum\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\}$, $prod\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\}$, $min\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\}$ and $max\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\}$. The variables $\boldsymbol{z}$ are free in the aggregate term, while x and $\boldsymbol{y}$ are local to the term. The sort of x must be a subsort of *int*. Given an interpretation I, these terms are interpreted by

- $(card\{\boldsymbol{y} \mid \varphi[\boldsymbol{y}, \boldsymbol{z}]\})^{I[\boldsymbol{z}/\boldsymbol{d}_z]}$ is the number of tuples of domain elements $\boldsymbol{d}$ such that $I[\boldsymbol{y}/\boldsymbol{d}_y, \boldsymbol{z}/\boldsymbol{d}_z] \models \varphi$;
- $(sum\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\})^{I[\boldsymbol{z}/\boldsymbol{d}_z]} = \sum_{I[x/d_x, \boldsymbol{y}/\boldsymbol{d}_y, \boldsymbol{z}/\boldsymbol{d}_z] \models \varphi} d_x$;
- $(prod\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\})^{I[\boldsymbol{z}/\boldsymbol{d}_z]} = \prod_{I[x/d_x, \boldsymbol{y}/\boldsymbol{d}_y, \boldsymbol{z}/\boldsymbol{d}_z] \models \varphi} d_x$;
- $(min\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\})^{I[\boldsymbol{z}/\boldsymbol{d}_z]} = min\{d_x \mid I[x/d_x, \boldsymbol{y}/\boldsymbol{d}_y, \boldsymbol{z}/\boldsymbol{d}_z] \models \varphi\}$;
- $(max\{x, \boldsymbol{y} \mid \varphi[x, \boldsymbol{y}, \boldsymbol{z}]\})^{I[\boldsymbol{z}/\boldsymbol{d}_z]} = max\{d_x \mid I[x/d_x, \boldsymbol{y}/\boldsymbol{d}_y, \boldsymbol{z}/\boldsymbol{d}_z] \models \varphi\}$;

Aggregates can be used everywhere in sentences or rule bodies where a term with a subsort of *int* can occur. The semantics for definitions containing recursion involving aggregates is the one presented in [18].

Example 7 (Frequent itemset mining). We illustrate the well-known machine learning problem of *frequent itemset mining* in the context of a warehouse domain. The warehouse has a number of items on offer; customers purchase sets of such items. The problem consists in determining sets of items that are often bought together. The search for frequent item sets can be represented as a model expansion problem with input vocabulary $\sigma =$ (sorts: $\{Item, Purchase\}$, vocabulary: $\{Bought(Purchase, Item)\}$), where *Bought* represents which items were bought at which purchases. We are interested in itemsets with a frequency higher than some value represented by an integer constant *Frequency*. The expansion predicate $ItemSet(Item)$ represents a frequent itemset if it satisfies the formula:

$$card\{t \mid \forall i\ ItemSet(i) \supset Bought(t, i)\} \geq Frequency.$$

Thus, model expansion applied on this axiom computes frequent itemsets.

Example 8 (Company control). A well-known example of recursion over aggregates in ASP is that in *company control* problem: a company A controls a company B if the sum of shares of B that are in A's control exceeds 50. In IDP, we model this problem using input vocabulary $\sigma =$ (sorts: $\{Share, Company\}$, vocabulary: $\{Owns(Company, Company) : Share\}$) and the expansion predicate $Controls(Company, Company)$. Solutions must satisfy the recursive definition:

$$\left\{ \begin{array}{l} Controls(x, y) \leftarrow sum\{s, z \mid (x = z \vee Controls(x, z)) \\ \qquad\qquad\qquad\qquad\qquad \wedge Owns(z, y) = s \wedge Share(s)\} > 50 \end{array} \right\}.$$

3 System Architecture

The IDP system consists of two systems: a grounder and a propositional model generator.

3.1 Grounder

The IDP grounder is GIDL [22]. Its task is to transform a given MX problem with a given input structure to a propositional theory, the models of which correspond to the expansion models.

