# The Intelligent Grounder of DLV*

Wolfgang Faber, Nicola Leone, and Simona Perri

Department of Mathematics, University of Calabria,
P.te P. Bucci, Cubo 30B, I-87036 Rende, Italy
{faber,leone,perri}@mat.unical.it

**Abstract.** In this work, we give an overview of the DLV *Intelligent Grounder*, one of the most popular Answer Set Programming instantiators, and a very strong point of the DLV system. Based on a variant of semi-naive evaluation, it also includes several advanced optimization techniques and supports a number of application-oriented features which allow for the successful exploitation of DLV in real-world contexts, also at an industrial level.

## 1 Introduction

Answer Set Programming (ASP) [18, 19, 11, 32] is a powerful logic-based programming language which is enjoying increasing interest within the scientific community also thanks to the availability of a number of efficient implementations [27, 15, 23, 29, 30, 2]. DLV [27] is one of the most successful and widely used ASP systems, which has stimulated some interest also in industry. DLV's implementation is based on solid theoretical foundations. It relies on advanced optimization techniques and sophisticated data structures.

The computation of answer sets in DLV is characterized by two phases, namely *program instantiation (grounding)* and *answer set search*. The former transforms the input program into a semantically equivalent one with no variables (ground) and the latter applies propositional algorithms on the instantiated program to generate answer sets.

Grounding in DLV is much more than a simple replacement of variables by all possible ground terms: It partially evaluates relevant program fragments, and efficiently produces a ground program which has precisely the same answer sets as the full one, but is much smaller in general. In order to highlight these features, we qualify it as Intelligent Grounder. Notably, it has the power of a full-fledged deductive database system, able to completely solve deterministic programs, most notably normal stratified programs. Moreover, it allows for dealing with recursive function symbols, and thus for expressing every computable function. More in detail, the Intelligent Grounder is able to finitely evaluate every program belonging to the powerful class of *finitely-ground ($\mathcal{FG}$)* programs with recursive functions defined in [5].

* Partially supported by the Regione Calabria and the EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l.

The DLV grounder plays a key role for the successful deployment of DLV in real-world contexts. Indeed, it incorporates many algorithms for improving the performance (see Section 5), including database optimization strategies and parallel evaluation techniques allowing for dealing with data-intensive applications. Moreover, it is endowed with a number of mechanisms that meet the requirement of real-world applications and make its usage feasible in practice: *database interoperability* is possible by means of an ODBC interface [25, 38] which allows for both importing input data from and exporting answer set data to an external database; *plugin functions* [4] provide a framework for integrating application specific functions in logic rules and dealing with external sources of computation; *non-ground queries* allow for advanced reasoning.

In this paper we overview the DLV Intelligent Grounder, focusing on its input language, the main evaluation strategy, and recalling the most relevant optimization techniques. While the software has been available for quite some time, this work provides its first comprehensive description, including all features up to the recent release of 2011-12-21.

## 2 The DLV system and its applications

In this section we provide an overview of the DLV system, in which the Intelligent Grounder is embedded, focussing on its use within projects and applications. The DLV project has been active for more than fifteen years, encompassing first the development and later on the continuous enhancement of the DLV system.

The DLV system offers a range of advanced knowledge modeling features, providing support for declarative problem solving. Its language extends basic Answer Set Programming (ASP) with a number of constructs, including aggregates [14], weak constraints [27], functional terms [5]. The latter considerably increase the expressiveness of the language, which is important in several real-world contexts. In addition, the system incorporates several front-ends for dealing with specific applications. Concerning efficiency, DLV is competitive with the most advanced systems in the area of ASP, as confirmed also by the results of the First and Second ASP System Competitions [16, 10], in which DLV won the MGS (Modeling, Grounding, Solving) category and the class of decision problems in P, respectively. DLV did not participate in the most recent competition [6], as the project team was organizing the event.

DLV is widely used by researchers all over the world: it is used for educational purposes in courses on Databases and Artificial Intelligence, both in European and American universities; it has been employed at CERN, the European Laboratory for Particle Physics, for a deductive database application; the Polish company Rodan Systems S.A. has exploited DLV in a tool for the detection of price manipulations and unauthorized use of confidential information, used by the Polish Securities and Exchange Commission. The European Commission funded a project on Information Integration, which produced a sophisticated and efficient data integration system based on DLV, called INFOMIX[24].

