# *Detection and Exploitation of Functional Dependencies for Model Generation*

BROES DE CAT AND MAURICE BRUYNOOGHE

*Department of Computer Science, KU Leuven, Belgium*
(*e-mail:* {broes.decat,maurice.bruynooghe}@cs.kuleuven.be)

*submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003*

**Abstract**

Recent work in Answer Set Programming has integrated ideas from Constraint Programming. This has led to a new field called ASP Modulo CSP (CASP), in which the ASP language is enriched with *constraint atoms* representing constraint satisfaction problems. These constraints have a more compact grounding and are handled by a new generation of search algorithms. However, the burden is on the modeler to exploit these new constructs in his declarative problem specifications. Here, we explore how to remove this burden by automatically generating constraint atoms. We do so in the context of FO$(\cdot)^{\text{IDP}}$, a knowledge representation language that extends first-order logic with, among others, inductive definitions, arithmetic and aggregates. We uncover functional dependencies in declarative problem specifications with a theorem prover and exploit them with a transformation that introduces functions. Experimental evaluation shows that we obtain more compact groundings and better search performance.

## 1 Introduction

Model generation is a widely used problem solving paradigm. A problem is specified as a theory in a declarative logic in such a way that models of the theory represent solutions to the problem. A closely related paradigm is bounded model expansion. Here, a partial input structure over a finite and known domain is extended into a complete structure that is a model of a given theory. These paradigms are studied and applied in the fields of Constraint Programming (CP) (Apt 2003), Answer Set Programming (Niemelä 2006) and Knowledge Representation (Baral 2003).

A state-of-the-art approach is to reduce the input theory, formulated in an expressive logic, in a model-equivalence preserving way to a theory in a fragment of the language supported by some search algorithm. Afterwards, this algorithm searches for models of the theory. For example, model generation/expansion for the language FO$(\cdot)$ (Denecker and Ternovska 2008) is performed by reducing theories to the ground language PC$(\cdot)$, for which efficient search algorithms are available. The term *grounding* refers to both the reduction process and to its outcome; the 2-step approach is called *ground-and-solve*.

A first generation of model expansion systems used search algorithms for (pseudo)-propositional languages, such as Clausal Normal Form (SAT solvers) and ground ASP (ASP solvers). An important bottleneck of such systems is the blowup caused by grounding the input theory, as the size of the theory increases rapidly with the size of the domain and the nesting depth of quantified variables. To overcome this limitation, techniques from Constraint Programming have been incorporated, giving rise to the field of

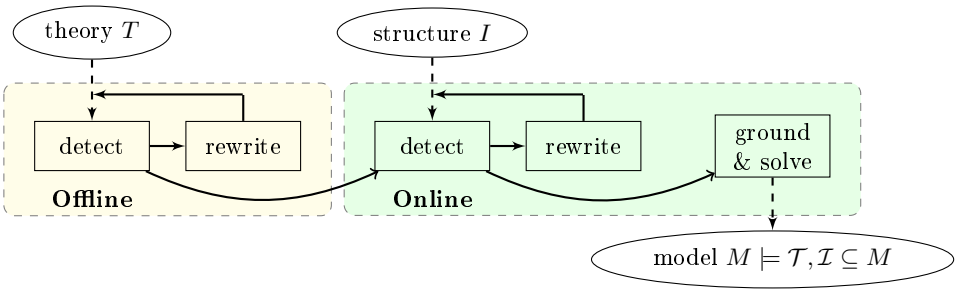


Fig. 1. Workflow. In a first "offline" phase, the theory is used to detect functional dependencies and functions are introduced until no more can be found (or a time-out is reached). This is repeated in the "online" phase, now combined with the input structure. The transformed theory is then passed to the ground-and-solve algorithm.

ASP modulo CSP (CASP) (Ostrowski and Schaub 2012). Search algorithms have been developed that allow *constraint atoms* in the input. These atoms are ground instead of propositional, have (non-Herbrand) function terms as arguments, and stand for the constraints of a CSP problem (Lierler 2012; Gebser et al. 2009). This gives rise to more compact groundings that often also yield better propagation. Among those next generation systems are the solvers Clingcon(Ostrowski and Schaub 2012), Ezcsp(Balduccini 2011) and Constraint(ID) (De Cat et al. 2013).

As the original ASP language does not support (non-Herbrand) functions, ASP modulo CSP systems extend the language with special-purpose constraints with a more compact grounding (see (Cabalar 2013; Bartholomew and Lee 2012; Lifschitz 2012; Balduccini 2012; Cabalar 2011; Lin and Wang 2008) for approaches to enhance ASP with functions); to a certain degree, this is also the case for the language Zinc (Marriott et al. 2008). However, the user has to use these constructs in his declarative specifications in order to obtain the benefits of a more compact grounding and a better performance.

This paper explores to what extent this burden can be removed from the modeler. We uncover functional dependencies in declarative problem specifications with a theorem prover and exploit them with a transformation that introduces functions and, in the process, eliminates quantified variables. This results in a more compact grounding and more efficient search. We do this in the context of FO$(\cdot)^{\text{IDP}}$, the language supported by the knowledge-base system IDP (Bogaerts et al. 2012), which extends FO with aggregate functions, inductive definitions, partial functions, types and arithmetic. The same ideas could be applied in the context of ASP languages. The analysis can be performed on theories both with and without input structure, giving rise to the workflow of Figure 1.

FO$(\cdot)^{\text{IDP}}$ supports both functions and predicates and users are free to use predicates when some of its arguments depend functionally on each other (out of preference, ignorance or because the theory is a translation of an ASP theory). Also, it may happen that a functional dependency only holds for a particular problem instance (e.g., a graph where each vertex has exactly one outgoing edge).

*Example 1*
Consider a scheduling application involving some events (*events*) to be planned, each exactly once, over a large period of time (*time*). A total order $<$ on events is given. One

possible constraint is that the planning of events has to follow their order. In FO($\cdot$), this can be represented as the theory consisting of the following sentences, with a grounding size of $\|event\|^2 \times \|time\|^2$ (typically measured in number of ground atoms):

$$\forall e : \exists! t : planned(e, t),$$
$$\forall e_1\ e_2\ t_1\ t_2 : e_1 < e_2 \wedge planned(e_1, t_1) \wedge planned(e_2, t_2) \Rightarrow t_1 < t_2.$$

However, if we can prove that the second argument of *planned* depends functionally on the first, then *planned* can be replaced by a function symbol, say $f_{planned} : event \mapsto time$. By equivalence preserving transformations, a theory with a grounding size of only $\|event\|^2$ can then be obtained, namely $\forall e_1\ e_2 : e_1 < e_2 \Rightarrow f_{planned}(e_1) < f_{planned}(e_2)$.