GIDL's algorithms have been described in [23, 22]. The grounding has two main phases : the first phase applies a form of approximate reasoning, based on the structure of the theory and on the input vocabulary, to compute *bounds* on the values of variables of all subformulas of the theory. This phase is independent of the input interpretation. These bounds are used in a second phase to produce a smaller grounding.

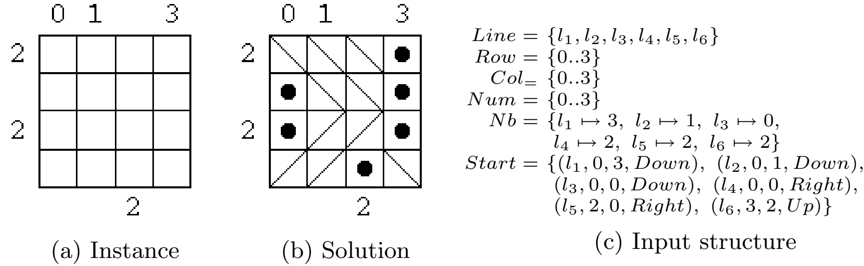
Despite the richness of the input language described in Section 2, GIDL ranks among the top performing grounding systems, both regarding time to produce the grounding, and size of the grounding. Cfr. [22] for a thorough experimental evaluation of the grounder.

The output language of GIDL is extended CNF (ECNF), an extension of CNF with propositional definitions and propositional aggregate expressions.

3.2 Solver

There are two propositional solvers for the IDP system: MIDL [12, 13] and the more recent MINISAT(ID) [14]. The former supports CNF with propositional definitions, the latter supports full ECNF. We focus on the latter.

The system is built as an extension of the SAT solver MINISAT [6]. Its algorithms are geared towards a seamless integration with the unit propagation and clause learning algorithms present in a DPLL-based SAT solver such as MINISAT. On pure CNF instances, MINISAT(ID) has the same performance as MINISAT (which has consistently ranked among the top SAT solvers in the past few SAT competitions); on ECNF instances, MINISAT(ID) ranks among the best systems with comparable capabilities, cfr. [14] for a thorough experimental evaluation.



(a) Instance (b) Solution (c) Input structure

Fig. 1: Example of the mirror puzzle problem.

4 Example Application

In this and other papers [11, 21], a number of applications of IDP have been demonstrated, including standard constraint problems such as k-colourability, N-queens, Hamiltonian circuits, planning problems such as the block's world and sokoban, as well as diagnosis and scheduling problems. One of the main assets of the IDP system is its rich input language: all of these problems can be encoded in a compact, natural way, without introducing syntactic workarounds. To illustrate this, we here discuss a moderately complex problem that involves a combination of different IDP primitives: function symbols, inductive definitions and aggregate expressions. A *mirror puzzle problem*[1] consists of a grid with some integer values along its sides, such as in Figure 1a. The problem is to fill the grid with persons and diagonally positioned mirrors, in such a way that when "looking" into the grid from a location with value n, one can "see" exactly n persons. In Figure 1b, a solution is given. In this solution, looking right from position $(2, 0)$ (with $n = 2$), we see subsequently positions $(2, 0), (2, 1)$, —upward deflection— $(1, 1)$, and —left deflection— $(1, 0)$. This path contains indeed 2 persons.

The sorts we distinguish in this domain are *Row* and *Col*, *Object* (either person, upward mirror, or downward mirror), *Direction* (up, down, left, right), numbers (of persons) and *Line*. A line represents a visual path along one the given values; its start position and direction are given. Problem instances are given through the function $Nb(Line) : Num$, specifying the value associated to a given line, and $Start(Line, Row, Col, Dir)$, specifying where and in what direction a given line starts. The instance in Figure 1a can thus be specified by the input structure given in Figure 1c.