Notably, DLV has stimulated quite some interest also in industry. Most industrial applications of DLV are currently supervised by two spin-off companies of the University of Calabria, EXEURA and DLVSYSTEM. EXEURA uses DLV in its line of knowledge management products intended for resale, which includes OntoDLV [35], a system for ontology specification and reasoning, OLEX [37], a corporate classification system supporting the entire content classification lifecycle, and HiLεX [36], a system for ontology-based information extraction from unstructured documents. DLVSYSTEM, on the other hand, maintains the DLV system itself and supervises direct deployment of DLV in industrial applications by means of consulting. The main industrial applications that directly employ DLV currently are IDUM [22], an intelligent e-tourism system, and a system for *Team Building at the seaport of Gioia Tauro* [20], used by the transshipment company ICO BLG. For more details on applications of DLV, see [21].

## 3 The Input Language

In this section we first provide syntax and semantics of the core language of DLV (disjunctive logic programs with functional terms) and then overview some linguistic extensions. A number of examples illustrate the knowledge modeling features of the language.

**Core Language**
A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant or a variable. If $t_1 \ldots t_n$ are terms and $f$ is a function symbol of arity $n$, then $f(t_1, \ldots, t_n)$ is a *functional term*. If $t_1, \ldots, t_k$ are terms and $p$ is a *predicate symbol* of arity $k$, then $p(t_1, \ldots, t_k)$ is an *atom*. A *literal* $l$ is of the form $a$ or not $a$, where $a$ is an atom; in the former case $l$ is *positive*, otherwise *negative*[1]. A *rule* $r$ is of the form $\alpha_1 \mathtt{v} \cdots \mathtt{v}\, \alpha_k \mathtt{:-}\ \beta_1, \ldots, \beta_n, \text{not}\, \beta_{n+1}, \ldots, \text{not}\, \beta_m.$ where $m \geq 0$, $k \geq 0$; $\alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_m$ are atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ (the *head* of $r$) and $B(r) = B^+(r) \cup B^-(r)$ (the *body* of $r$), where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ (the *positive body* of $r$) and $B^-(r) = \{\text{not}\ \beta_{n+1}, \ldots, \text{not}\ \beta_m\}$ (the *negative body* of $r$). If $H(r) = \emptyset$ then $r$ is a *(strong) constraint*; if $B(r) = \emptyset$ and $|H(r)| = 1$ then $r$ is a *fact*.

A rule $r$ is safe if each variable of $r$ has an occurrence in $B^+(r)$. A DLV program is a finite set $P$ of safe rules. A program (a rule, a literal) is said to be *ground* if it contains no variables. A predicate is defined by a rule if the predicate occurs in the head of the rule. A predicate defined only by facts is an *EDB* predicate, the remaining predicates are *IDB* predicates. The set of all facts in $P$ is denoted by $Facts(P)$; the set of instances of all EDB predicates in $P$ is denoted by $EDB(P)$.

Given a program $P$, the *Herbrand universe* of $P$, denoted by $U_{\mathcal{P}}$, consists of all (ground) terms that can be built combining constants and function symbols appearing in $P$. The *Herbrand base* of $P$, denoted by $B_{\mathcal{P}}$, is the set of all ground

[1] Note that the DLV language also supports strong negation; however, for simplicity, in this paper we do not consider it, since it is irrelevant for the instantiation process.