The grounding contains constraint atoms $f_{planned}(e_1) < f_{planned}(e_2)$. In CASP clingcon syntax, the constraint atom is written as $f_{planned}(E1)\$ < f_{planned}(E2)$.

The paper is organized as follows. In Section 2, FO($\cdot$) and necessary concepts are introduced. Next, we present the detection algorithm in Section 3 and the theory transformations in Section 4. In Section 5, experimental results are presented; we finish with related work and conclusions in Section 6.

## 2 Preliminaries

This paper makes use of FO($\cdot$)$^{\text{IDP}}$, a many-sorted logic that extends First-Order Logic (FO) with aggregate functions, arithmetic, inductive definitions and partial functions. An FO($\cdot$)$^{\text{IDP}}$ vocabulary consists of types and typed predicate and function symbols. The signatures of $n$-ary predicates $P$ and functions $f$ are denoted, respectively, as $P(T_1, \ldots, T_n)$ and $f(T_1, \ldots, T_n) : T_{n+1}$. For each type $T$, a predicate symbol $T(T)$ exists, interpreted *true* for all domain elements in type $T$. An atom/term is badly-typed if at least one of its arguments is outside the interpretation of the declared type of that argument position. Badly-typed *atoms* or *atoms* containing badly-typed terms are always interpreted *false*.[1]

We assume familiarity with first-order logic. We often follow some conventions for symbols: we use $a$ for an atom, $l$ for a literal (an atom or its negation), $x$ and $y$ for variables, $\overline{x}$ for a tuple of variables, $D$ for a set of domain elements, $t$ for a term, $\overline{t}$ for a tuple of terms, $\overline{t} :: t$ for the concatenation of a tuple and a term, $\varphi$ for a formula, $c$ for a constant, $f$ for a function and $P$ and $Q$ for predicates. With $\overline{t} = \langle t_1, \ldots, t_n \rangle$ a tuple and $S = [s_1, \ldots, s_m]$ a subsequence of $[1, n]$ (an *index set*), $\overline{t}|_S$ denotes the tuple $\langle t_{s_1}, \ldots, t_{s_m} \rangle$ while $S^c$ denotes the complement of $S$ with respect to $[1, n]$, i.e., the elements of $S$ are removed. Given two tuples $\overline{t}$ and $\overline{t}'$, both of length $n$, $\overline{t} = \overline{t}'$ denotes the conjunction $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$. With $t[t']$ we denote a term $t$ with occurrences of $t'$ and with $t[t'/t'']$ the term where these occurrences are replaced by $t''$.

A term is an FO term or an *aggregate term*. Aggregate terms consist of an aggregate function (cardinality, sum, product, minimum or maximum) and a set comprehension of the form $\{\overline{x} : \varphi : t\}$, with $\varphi$ a formula and $t$ a term. It is interpreted as the aggregate function applied to the multiset of instances of $t$, one for each distinct instance of $\overline{x}$ for which $\varphi$ is true. The cardinality function, $\#\{\overline{x} : \varphi : t\}$, counts the number of distinct instances of $\overline{x}$ that makes $\varphi$ true and can be abbreviated as $\#\{\overline{x} : \varphi\}$ as $t$ is irrelevant.

[1] The negation of a badly-typed atom is true.

The aggregate functions sum, product, min and max map the multiset of instances of $t$ to, respectively, the sum, product, minimum, and maximum of the multiset of terms. When the multiset is empty, the result is 0, 1, $+\infty$ and $-\infty$, respectively. For example, $sum(\{x\ y : P(x,y) : f(x,y)\})$ is interpreted as the sum of the values of $f(x,y)$ for all instantiations of $x$ and $y$ for which $P(x,y)$ is true.

Functions are *total* unless declared as *partial*. An interpretation $\mathcal{I}$ is two-valued for a total (partial) function if, for each tuple of domain elements in its domain, it maps to exactly one (at most one) domain element in its codomain.[2] A formula $\exists x : f(\overline{y}) = x$ is true iff $f$ has an image in its codomain (given $\mathcal{I}$); it is abbreviated as $HasImage(f(\overline{y}))$.[3] An atom $a$ containing a term $f(\overline{y})$ over a partial function symbol $f$ is interpreted as $HasImage(f(\overline{y})) \wedge a$; it is false when $f(\overline{y})$ has no image.

A function can be *defined* by a set of rules of the form $\forall \overline{x} : f(\overline{t}) = t' \leftarrow \varphi$, a predicate by rules $\forall \overline{x} : P(\overline{t}) \leftarrow \varphi$. These rule sets are called definitions and are interpreted according to the well-founded semantics (Van Gelder 1993). [4] We assume rule sets for which the well-founded semantics are two-valued. The completion of the rules defining a symbol $s$ in definition $\Delta$ is denoted by $comp_{\Delta,s}$.

A theory consists of sentences and definitions. The notation $voc(\mathcal{T})$ is used to denote the vocabulary of a theory $\mathcal{T}$. Besides a vocabulary and a theory, an FO$(\cdot)^{\text{IDP}}$ specification also contains an input structure that interprets all types of the theory and (partially) interprets the functions and predicates. A typical computational task (model expansion) is to find an extension of the input structure that satisfies the sentences and is the two-valued well-founded model of the defined predicates and functions.

Given a vocabulary $\Sigma$, two theories $T$ and $T'$ are $\Sigma$-*equivalent* if each model of $T$ restricted to $\Sigma$ can be extended to a model of $T'$ and vice-versa. Two theories $T$ and $T'$ are *strongly* $\Sigma$-*equivalent* if the above extensions are also unique.

We assume theories are in *flat negation normal form*(FNNF): negations only occur directly in front of atoms, $\Rightarrow$ and $\Leftrightarrow$ are eliminated and a logical operator never occurs as a direct subformula of the same operator (e.g., $a_1 \vee a_2 \vee a_3$ is in FNNF, but $a_1 \vee (a_2 \vee a_3)$ is not). This assumption is without loss of generality as any theory can be transformed into an equivalent FNNF theory in time polynomial in the size of the theory.