Our theory uses auxiliary predicates. $Turn(d, o, d')$ means that an object o deflects direction d into d' (e.g., $Turn(Up, Upmirror, Right)$). $NextR(r, r', d)$ means that the next row of row r in direction d is r' (e.g., $NextR(0, 0, left)$, $NextR(0, 1, Down), \ldots$). $NextC$ is the analogous relation for columns. Each of these predicates has a straightforward definition. Another auxiliary predicate is

[1] `http://www.stetson.edu/~efriedma/mirror`

$Pass(Line, Row, Col, Direction)$. $Pass(l, r, c, d)$ holds if the visual path starting in l, reaches position (r, c) from direction d. This predicate is defined inductively as follows:

$$\left\{ \begin{array}{l} Pass(l,r,c,d) \leftarrow Start(l,r,c,d), \\ Pass(l,r,c,d) \leftarrow Pass(l,r',c',d') \\ \qquad\qquad\qquad \wedge\, Turn(d', Contains(r',c'), d) \\ \qquad\qquad\qquad \wedge\, NextR(r,r',d) \\ \qquad\qquad\qquad \wedge\, NextC(c,c',d) \end{array} \right\} \tag{5}$$

Note that if a visual path passes more than once in the same position, it will reach this position from different directions. Otherwise, the path would be cyclic; such paths cannot exist in this problem.

With these definitions in place, the problem consists in finding a function $Contains(Row, Col) : Object$ satisfying (6):

$$\forall l\ card\{r, c, d \mid Pass(l, r, c, d) \wedge Contains(r, c) = Person\} = Nb(l) \tag{6}$$

5 Experimental validation

In this section we compare the performance of IDP system to that of systems with similar expressivity. The main purpose of this comparison is to show that IDP has a reasonable performance compared to state of the art systems—its rich and convenient specification language is its main asset. Our results show that IDP is a worthy competitor—and has one of the most constant and robust performances—despite this rich language.

Comparing performances is a difficult task: many systems have a different input language. In our experiment, we have tried as much as possible to use the modelling used by the authors of the different systems. We obtained these from the "Modelling-Grounding-Solving" section of the ASP competition [9], from the Asparagus website and from the systems' authors' websites. All of them can be found on the website `http://www.cs.kuleuven.be/~dtai/krr/software.html`. We have chosen a range of different problems, featuring different kinds of expressions: the k-Colouring encoding employs only FO statements in IDP; the Hamiltonian cycle encoding employs also an inductive definition (the MXG solver cannot handle inductive definitions, and uses an FO encoding based on a total order of vertices); the Magic series encoding uses a cardinality expression; the Social golfer encoding uses the cardinality aggregate; and the Blocked N-queens problem can be modelled easily in pure FO, but has also beautiful solutions using the "there exists a unique"-quantifier ($\exists_{=1}$) and cardinality statements.

Results are given in Table 1. They are run on a C2D 3GHz processor with 2GB memory, and a timeout of 10 minutes (for the grounding+solving time). The solvers used are: "Clasp": Lparse 1.0.17 [19] + Clasp 1.0.5 [8], "DLV": DLV Oct 11 2007 [2], "MXG": mxg 0.16 + mxc (sr-08) [15], "aspps": psgrnd 7-Jul-2005 + aspps 2003.06.04 [5], "IDP": GIDL 1.5.0 + MINISAT(ID) 1.3b. It appears that the IDP system has the best performance on at least three aggregated measures:

number of unsolved instances, number of instances solved in less then 10 seconds, and total time (where timeout counts for exactly $600s$).

6 Conclusion

We presented the IDP system for solving model expansion problems in the context of a very rich extension of classical logic. We described its input language and illustrated with several examples how the language allows a natural and very compact encoding of many search problems. Despite its rich input language, our experiments showed that IDP performs well compared to other MX and ASP solvers.