atoms obtainable from the atoms of $P$ by replacing variables with elements from $U_{\mathcal{P}}$. A *substitution* for a rule $r \in P$ is a mapping from the set of variables of $r$ to the set $U_{\mathcal{P}}$ of ground terms. A *ground instance* of a rule $r$ is obtained applying a substitution to $r$. The *instantiation (grounding)* $Ground(\mathcal{P})$ of $P$ is defined as the set of all ground instances of its rules over $U_{\mathcal{P}}$. An *interpretation* $I$ for $P$ is a subset of $B_{\mathcal{P}}$. A positive literal $a$ (resp., a negative literal not $a$) is true w.r.t. $I$ if $a \in I$ (resp., $a \notin I$); it is false otherwise. Given a ground rule $r$, we say that $r$ is satisfied w.r.t. $I$ if some atom appearing in $H(r)$ is true w.r.t. $I$ or some literal appearing in $B(r)$ is false w.r.t. $I$. Given a ground program $P$, we say that $I$ is a *model* of $P$, iff all rules in $Ground(\mathcal{P})$ are satisfied w.r.t. $I$. A model $M$ is *minimal* if there is no model $N$ for $P$ such that $N \subset M$. The *Gelfond-Lifschitz reduct* [19] of $P$, w.r.t. an interpretation $I$, is the positive ground program $P^I$ obtained from $Ground(\mathcal{P})$ by: $(i)$ deleting all rules having a negative literal false w.r.t. $I$; $(ii)$ deleting all negative literals from the remaining rules. $I \subseteq B_{\mathcal{P}}$ is an *answer set* for a program $P$ iff $I$ is a minimal model for $P^I$. The set of all answer sets for $P$ is denoted by $AS(P)$.

It is worthwile noting that, even disregarding the extensions that will be presented in the next subsection, the DLV core language is quite expressive. Indeed, its function-free fragment allows us to express, in a precise mathematical sense, every property of finite structures over a function-free first-order structure that is decidable in nondeterministic polynomial time with an oracle in NP [11] (i.e., it captures the complexity class $\Sigma_2^P$). Thus, even this fragment allows for encoding problems that cannot be translated to SAT in polynomial time. Importantly, the encoding of a large variety of problems is very concise, simple, and elegant.

*Example 1.* Consider the following problem, called EXAM-SCHEDULING, which consists of scheduling examinations for courses. In particular, we want to assign exams to time slots such that no two exams are assigned for the same time slot if the respective courses have a student in common (we call such courses "incompatible"). Supposing that there are three time slots available, namely, $ts_1$, $ts_2$ and $ts_3$, we express the problem by the following program $P_{sch}$:

$$r_1: \ assign(X, ts_1) \vee assign(X, ts_2) \vee assign(X, ts_3) :- course(X).$$
$$s_1: \ :- assign(X, S), assign(Y, S), incompatible(X, Y).$$

Here we assume that the courses and the pair of incompatible courses are specified by a set $F$ of input facts with predicate *course* and *incompatible*, respectively. Rule $r_1$ says that every course is assigned to one of the three time slots; strong constraint $s_1$ (a rule with empty head) expresses that no two incompatible courses can be overlapped, that is, they cannot be assigned to the same time slot. There is a one-to-one correspondence between the solutions of the EXAM-SCHEDULING problem and the answer sets of $P_{sch} \cup F$.

**Linguistic Extensions**

An important feature of the DLV language are *weak constraints* [27], which allow for expressing optimization problems. A weak constraint is denoted like a strong constraint, but using the symbol $:\sim$ instead of $:-$. Intuitively, weak constraints

allow for expressing conditions that *should* be satisfied, but not necessarily have to be. The informal meaning of a weak constraint $:\sim B.$ is "$B$ should preferably be false". Additionally, a weight and a priority level for the weak constraint may be specified enclosed in square brackets (by means of positive integers or variables). When not specified, these values default to 1. Optimal answer sets are those minimizing the sum of weights of the violated weak constraints in the highest priority level and, among them, those which minimize the sum of weights of the violated weak constraints in the next lower level, and so on. Weak constraints allow us to express "desiderata" and are very useful in practice, since they allow for obtaining a solution (answer set) also when the usage of strong constraints would imply that there is no answer set.

*Example 2.* In specific instances of EXAM-SCHEDULING, there could be no way to assign courses to time slots without having some overlapping between incompatible courses. However, in real life, one is often satisfied with an approximate solution, that is, one in which constraints are satisfied as much as possible. In this light, the problem at hand can be restated as follows (APPROX-SCHEDULING): "assign exams to time slots trying to not overlap incompatible courses". This can be expressed by the program $P_{asch}$ using weak constraints:

$$r_1: \quad assign(X, ts_1) \vee assign(X, ts_2) \vee assign(X, ts_3) :\!- course(X).$$
$$w_1: \quad :\sim assign(X, S), assign(Y, S), incompatible(X, Y).$$

An informal reading of the above weak constraint $w_1$ is: "preferably, do not assign the exams $X$ and $Y$ to the same time slot if they are incompatible". Note that the above two programs $P_{sch}$ and $P_{asch}$ have exactly the same answer sets if all incompatible courses can be assigned to different time slots. However, when $P_{sch}$ has no answer set, $P_{asch}$ provides answer sets corresponding to ways to satisfy the problem constraints "as much as possible".