## 3 Detecting Functional dependencies

On the next page, the FO$(\cdot)^{\text{IDP}}$ theory for the well-known *packing*-problem for squares is shown. It makes use of the predicates $size(id, nb)$ and $area(nb, nb)$ (interpreted in the input structure, together with the types $id$ and $nb$) and of the predicate $pos(id, nb, nb)$ that specifies the $x$- and $y$-coordinate of the bottom-left corner of the square $id$. The sentences express respectively the constraints (as FO sentences) that (1) each square is placed at exactly one (the $\exists!$ quantifier) position, (2) no squares overlap and (3) each square fits completely inside the specified area. The predicate $noOverlap/3$, used in (2), is defined between "{" and "}" and makes use of two auxiliary defined predicates.

[2] This deviates from the standard definition in order to take the type definition into account.
[3] A search algorithm with support for functions need not ground the formula for all values of $x$.
[4] An interpretation $\mathcal{I}$ satisfies the definition of an $n$-ary function $f$ if it satisfies its graph (i.e., the $n+1$-ary relation corresponding to the function) and $f$ is functional in $\mathcal{I}$ (partial if $f$ is partial).

$$\forall id : \exists!(x\ y) : pos(id, x, y) \quad (1) \qquad \forall id_1\ id_2 : id_1 \neq id_2 \Rightarrow noOverlap(id_1, id_2) \quad (2)$$

$$\forall id\ x\ y\ a\ b : (pos(id, x, y) \wedge area(a, b) \wedge size(id, s)) \Rightarrow (x \geq 0 \wedge y \geq 0 \wedge x + s \leq a \wedge y + s \leq b) \quad (3)$$

$$\left\{ \begin{array}{ll} \forall id_1\ id_2 : leftof(id_1, id_2) & \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : pos(id_1, x_1, y_1) \wedge size(id_1, s_1) \\ & \quad \wedge pos(id_2, x_2, y_2) \wedge x_1 + s_1 \leq x_2 \quad (4) \\ \forall id_1\ id_2 : below(id_1, id_2) & \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : pos(id_1, x_1, y_1) \wedge size(id_1, s_1) \\ & \quad \wedge pos(id_2, x_2, y_2) \wedge y_1 + s_1 \leq y_2 \quad (5) \\ \forall id_1\ id_2 : noOverlap(id_1, id_2) & \leftarrow leftof(id_1, id_2) \vee leftof(id_2, id_1) \\ & \quad \vee\ below(id_1, id_2) \vee below(id_2, id_1) \quad (6) \end{array} \right\}$$

The grounding of this theory can become very large. For example, rule (4) has a grounding size of $n^2a^2b^2$, with $n$ the number of squares and $a$ and $b$ respectively the length and width of the area. However, *pos*, *size* and *area* in fact represent respectively 2, 1 and 2 functional relationships. So, for example, the body of (4) could be replaced by the formula $pos_x(id_1) + size(id_1) \leq pos_x(id_2)$ where $pos_x$ and *size* are functions derived from the predicates *pos* and *size*; this formula has a grounding size of only $n^2$.

A predicate has a functional dependency from a set of arguments $S$ (an index set) to an argument $j$ if a value for the arguments in $S$ uniquely determines the value of the argument $j$. For an $n-1$-ary function, the output can be considered as the $n^{th}$ argument; a function always has a dependency from the $n-1$ input arguments to the output argument, but also other dependencies can be present. A functional dependency can be formalised as a mapping from an index set to an argument position.

*Definition 1* (*Functional dependency*)
Let $P(T_1, \ldots, T_n)/f(T_1, \ldots, T_{n-1}) : T_n$ be the signature of a predicate/function, $S$ an index set over $[1, n]$, $j$ an index in $S^c$, and $\mathcal{T}$ a theory in which $P/f$ occurs. We have a *partial functional dependency* from $S$ to $j$ if, in each model $\mathcal{I}$ of $\mathcal{T}$, it holds that for each tuple $\overline{d}_r \in (T_1 \times \ldots \times T_n)\big|_S$, all tuples $\overline{d}'$ in $P^{\mathcal{I}}/f^{\mathcal{I}}$ for which $\overline{d}'\big|_S = \overline{d}_r$ have the same value for $d'_j$ (*uniqueness*). We have a *total functional dependency* if, in addition, for each tuple $\overline{d}_r \in (T_1 \times \ldots \times T_n)\big|_S$, there is a tuple $\overline{d}'$ in $P^{\mathcal{I}}/f^{\mathcal{I}}$ for which $\overline{d}'\big|_S = \overline{d}_r$ (*existence*).

The *uniqueness* property expresses that a tuple in the index set $S$ maps to at most one value for the $j^{th}$ argument; the *existence* property that there is such a value for each well-typed tuple over $S$. Dependencies as above are denoted by $d\langle P(T_1, \ldots, T_n), S, j\rangle$/ $d\langle f(T_1, \ldots, T_{n-1}) : T_n, S, j\rangle$, a subscript *total* (*partial*) is added to denote total (partial) dependencies whenever relevant. The index set $S$ and the argument $j$ are called, respectively, the *domain* and the *codomain* of the functional dependency. Note that in the case of functions, the codomain (index $n$) of the function can be part of the domain of a dependency. For example, a bijective function $f(T_1) : T_2$ has the dependencies $d\langle f(T_1) : T_2, \{1\}, 2\rangle$ and $d\langle f(T_1) : T_2, \{2\}, 1\rangle$.

*Proposition 1* (*Function constraints*)
For an index set $S$ and index $j$, the existence property holds for a predicate $P(\overline{T})$ (a

function $f(\overline{T}) : T_n$) in a theory $\mathcal{T}$ iff $\mathcal{T}$ entails the *existence constraint*[5]

$$\forall (x_i | i \in S) : \textstyle\bigwedge_{i \in S} T_i(x_i) \Rightarrow \exists (x_i | i \in S^c) : \bigwedge_{i \in S^c} T_i(x_i) \wedge P(\overline{x}) \text{ (predicate).}$$
$$\forall (x_i | i \in S) : \textstyle\bigwedge_{i \in S} T_i(x_i) \Rightarrow \exists (x_i | i \in S^c) : \bigwedge_{i \in S^c} T_i(x_i) \wedge f(\overline{x}) = x_n \text{ (function).}$$

The uniqueness property holds iff $\mathcal{T}$ entails the *uniqueness constraint*

$$\forall \overline{x}\ \overline{x}' : P(\overline{x}) \wedge P(\overline{x}') \wedge \overline{x}\big|_S = \overline{x}'\big|_S \Rightarrow x_j = x'_j \text{ (predicate).}$$
$$\forall \overline{x}\ \overline{x}' : f(\overline{x}) = x_n \wedge f(\overline{x}') = x'_n \wedge (\overline{x} :: x_n)\big|_S = (\overline{x}' :: x_n)\big|_S \Rightarrow x_j = x'_j \text{ (function).}$$

In what follows, shorthands $C_{exists}(P(\overline{T}), S, j)$, respectively, $C_{exists}(f(\overline{T}) : T_n, S, j)$, $C_{unique}(P(\overline{T}), S, j)$ and $C_{unique}(f(\overline{T}) : T_n, S, j)$, are used for these constraints.