References

1. Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors. *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*. Springer, 2007.
2. Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, and Gerald Pfeifer. System description: DLV. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 424–428. Springer, 2001.
3. Marc Denecker. Extending classical logic with inductive definitions. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 703–717. Springer, 2000.
4. Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions. *Transactions On Computational Logic (TOCL)*, 9(2), 2008.
5. Deborah East and Miroslaw Truszczyński. Predicate-calculus-based logics for modeling and solving search problems. *ACM Trans. Comput. Log.*, 7(1):38–83, 2006.
6. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
7. Herbert B. Enderton. *A Mathematical Introduction To Logic*. Academic Press, 1972.
8. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp*: A conflict-driven answer set solver. In Baral et al. [1], pages 260–265.
9. Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Miroslaw Truszczynski. The first answer set programming system competition. In Baral et al. [1], pages 3–17.
10. Victor W. Marek and Mirek Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, V. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25 Years Perspective*, pages pp. 375–398. Springer-Verlag, 1999.
11. Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

Problem	Clasp	DLV	MXG	aspps	IDP
3-Colouring					
simplex30	0.34	0.18	0.23	0.18	0.19
simplex60	0.55	1.02	0.55	0.25	0.32
simplex120	3.73	5.54	1.75	0.87	1.06
simplex240	40.64	78.47	5.36	2.65	2.50
simplex360	segf	397.78	11.92	6.18	5.93
Hamiltonian circuit					
graph1	0.16	1.38	54.95	0.31	0.27
graph2	0.15	1.22	53.06	0.31	0.26
graph3	0.15	1.22	49.71	0.29	0.27
graph4	0.18	1.31	50.31	0.32	0.25
graph5	0.34	2.61	424.95	0.49	0.34
graph6	0.32	2.91	203.48	0.49	0.35
graph7	0.35	6.84	>600	0.85	0.45
graph8	0.56	7.36	>600	0.83	0.48
Magic series					
size10	0.16	385.10	n/a	0.08	0.14
size20	2.02	>600	n/a	0.22	0.42
size30	4.19	>600	n/a	0.93	0.57
size40	121.15	>600	n/a	3.51	4.05
size50	112.19	>600	n/a	10.84	0.77
size60	>600	>600	n/a	27.53	3.47
size70	>600	>600	n/a	64.85	>600
size80	>600	>600	n/a	131.55	65.12
Social golfer					
w2g4s5	>600	187.16	>600	>600	>600
w3g6s4	0.25	6.64	1.31	0.22	0.40
w3g6s5	0.38	29.99	2.43	0.37	0.32
w4g3s4	7.68	16.35	1.72	459.78	0.37
w4g5s4	0.21	6.51	1.39	0.23	0.24
w4g5s5	0.33	>600	2.74	408.82	0.34
w6g6s3	0.31	0.47	1.61	0.26	0.29
w8g3s3	0.13	>600	10.80	>600	0.76
Blocked 48-queens					
block1	22.90	>600	17.99	0.36	4.72
block2	10.50	533.58	14.98	2.38	9.20
block3	10.66	163.58	16.76	2.23	8.98
block4	3.43	20.58	13.27	2.04	5.60
block5	1.87	381.41	15.15	0.18	6.79
block6	5.12	63.25	10.63	0.59	1.20
block7	3.24	87.11	8.60	0.82	1.91
block8	4.63	581.04	8.22	1.17	3.41
No. unsolved instances	5	10	11	2	2
No. instances $>10s$	11	23	25	8	3
Total time (s)	3359	8971	7584	2333	1332

Table 1: Modelling-Grounding-Solving: timings (s) of different systems

12. Maarten Mariën, Johan Wittocx, and Marc Denecker. Integrating inductive definitions in SAT. In Nachum Derschowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2007.
13. Maarten Mariën, Johan Wittocx, and Marc Denecker. MidL: a SAT(ID) solver. In *4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, pages 303–308, 2007.
14. Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2008.
15. David Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR2006-24, Simon Fraser University, 2006.
16. David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005.
17. Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
18. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *TPLP*, 7(3):301–353, 2007.
19. Tommi Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Helsinki University of Technology, Finland, 1998.
20. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
21. Johan Wittocx and Maarten Mariën. The IDP system. `http://www.cs.kuleuven.be/~dtai/krr/software/idpmanual.pdf`, 2008.
22. Johan Wittocx, Maarten Mariën, and Marc Denecker. GidL: A grounder for FO^+. In Michael Thielscher and Maurice Pagnucco, editors, *NMR'08*, 2008. To appear.
23. Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding with bounds. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 572–577. AAAI Press, 2008.