The DLV language also supports *aggregate atoms* [14], allowing for representing in a simple and natural manner also properties that require the use of arithmetic operators on (multi-)sets, often arising in real-world applications. Aggregate atoms consist of an aggregation function (currently one of cardinality, sum, product, maximum, minimum), evaluated over a multiset of terms, the content of which depend on the truth of non-aggregate atoms. The syntax is $L \prec_1 \mathbf{F}\{Vars : Conj\} \prec_2 U$ where $\mathbf{F}$ is a function among `#count`, `#min`, `#max`, `#sum`, and `#times`, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, $L$ and $U$ are integers or variables, called guards, and $\{Vars : Conj\}$ is a symbolic set, which intuitively represents the set of values for $Vars$ for which the conjunction $Conj$ is true. For instance, the symbolic set $\{X, Y{:}a(X, Y, Z), \text{not } p(Y)\}$ stands for the set of pairs $(X, Y)$ satisfying the conjunction $a(X, Y, Z), \text{not } p(Y)$, i.e., $S = \{(X, Y) \mid \exists Z : a(X, Y) \wedge \text{not } p(Y) \textit{ is true}\}$. When evaluating an aggregate function over it, the projection on the first elements of the pairs is considered, which yields a multiset in general. The value yielded by the function evaluation is compared against the guards, determining the truth value of the aggregate.

*Example 3.* Consider, for instance, a TEAM-BUILDING problem, where a project team has to be built according to the following specifications:

($p_1$) The team consists of a certain number of employees.
($p_2$) At least a given number of different skills must be present in the team.
($p_3$) The sum of the salaries of the employees working in the team must not exceed the given budget.
($p_4$) The salary of each individual employee is within a specified limit.
($p_5$) The team must include at least a given number of female employees.

Information on the employees is provided by a number of facts of the form $emp(EmpId, Sex, Skill, Salary)$. The size of the team, the minimum number of different skills in the team, the budget, the maximum salary, and the minimum number of female employees are given by facts $nEmp(N)$, $nSkill(N)$, $budget(B)$, $maxSal(M)$, and $women(W)$. We then encode each property $p_i$ above by an aggregate atom $A_i$, and enforce it by an integrity constraint containing not $A_i$.

$r_1:\ in(I) \vee out(I) \text{ :- } emp(I, Sx, Sk, Sa).$
$s_1:\ \text{ :- } nEmp(N), \text{not } \#\texttt{count}\{I : in(I)\} = N.$
$s_2:\ \text{ :- } nSkill(M), \text{not } \#\texttt{count}\{Sk : emp(I, Sx, Sk, Sa), in(I)\} \geq M.$
$s_3:\ \text{ :- } budget(B), \text{not } \#\texttt{sum}\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\} \leq B.$
$s_4:\ \text{ :- } maxSal(M), \text{not } \#\texttt{max}\{Sa : emp(I, Sx, Sk, Sa), in(I)\} \leq M.$
$s_5:\ \text{ :- } women(W), \text{not } \#\texttt{count}\{I : emp(I, f, Sk, Sa), in(I)\} \geq W.$

Intuitively, $r_1$ "guesses" whether an employee is included in the team or not, while each constraint $s_1$-$s_5$ corresponds one-to-one to a requirement $p_1$-$p_5$.

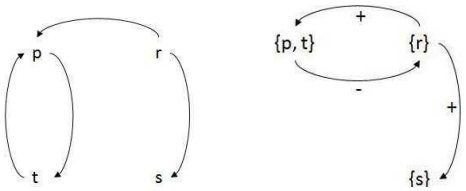
## 4 Basic Grounding Methods

All currently competitive ASP systems mimic the definition of the semantics as given in Section 3 by first creating a ground program without variables. This phase is usually referred to as *grounding* or *instantiation*. The program created is usually a subset of the ground program as defined in Section 3. Still, this task is computationally expensive (see [11, 9]), and its efficiency is important for the performance of the entire system. Indeed, the grounding frequently forms a bottleneck and is crucial in real-world applications involving large input data.