Proposition 1 is the basis for a straightforward detection algorithm that iterates over all predicate and function symbols of a given theory $\mathcal{T}$. For each symbol and each of its possible index sets $S$ and argument positions $j \in S^c$, it checks whether the corresponding uniqueness property is entailed by $\mathcal{T}$. If so, we have a partial functional dependency. If, in addition, also the corresponding existence property is entailed, a total functional dependency is detected. Whenever a dependency is detected, the theory can be rewritten to make the dependency explicit (see next section) and the detection algorithm continues with the new theory, until all possible dependencies have been checked.

Using a theorem prover for checking the entailment of the constraints, the algorithm has two issues. First, checking whether a particular property holds may take an excessive amount of time, especially when the property does not hold. So a time-out is necessary. Second, the number of potential dependencies is exponential in the arity of symbols, so another time-out is needed. This means we have to use an anytime algorithm[6] and have to decide on the order in which we iterate over all candidate dependencies. However, the following proposition allows us to prune the search.

*Proposition 2 ((Armstrong 1974))*
For a theory $\mathcal{T}$, a total (partial) functional dependency of a position $j$ on an index set $S$ for a symbol in $\mathcal{T}$ implies a total (partial) functional dependency of $j$ on all index sets of the given symbol that are supersets of $S$ and do not contain $j$.

This proposition offers two opportunities to prune the search. First, if one can prove $\neg C_{unique}(P(\overline{T}), S, j)$ (or $\neg C_{unique}(f(\overline{T}) : T_n, S, j)$, then there is no dependency of $S$ on $j$ and one need not consider subsets of $S$. Second, if a dependency from $S$ to $j$ is found, one need not consider supersets of $S$. Our current implementation only exploits the latter and starts from the smallest candidate index sets (starting with $\emptyset$, i.e., a constant argument). Each time one is found, the theory is rewritten (see next section) and detection continues on the new theory. However, that way, one likely never analyses the largest index sets (due to time-out), while dependencies involving them are quite frequent for predicates. So, for predicates, before exploring index sets from small to large, we first check for a dependency with an index set of size $n - 1$ and store it when found. Then we process index sets from small to large. If the algorithm aborts because of a time-out, the stored dependency, if present and not pruned, is used to rewrite the theory.

[5] $(x_i | i \in S)$ is a shorthand for $(x_{s_1}, \ldots, x_{s_m})$, with $S = \{s_1, \ldots, s_m\}$.
[6] Anytime algorithms can be halted at any point in time and return the best solution found till then.

As already mentioned, we use a theorem prover to check for the functional dependencies. Our current prototype uses SPASS (Weidenbach et al. 2009), a prover for theories in FO. Although the existence and uniqueness constraints are FO formulas, our theories however are in FO$(\cdot)$, which includes definitions and aggregates. In some cases, definitions are equivalent to their completion (which are FO sentences); in other cases, the completion is a weaker theory. However, because a definition entails its completion and entailment is transitive, dependencies derived from the completion also hold for the original theory. Below, we describe two transformations from an FO$(\cdot)$ theory to an FO theory. The first is equivalence preserving, the second produces a weaker theory. Note that standard FO is untyped and only supports total functions.

*Definition 2* (*Strong* FO$(\cdot)$*-to-*FO *transformation*)
The following rewrite rules are applied, in the given order:

1. For every $n$-ary predicate $P(T_1, \ldots, T_n)$, type information is made explicit by adding the sentence $\forall \overline{x} : P(\overline{x}) \Rightarrow T_1(x_1) \wedge \ldots \wedge T_n(x_n)$ (similarly for functions).
2. For every partial function $f(T_1, \ldots, T_{n-1}) : T_n$ do:
   **(i)** Introduce a new predicate symbol $P_f(T_1, \ldots, T_n)$ and add the constraint $C_{unique}(P_f(T_1, \ldots, T_n), [1, n-1], n)$ to $\mathcal{T}$.
   **(ii)** Occurrences $f(\overline{t})$ not at the top level of atoms $f(\overline{t}) = y$ with $y$ a variable, are unnested by introducing a fresh variable.
   **(iii)** Atoms of the form $f(\overline{t}) = y$, with $y$ a variable, are replaced by $P_f(\overline{t} :: y)$.
3. Atoms $\#\{\overline{x} : \varphi\} \geq n$, with $n$ a natural number (cardinality aggregates with a known bound), are replaced by $\exists \overline{x}_1 \ldots \overline{x}_n : (\bigwedge_{i,j \in [1,n], i \neq j} \overline{x}_i \neq \overline{x}_j) \bigwedge_{i \in [1,n]} \varphi[\overline{x}/\overline{x}_i]$. Other comparison operators are rewritten in a similar way.
4. Aggregates are transformed into FO as suggested in Section 5.4 of (Pelov 2004).
5. Definitions $\Delta$ are rewritten using a $voc(\Delta)$-equivalence preserving transformation based on ideas in (Janhunen et al. 2009; Pelov and Ternovska 2005).
6. Formulas $\forall \overline{x} \in \overline{T} : \varphi$ are replaced by $\forall \overline{x} : (\bigwedge_{i \in [1,n]} T_i(x_i)) \Rightarrow \varphi$; similarly for existential quantifications and set closures.

Details on cases 4 and 5 are in Appendix A. These two cases often result in large formulas (with lots of arithmetic) and hence can substantially increase the run-time of the theorem prover. Alternatively, a *weak* FO$(\cdot)$*-to-*FO transformation can be applied. It is identical to the above one except that rule 3 is only applied for small $n$ (e.g., $n < 3$) and that rules 4 and 5 are replaced by the following rules.