In this Section we give a general description of the DLV grounder. We first describe the algorithm for instantiating programs of the DLV core language and then briefly discuss the linguistic extensions. Note that the DLV core language also permits function symbols, which can imply non-termination of the instantiation. We hence examine *finitely-ground* programs [5] for which the DLV grounder is guaranteed to terminate and explain how to identify programs of this class.

### 4.1 Dependency and Component Graphs

Given an input program $\mathcal{P}$, the DLV grounder creates a subset of $Ground(\mathcal{P})$, whose ground rules only contain literals that can potentially become true. To this end, structural information of the input program is analyzed. The *Dependency Graph* of a program $\mathcal{P}$ is a directed graph $G_\mathcal{P} = \langle N, E \rangle$, where $N$ is the set of IDB predicates of $\mathcal{P}$, and $E$ contains an arc $(p, q)$ if there is a rule $r$ in $\mathcal{P}$ such




$G_{\mathcal{P}}$ $G^c_{\mathcal{P}}$

**Fig. 1.** Dependency and Component Graphs.

that $q$ occurs in the head of $r$ and $p$ occurs in a positive literal of the body of $r$. The graph $G_{\mathcal{P}}$ induces a partition of $\mathcal{P}$ into subprograms (also called *modules*) allowing for a modular evaluation: For each strongly connected component (SCC) $C$ of $G_{\mathcal{P}}$ (a set of predicates), the set of rules defining the predicates in $C$ is called *module* of $C$ and is denoted by $\mathcal{P}_C$. A rule $r$ occurring in a module $\mathcal{P}_C$ (i.e., defining some predicate $q \in C$) is said to be *recursive* if there is a predicate $p \in C$ in the positive body of $r$; otherwise, $r$ is said to be an *exit rule*.

*Example 4.* Consider the following program $\mathcal{P}$, where $a$ is an EDB predicate:

$$a(g(1)). \quad t(X, f(Y)) \text{ :- } p(X,Y), a(Y). \quad p(g(X),Y) \vee s(Y) \text{ :- } r(X), r(Y).$$
$$p(X,Y) \text{ :- } r(X), t(X,Y). \qquad r(X) \text{ :- } a(g(X)), \mathit{not}\ t(X, f(X)).$$

Graph $G_{\mathcal{P}}$ is illustrated in Figure 1; the strongly connected components of $G_{\mathcal{P}}$ are $\{s\}$, $\{r\}$ and $\{p,t\}$. They correspond to the three following modules:

$$\mathcal{P}_{\{s\}} = \{p(g(X),Y) \vee s(Y) \text{ :- } r(X), r(Y).\ \}$$
$$\mathcal{P}_{\{r\}} = \{r(X) \text{ :- } a(g(X)), \mathit{not}\ t(X, f(X)).\ \}$$
$$\mathcal{P}_{\{p,t\}} = \{p(g(X),Y) \vee s(Y) \text{ :- } r(X), r(Y). \quad p(X,Y) \text{ :- } r(X), t(X,Y).$$
$$t(X, f(Y)) \text{ :- } p(X,Y), a(Y).\ \}$$

Moreover, $\mathcal{P}_{\{s\}}$ and $\mathcal{P}_{\{r\}}$ do not contain recursive rules, while $\mathcal{P}_{\{p,t\}}$ contains one exit rule ($p(g(X),Y) \vee s(Y) \text{ :- } r(X), r(Y).$) and two recursive rules.

The *Component Graph* of a program $\mathcal{P}$ is a directed labeled graph $G^c_{\mathcal{P}} = \langle N, E, lab \rangle$, where $N$ is the set of strongly connected components of $G_{\mathcal{P}}$, and $E$ contains an arc $(B, A)$ with $lab((B,A)) =$"+", if there is a rule $r$ in $\mathcal{P}$ such that $q \in A$ occurs in the head of $r$ and $p \in B$ occurs in a positive literal of the body of $r$, and $E$ contains an arc $(B, A)$, with $lab((B,A)) =$"-", if there is a rule $r$ in $\mathcal{P}$ such that $q \in A$ occurs in the head of $r$ and $p \in B$ occurs in a negative literal of the body of $r$, and there is no arc $e'$ in $E$, with $lab(e') =$"+".

The Component Graph induces a partial ordering among the SCCs of the Dependency Graph as follows. For any pair of nodes $A, B$ of $G^c_{\mathcal{P}}$, $A$ *positively precedes* $B$ in $G^c_{\mathcal{P}}$ (denoted $A \prec_+ B$) if there is a *path* in $G^c_{\mathcal{P}}$ from $A$ to $B$ in which all arcs are labeled with "+"; $A$ *negatively precedes* $B$ (denoted $A \prec_- B$), if there is a path in $G^c_{\mathcal{P}}$ from $A$ to $B$ in which at least one arc is labeled with "-". This ordering induces *admissible component sequences* $C_1, \ldots, C_n$ of SCCs of $G_{\mathcal{P}}$ such that for each $i < j$, i) $C_j \not\prec_+ C_i$; ii) if $C_j \prec_- C_i$ then there is a cycle in $G^c_{\mathcal{P}}$ from $C_i$ to $C_j$(i.e. either $C_i \prec_+ C_j$ or $C_i \prec_- C_j$). Several such sequences exist in general.

*Example 5.* Given the program $\mathcal{P}$ of Example 4 its Component Graph is illustrated in Figure 1. It easy to see that $\{r\} \prec_+ \{p,t\}$, $\{r\} \prec_+ \{s\}$, while $\{p,t\} \prec_- \{r\}$ and $\{p,t\} \prec_- \{s\}$. An admissible component sequence would be $\{r\}$, $\{p,t\}$, $\{s\}$.

Intuitively, this ordering allows incremental grounding, one module at a time. If a module $A$ positively precedes a module $B$ then $A$ must be evaluated before $B$. If $A$ negatively precedes $B$ then $A$ should be possibly evaluated before $B$. Negative precedences are only overridden for unstratified components.

### 4.2 Instantiation Procedure

The procedure *Instantiate* shown in Figure 2 takes as input both a program $\mathcal{P}$ to be instantiated and the Component Graph $G^c_{\mathcal{P}}$, and outputs a set $\Pi$ of ground rules containing only atoms which can possibly be derived from $\mathcal{P}$, such that $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$. As already pointed out, the input program $\mathcal{P}$ is divided into modules corresponding to the nodes of $G^c_{\mathcal{P}}$(i.e. the SCCs of the Dependency Graph $G_{\mathcal{P}}$), and these modules are evaluated one at a time following an admissible component sequence.

The algorithm keeps a set of significant atoms $S$, a subset of the Herbrand Base, such that instantiated rules contain only atoms from $S$. Initially, $S = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, an admissible component sequence $(C_1, \ldots, C_n)$ is created, and each module corresponding to $C_i$ is grounded by invoking *InstantiateModule*. *Instantiate* runs until all components have been considered.

Procedure *InstantiateModule* handles the grounding of one module. Its inputs are the component $C$ to be grounded and $S$, and it computes those ground instances for each $r$ in the module for $C$ that contain only atoms from $S$. It also updates the set $S$ with the atoms occurring in the heads of rules in $\Pi$. The grounding of single rules is handled by the procedure *InstantiateRule*: Given the set of atoms that are known to be significant up to now, it builds all the significant ground instances of $r$, adds them to $\Pi$, and adds the head atoms of the newly generated ground rules to the significant ones. The evaluation of $r$ is essentially performed by evaluating the relational join of the positive body literals. Since the rule is safe, each variable occurring either in a negative literal or in the head of the rule appears also in some positive body literal, thus the instantiation of positive literals implies that all rule variables have been instantiated (details about this procedure can be found in [34]). Moreover, before being added to $\Pi$, each ground rule is simplified by removing from the body those (positive and negative) literals which are already known to be true. Those rules whose head is already true, or where a negative body literal is already known to be false are not added to $\Pi$. This mechanism will be described in detail in Section 4.3.

Note that a disjunctive rule $r$ may appear in the program modules of two different components. Thus, before processing $r$, *InstantiateRule* checks whether it has been already grounded during the instantiation of another component. This ensures that a rule is actually processed only within one program module.