**4w-a.** Unnest any aggregate term $agg$ not in an atom $agg = y$ with $y$ a variable, by introduction of a fresh variable, confer rule 2.(ii).
**4w-b.** Replace any positive (negative) atom occurrence containing an aggregate term by true (false) if it is not in a definition; and by $P(\overline{x})$ otherwise, with $\overline{x}$ the free variables of the atom and $P$ a new predicate. For FNNF theories, this results in a weaker theory.
**5w.** Replace a definition by its completion.

*Proposition 3*
Let $\mathcal{T}_s$ be the strong and $\mathcal{T}_w$ be the weak FO$(\cdot)$-to-FO transformation of $\mathcal{T}$. With the understanding that a partial function $f_p$ in $\mathcal{T}$ corresponds to a predicate $P_f$ in $\mathcal{T}_s$ and $\mathcal{T}_w$, it holds that: (i) a dependency is entailed by $\mathcal{T}$ if it is entailed by $\mathcal{T}_s$ and (ii) a dependency entailed by $\mathcal{T}_w$ or $\mathcal{T}_s$ on symbols in $voc(\mathcal{T})$ is entailed by $\mathcal{T}$.

*Proof sketch*
(i) holds because rules 1, 3, 4, and 5 preserve strong equivalence. As for rule 2, a tuple $\overline{d}$ is part of the interpretation of $f$ in a model $\mathcal{I}$ of $\mathcal{T}$ iff it is part of the corresponding model of $P_f$ in $\mathcal{T}_s$. (ii) holds because rule 4w-a preserves strong equivalence and rules 4w-b and 5w make sure that formulas are replaced by weaker formulas, hence models are preserved and only extra models can be created, so no new functional dependencies can be introduced by the transformation. □

*Example 2*
Applying the transformation on the square-packing example replaces constraint (1) (in fact a cardinality constraint) by the sentences $\forall id : id(id) \Rightarrow \exists x\ y : pos(id, x, y)$ and $\forall id_1\ x_1\ y_1\ x_2\ y_2 : pos(id_1, x_1, y_1) \wedge pos(id_1, x_2, y_2) \Rightarrow x_1 = x_2 \wedge y_1 = y_2$ (rule 3). Rule 1 is applied to all predicates; e.g., for the predicate $pos(id, nb, nb)$, the sentence $\forall id\ x\ y : pos(id, x, y) \Rightarrow id(id) \wedge nb(x) \wedge nb(y)$ is added. As for the definition part, rule 5w replaces the three rules by their completion. Equivalence is preserved in this case.

SPASS can prove, e.g., $C_{unique}(pos(id, x, y), 1, 2)$ and $C_{exists}(pos(id, x, y), 1, 2)$ in the resulting theory, i.e., each square has exactly one x-coordinate in models of the theory.

The detection algorithm can be used both *offline* and *online*. In an offline setting, the theory is optimized (often without time bounds) to improve subsequent uses. If the functional dependency is only present in the instance at hand, it might also be worthwhile to do online detection. Consider for example an instance of a graph problem where each node has exactly one outgoing edge. The FO(·)-to-FO transformation can be extended to also transform the input structure into FO sentences in a straightforward way.

## 4 Rewriting the theory

As said in the previous section, each time a dependency is detected, the theory is rewritten into an equivalent theory. First, we explain how a theory is rewritten in case a functional dependency is detected for a symbol that is not defined. Next, we extend the method for defined symbols. Finally we describe how the detection and rewriting of functional dependencies is integrated into our model expansion methodology.

A theory can entail multiple functional dependencies on the same symbol and it is not clear what is the best way to exploit all of them. E.g. for a bijection, we have to decide which one to use in the rewriting. For the other, we have to decide whether to add the constraints to the theory[7]. Different choices affect grounding size and search behavior differently. Our heuristic is to use dependencies that result in symbols with lower arities. Hence we do not look for dependencies on functions with $\#(S) = n - 1$.

*Definition 3* (*dep-reduce*)
The rewriting for a functional dependency $d\langle f(\overline{T}) : T_n, S, j\rangle$ starts with a preprocessing phase. Each rule of the form $\forall \overline{x} : head[f(\bar{t})] \leftarrow \varphi$ is replaced by $\forall \overline{x}, y \in T_n : head[f(\bar{t})/x] \leftarrow y = f(\bar{t}) \wedge \varphi$. Also, if an aggregate term $agg(\{\overline{x} : \varphi : t\})$ has an occurrence of $f(\bar{t})$ in $t$ (or $t$ is $f(\bar{t})$) then it is replaced by $agg(\{\overline{x}, y \in T_n : \varphi \wedge y = t : y\})$. Finally, if $n \in S$, $f$ is replaced by a predicate $P_f$ as follows: atoms $a[f(\bar{t})]$, apart from

[7] Redundant constraints can improve search performance.

$f(\bar{t}) = y$ with $y$ a variable, are replaced by $\exists x \in T_n : a[f(\bar{t})/x] \wedge x = f(\bar{t})$. Next, the atoms of the form $x = f(\bar{t})$ are replaced by $P_f(\bar{t} :: x)$ with $P_f(\overline{T} :: T_n)$ a new predicate. Finally, the constraints $C_{unique}$ and, if $f$ is total, $C_{exists}$ for the functional dependency $d\langle P_f(\overline{T} :: T_n), [1, n-1], n\rangle$ are added to the theory and the input dependency is rewritten as $d\langle P_f(\overline{T} :: T_n), S, j\rangle$.

The main rewriting distinguishes between predicates and functions.

- Let $d\langle f(\overline{T}) : T_n, S, j\rangle$ be a (partial) functional dependency ($n \notin S$) with $\#(S) < n-1$. For the occurrences of $f(\bar{t})$, we identify two cases. In both, a new (partial) function $f_d(\overline{T}|_S) : T_j$ is introduced.
  - $j \neq n$ : A new function $f_r(\overline{T}|_{\{j\}^C}) : T_n$ is added; $f_r$ is partial iff $f$ is. Occurrences of $f(\bar{t})$ are eliminated by replacing atoms $a[f(\bar{t})]$ by $a[f_r(\bar{t}|_{\{j\}^C})] \wedge t_j = f_d(\bar{t}|_S)$.
  - $j = n$ : A new predicate $P_r(\overline{T})$ is introduced. Occurrences of $f(\bar{t})$ are eliminated by replacing atoms $a[f(\bar{t})]$ by $a[f(\bar{t})/f_d(\bar{t}|_S)] \wedge P_r(\bar{t})$.
- Let $d\langle P(\overline{T}), S, j\rangle$ be a (partial) functional dependency for a predicate $P$. If $\#(S) = n-1$, a new (partial) function $f_d(\overline{T}|_S) : T_j$ is introduced and each atom $P(\bar{t})$ is replaced by the atom $t_j = f_d(\bar{t}|_S)$; otherwise, also a predicate $P_r(\overline{T}|_{\{j\}^C})$ is introduced and atoms $P(\bar{t})$ are replaced by $P_r(\bar{t}|_{\{j\}^C}) \wedge t_j = f_d(\bar{t}|_S)$.

To translate a model of the new theory into the alphabet of the original theory, we have to define the removed symbol in terms of the new symbols (in fact, this is only necessary if the symbol is part of the output vocabulary of the problem at hand).

*Definition 4* (*Definition introduction*)

- If the preprocessing replaced $f(\overline{T}) : T_n$ by $P_f(\overline{T} :: T_n)$ then add $\{\forall \overline{x}, x_n \in \overline{T}, T_n : f(\overline{x}) = x_n \leftarrow P_f(\overline{x} :: x_n)\}$.
- If $P(\overline{T})$ was removed, then add $\{\forall \overline{x} \in \overline{T} : P(\overline{x}) \leftarrow f_d(\overline{x}|_S) = x_j\}$ when $\#(S) = n-1$; otherwise, add $\{\forall \overline{x} \in \overline{T} : P(\overline{x}) \leftarrow P_r(\overline{x}|_{\{j\}^C}) \wedge f_d(\overline{x}|_S) = x_j\}$.
- If $f(\overline{T}) : T_n$ was removed, then add $\{\forall \overline{x}, x_n \in \overline{T}, T_n : f(\overline{x}) = x_n \leftarrow f_r(\overline{x}|_{\{j\}^C}) = x_n \wedge f_d(\overline{x}|_S) = x_j\}$ when $j \neq n$ and $n \notin S$; otherwise ($j = n$), add $\{\forall \overline{x}, x_n \in \overline{T}, T_n : f(\overline{x}) = x_n \leftarrow P_r(\overline{x}) \wedge f_d(\overline{x}|_S) = x_n\}$.

*Proposition 4*

Let $\mathcal{T}$ be a theory and $d$ a functional dependency of $\mathcal{T}$. Let $\mathcal{T}$' be the theory obtained after applying the dep-reduce rewriting of Definition 3 for $d$ and the definition introduction of Definition 4. $\mathcal{T}$' is strongly voc($\mathcal{T}$)-equivalent with $\mathcal{T}$.

The proof is included in Appendix B.

*Example 3*

Consider rule (4) of our packing problem; applying dep-reduce for the functional dependency $d_{total}\langle pos_r(id, nb), \{1\}, 2\rangle$ introduces a function we rename as $pos_x(id) : nb$ and a relation $pos_r(id, nb)$. For the new theory, another functional dependency can be proven, namely $d_{total}\langle pos_r(id, nb), \{1\}, 2\rangle$. Again applying dep-reduce introduces $pos_y(id) : nb$. After these two steps, rule (4) is rewritten into:

$$\left\{ \begin{array}{l} \forall id_1\ id_2 : leftof(id_1, id_2) \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : y_1 = pos_y(id_1) \wedge x_1 = pos_x(id_1) \\ \qquad\qquad \wedge\ size(id_1, s_1) \wedge y_2 = pos_y(id_2) \wedge x_2 = pos_x(id_2) \wedge x_1 + s_1 \leq x_2 \end{array} \right\}$$

While we have now replaced symbols by symbols of lower arity, we have not eliminated any variables. However, postprocessing can do so. For example the body of the rule of the above example can be simplified into $\exists s_1 : size(id_1, s_1) \wedge pos_x(id_1) + s_1 \leq pos_x(id_2)$. The FO$(\cdot)^{\text{IDP}}$ grounder, which aims at grounding human-written theories, does a poor job on such formulas. We preprocess the theory by the following set of equivalence preserving rewrite rules (applied on formulas in FNNF)[8].

- The atom $x = f(\bar{t})$ ($\neg(x = f(\bar{t}))$) is a conjunct (disjunct) of a conjunction (disjunction) $\varphi$: Replace $x$ by $f(\bar{t})$ in the other conjuncts (disjuncts) of $\varphi$.
- A formula $agg(\{\overline{x}, y : y = f(\bar{t}) \wedge \varphi : y\})$. Replace it by $agg(\{\overline{x} : \varphi : f(\bar{t})\})$.
- A rule $\forall \overline{x} : a \leftarrow f(\bar{t}) = x \wedge \varphi$: Replace $x$ in $a$ by $f(\bar{t})$.
- A formula $\exists x : \varphi$ and the only occurrence of $x$ is in a conjunct $x = f(\bar{t})$ of $\varphi$: If $f$ is total, remove the conjunct; otherwise replace it by $HasImage(f(\bar{t}))$.
- A formula $\forall x : \varphi$ and the only occurrence of $x$ is in a disjunct $\neg(x = f(\bar{t}))$ of $\varphi$: If $f$ is total, remove the disjunct; otherwise replace it by $\neg HasImage(f(\bar{t}))$[9].
- A rule $\forall \overline{x} : a \leftarrow f(\bar{t}) = x \wedge \varphi$ and the only occurrence of $x$ is in $f(\bar{t}) = x$: If $f$ is total, remove the conjunct; otherwise replace it by $HasImage(f(\bar{t}))$.
- A formula $\forall x : \varphi$ or $\exists x : \varphi$ such that $x$ does not occur in $\varphi$: Replace it by $\varphi$.

*Example 4*
Consider the problem of scheduling courses at a university. A naive modeler might use a symbol *planned*/5 to associate a session with a student group, a classroom, a time slot and a teacher all at once. The restriction that a teacher cannot teach multiple sessions at the same time might then be expressed by $\forall sid\ sg\ c\ ts\ te : planned(sid, sg, c, ts, te) \Rightarrow \neg\exists sid_2\ sg_2\ c_2 : sid_2 \neq sid \wedge planned(sid_2, sg_2, c_2, ts, te)$, which has an impractical grounding size in the order of $\|sessions\|^2 \times \|groups\|^2 \times \|rooms\|^2 \times \|slots\| \times \|teachers\|$ atoms.

As all those relations are functional, function detection and rewriting will split *planned* in four function symbols and produce the sentence $\forall sid\ sid_2 : sid \neq sid_2 \wedge teacher(sid) = teacher(sid_2) \Rightarrow \neg(timeslot(sid) = timeslot(sid_2))$. This is the theory an experienced modeler would construct, but generated from the specification of an inexperienced user.

So far, we have only handled dependencies for non-defined symbols. When the symbol is defined, the rewriting dep-reduce is first applied to all atoms except the heads of rules. Afterwards, we have to replace those by rules for the new symbols introduced by dep-reduce. This is achieved by the following definition which distinguishes several cases.

*Definition 5 (Define new symbols)*
- A dependency $d\langle f(\overline{T}) : T_n, S, j\rangle$ with $j \neq n$ and $n \notin S$. Each rule $f(\overline{x}) = x_n \leftarrow \varphi$ is replaced by $f_d(\overline{x}|_S) = x_j \leftarrow \varphi$ and $f_r(\overline{x}|_{\{j\}^C}) = x_n \leftarrow \varphi$.
- A dependency $d\langle f(\overline{T}) : T_n, S, j\rangle$ with $j = n$. Each rule $f(\overline{x}) = x_n \leftarrow \varphi$ is replaced by $f_d(\overline{x}|_S) = x_n \leftarrow \varphi$ and $P_r(\overline{x}) \leftarrow \varphi$.
- A dependency $d\langle P(\overline{T}), S, j\rangle$. Each rule $P(\overline{x}) \leftarrow \varphi$ is replaced by $f_d(\overline{x}|_S) = x_j \leftarrow \varphi$ and $P_r(\overline{x}|_{\{j\}^C}) \leftarrow \varphi$; the latter only if $\#(S) < n - 1$.

[8] In the rewriting, only atoms $x = f(\bar{t})$ are used where the type of $x$ is the output type of $f$. If the type of $x$ is empty, the atom is false.
[9] $\forall x : \neg(x = f(\bar{t}))$ is equivalent with $\neg\exists x : x = f(\bar{t})$ which is equivalent with $\neg HasImage(f(\bar{t}))$.

*Example 5*
Consider a definition of nodes reachable from a given start node $s$ and the minimal cost for reaching them, given a graph $e(node, node)$ and a cost function $c(node, node) : weight$:

$$\left\{ \begin{array}{ll} r(s,0) & \leftarrow \\ \forall x\ y\ c_o : r(x, c_o + c(y,x)) & \leftarrow e(y,x) \wedge r(y,c_o) \\ \qquad \wedge \forall y'\ c'_o : \neg(y = y') \wedge r(y',c'_o) \wedge e(y',x) \Rightarrow c(y,x) + c_o \leq c(y',x) + c'_o & \end{array} \right\}$$

The theory entails that the cost is a partial function on nodes, so can be rewritten as

$$\left\{ \begin{array}{ll} f_r(s) = 0 & \leftarrow \\ \forall x\ y : f_r(x) = f_r(y) + c(y,x) & \leftarrow e(y,x) \\ \qquad \wedge \forall y' : \neg(y = y') \wedge e(y',x) \Rightarrow c(y,x) + f_r(y) \leq c(y',x) + f_r(y') & \end{array} \right\}$$

The current solver has no support for *defined* function symbols; however further transformations make exploitation possible. See Appendix C for details.

*Integration within model expansion.* When our theory rewriting is done as part of a model generation inference task, we cannot just replace a symbol with a set of symbols of lower arity. Indeed, the input structure uses the original alphabet and also the model should be presented to the user in the original alphabet. The rules given in Definition 4 provide the link between both alphabets. They are used during the grounding phase to translate the given input structure into corresponding input in the new alphabet. They are not needed during the solving (hence need not be grounded), but are used again to translate the model expressed in the new alphabet back to the original alphabet, which can also be done efficiently (without grounding) by bottom-up evaluation of the definitions.

## 5 Experiments

Our implementation uses the IDP system(Bogaerts et al. 2012), a knowledge-base system supporting state-of-the-art model expansion, as can be observed from previous ASP competitions (Denecker et al. 2009; Calimeri et al. 2012). As back-end, IDP uses the search algorithm CONSTRAINT(ID) (De Cat et al. 2013), which integrates SAT with, among others, unfounded set detection, aggregates and finite-domain constraints. To detect dependencies, we used the award-winning prover SPASS (Weidenbach et al. 2009). As benchmarks, we used faithful translations of the ASP-Core-2[10] encodings to FO$(\cdot)^{\text{IDP}}$ and instances of the 2013 ASP competition[11]. We added type information (required by IDP) and constraints on the input structure if these were specified in the problem description (which were often not modeled, but are crucial for offline detection).

For the resulting encodings, we did two types of experiments.[12] In the first series of experiments, the detection and rewriting algorithm was applied to each of the encodings with one selected instance to measure how many functional dependencies were detected and how much the rewriting reduced the number of quantified variables.

[10] ASP-Core-2 supports no function symbols except aggregate functions.
[11] See `https://www.mat.unical.it/aspcomp2013/OfficialProblemSuite`
[12] All experiments were run on an 64-bit Ubuntu 12.10 system with a quad-core 2.53 Ghz processor and 4 Gb of RAM. All benchmarks, experimental data and results can be found on `http://dtai.cs.kuleuven.be/krr/files/experiments/iclp2013-function-experiments.tar.gz`

The details of the experimental results can be found in Appendix D; here we only provide a summary. Out of 19 benchmarks, functional dependencies were detected in all but 3 benchmarks. In those 16 benchmarks, 45% of the detected dependencies were partial, of which 75% were detected in two benchmarks. The subsequent rewrite transformation erased on average 52% of all quantified variables, with peaks above 85%, and was able to strongly reduce the size of the grounding. Total detection time ranged from less than 1 second to 450 seconds and was directly proportional to the number of symbols and their arity, as most calls to SPASS timed out (a 2 second timeout was used).

Close inspection showed that the prover was unable to detect functional dependencies in constraints of the form $\forall x : P(x,t) \Leftrightarrow (x = initialvalue \wedge t = 0) \vee (\dots P(y, t-1) \dots)$, which occur frequently when reasoning over time (e.g., in planning problems). Indeed, SPASS does not support the required inductive reasoning. While one could organize it (prove first for $t = 0$, then the induction step), our current implementation does not.

A second series of experiments was performed to evaluate the effect on the solver's performance. These results are only preliminary, partly because at the time of writing, MINISAT(ID) only supported total functions with a numeric codomain. The results are promising however: for each benchmark, the number of solved instances often increased significantly while the running times improved substantially for the harder problems.

To summarize, offline detection of functional dependencies is certainly worthwhile, as detected dependencies can result in a significant performance boost for the solver, while the performance is unaffected when none are detected. Whether to use online detection depends on the application at hand as the proving overhead could be significant.

## 6 Concluding remarks

The main contribution of this work is to use FO theorem proving to detect functional dependencies in declarative problem statements and to exploit these dependencies by rewriting the theory. This reduces the size of the grounding; moreover, the grounding can exploit the constraint programming features of the latest generation of grounders and search algorithms. Preliminary experimental results show that many functional dependencies can indeed be automatically detected and that the effect on grounding size and solver performance is often significant. Part of future work is to extract other types of implicit knowledge, such as smaller types, definitional totality and more complex finite-domain constraints such as *all-different*. In the field of CP, Mears et al. (2008) search for symmetries in a problem specification using a different approach, namely by generating multiple solutions for several instances, from which symmetry candidates are extracted, which can then be verified using e.g., theorem provers. Wittocx et al. (2013) have studied unit propagation on a symbolic level by deriving formulas entailed by the theory by reasoning on unit propagation schemes . It remains an open question whether theorem provers with additional reasoning capabilities (e.g. arithmetic), such as Melia (Baumgartner et al. 2012), Princess (Rümmer 2008) and Z3 (de Moura and Bjørner 2008), are capable of proving the presence of even more functional dependencies.

*Acknowledgements* Broes De Cat is funded by the Institute for Science and Technology Flanders (IWT). We thank Philipp Rümmer for sharing with us his insights into theorem provers and the reviewers for their valuable comments.

## References


Apt, K. R. 2003. *Principles of Constraint Programming.* Cambridge University Press.

Armstrong, W. W. 1974. Dependency structures of data base relationships. *IFIP Congress*, 580–580.

Balduccini, M. 2011. Industrial-size scheduling with asp+cp. In *LPNMR*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 284–296.

Balduccini, M. 2012. A "conservative" approach to extending answer set programming with non-herbrand functions. In *Correct Reasoning*, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds. Lecture Notes in Computer Science, vol. 7265. Springer Berlin Heidelberg, 24–39.

Baral, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, New York, NY, USA.

Bartholomew, M. and Lee, J. 2012. Stable models of formulas with intensional functions. See Brewka et al. (2012).

Baumgartner, P., Pelzer, B., and Tinelli, C. 2012. Model evolution with equality - revised and implemented. *J. Symb. Comput. 47,* 9, 1011–1045.

Bogaerts, B., De Cat, B., De Pooter, S., and Denecker, M. 2012. IDP website. `http://dtai.cs.kuleuven.be/krr/software/idp`.

Brewka, G., Eiter, T., and McIlraith, S. A., Eds. 2012. *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012.* AAAI Press.

Cabalar, P. 2011. Functional answer set programming. *TPLP 11,* 2-3, 203–233.

Cabalar, P. 2013. Setting the stage for asp functions. *The Association for Logic Programming Newsletter 26,* 2 (june).

Calimeri, F., Ianni, G., and Ricca, F. 2012. The third open answer set programming competition. *CoRR abs/1206.3111.*

De Cat, B., Bogaerts, B., Devriendt, J., and Denecker, M. 2013. Model expansion in the presence of function symbols using constraint programming. Tech. Rep. CW 644, Departement of Computer Science, Katholieke Universiteit Leuven.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *TACAS*, C. R. Ramakrishnan and J. Rehof, Eds. LNCS, vol. 4963. Springer, 337–340.

Denecker, M. and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL) 9,* 2 (Apr.), 14:1–14:52.

Denecker, M., Vennekens, J., Bond, S., Gebser, M., and Truszczyński, M. 2009. The second Answer Set Programming competition. In *LPNMR*. 637–654.

Erdem, E., Lin, F., and Schaub, T., Eds. 2009. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings.* LNCS, vol. 5753. Springer.

Gebser, M., Ostrowski, M., and Schaub, T. 2009. Constraint answer set solving. In *ICLP*, P. M. Hill and D. S. Warren, Eds. LNCS, vol. 5649. Springer, 235–249.

Janhunen, T., Niemelä, I., and Sevalnev, M. 2009. Computing stable models via reductions to difference logic. See Erdem et al. (2009), 142–154.

Lierler, Y. 2012. On the relation of constraint answer set programming languages and algorithms. In *AAAI*, J. Hoffmann and B. Selman, Eds. AAAI Press.

Lifschitz, V. 2012. Logic programs with intensional functions. See Brewka et al. (2012).

Lin, F. and Wang, Y. 2008. Answer set programming with functions. In *KR*, G. Brewka and J. Lang, Eds. AAAI Press, 454–465.

Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P. J., de la Banda, M. G., and Wallace, M. 2008. The design of the Zinc modelling language. *Constraints 13,* 3, 229–267.

Mears, C., de la Banda, M. J. G., Wallace, M., and Demoen, B. 2008. A novel approach

for detecting symmetries in csp models. In *CPAIOR*, L. Perron and M. A. Trick, Eds. Lecture Notes in Computer Science, vol. 5015. Springer, 158–172.

Niemelä, I. 2006. Answer set programming: A declarative approach to solving search problems. In *JELIA*. 15–18. Invited talk.

Ostrowski, M. and Schaub, T. 2012. Asp modulo csp: The clingcon system. *TPLP 12,* 4-5, 485–503.

Pelov, N. 2004. Semantics of logic programs with aggregates. Ph.D. thesis, K.U.Leuven, Leuven, Belgium.

Pelov, N. and Ternovska, E. 2005. Reducing inductive definitions to propositional satisfiability. In *ICLP*, M. Gabbrielli and G. Gupta, Eds. LNCS, vol. 3668. Springer, 221–234.

Rümmer, P. 2008. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, I. Cervesato, H. Veith, and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 5330. Springer, 274–289.

Van Gelder, A. 1993. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences 47,* 1, 185–221.

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., and Wischnewski, P. 2009. Spass version 3.5. In *CADE*, R. A. Schmidt, Ed. Lecture Notes in Computer Science, vol. 5663. Springer, 140–145.

Wittocx, J., Denecker, M., and Bruynooghe, M. 2013. Constraint propagation for first-order logic and inductive definitions. *ACM Transactions on Computational Logic*. Accepted.