Recursive rules are processed several times using a variant of the semi-naïve evaluation technique [39], in which at each iteration $n$ only the significant information derived during iteration $n-1$ is used. This is implemented by partitioning

**Procedure** *Instantiate*($\mathcal{P}$: Program; $G^c_\mathcal{P}$: ComponentGraph; **var** $\Pi$: GroundProgram)
  **var** $S$: SetOfAtoms, $(C_1, \ldots, C_n)$: List of nodes of $G^c_\mathcal{P}$;
  $S = EDB(\mathcal{P})$; $\Pi := \emptyset$;
  $(C_1, \ldots, C_n) :=$ *OrderedNodes*($G^c_\mathcal{P}$); /* *admissible component sequence* */
  **for** i = 1 ... n **do** *InstantiateModule*($\mathcal{P}, C_i, S, \Pi$);

**Procedure** *InstantiateModule* ($\mathcal{P}$: Program; $C$: SetOfPredicates;
  **var** $S$: SetOfAtoms; **var** $\Pi$: GroundProgram)
  **var** $\mathcal{N}S$: SetOfAtoms, $\Delta S$: SetOfAtoms;
  $\mathcal{N}S := \emptyset$ ; $\Delta S := \emptyset$;
  **for each** $r \in Exit(C, \mathcal{P})$ **do** *InstantiateRule*($r, S, \Delta S, \mathcal{N}S, \Pi$);
  **do**
    $\Delta S := \mathcal{N}S$; $\mathcal{N}S = \emptyset$;
    **for each** $r \in Recursive(C, \mathcal{P})$ **do** *InstantiateRule*($r, S, \Delta S, \mathcal{N}S, \Pi$);
    $S := S \cup \Delta S$;
  **while** $\mathcal{N}S \neq \emptyset$

**Procedure** *InstantiateRule*($r$: rule; $S$: SetOfAtoms; $\Delta S$: SetOfAtoms;
  **var** $\mathcal{N}S$: SetOfAtoms; **var** $\Pi$: GroundProgram)
/* *Given $S$ and $\Delta S$ builds the ground instances of $r$, simplifies them (see Sec. 4.3), adds them to $\Pi$, and add to $\mathcal{N}S$ the head atoms of the generated ground rules.* */

**Fig. 2.** The DLV Instantiation Procedure.

the significant atoms into three sets: $\Delta S$, $S$, and $\mathcal{N}S$. $\mathcal{N}S$ is filled with atoms computed during the current iteration (say $n$); $\Delta S$ contains atoms computed during the previous iteration (say $n-1$); and $S$ contains those computed earlier (up to iteration $n-2$). Initially, $\Delta S$ and $\mathcal{N}S$ are empty, and the exit rules contained in the program module of $C$ are evaluated by a single call to procedure *InstantiateRule*; then, the recursive rules are evaluated (do-while loop). At the beginning of each iteration, $\mathcal{N}S$ is assigned to $\Delta S$, i.e. the new information derived during iteration $n$ is considered as the significant information for iteration $n+1$. Then, *InstantiateRule* is invoked for each recursive rule $r$, and, at the end of each iteration, $\Delta S$ is added to $S$ (since it has already been dealt with). The procedure stops whenever no new information has been derived (i.e. $\mathcal{N}S = \emptyset$). Intuitively, the instantiation procedure of DLV allows for dynamically computing extensions of predicates; head atoms resulting from a rule instantiation immediately become members of the domains for the next iteration, even during the instantiation of the same recursive component.

Note that the algorithm described here does not take into account strong constraints, since they do not belong to any module of the input. Since their evaluation does not produce new significant atoms to be added to $S$, they are processed when the instantiation of all program modules has terminated, by means of a simplified version of procedure *InstantiateRule* for which sets $\Delta S$ and $\mathcal{N}S$ are useless. We remark that, as ground rules, also ground constraints are simplified, possibly resulting in constraints with empty body, and thus violated. In this case the computation is aborted since the input program is inconsistent.

It can be proved that the ground program generated by the DLV instantiation algorithm has the same answer sets as the non ground input program.

**Proposition 1.** *Let $\mathcal{P}$ be an ASP program, and $\Pi$ be the ground program generated by the algorithm* Instantiate. *Then $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$ (i.e. $\mathcal{P}$ and $\Pi \cup EDB(\mathcal{P})$ have the same answer sets).* □

The proof is a generalization of the one provided in [13] for function-free programs and is omitted for space reasons.

### 4.3 Instance Simplifications

Each ground rule generated by procedure *InstantiateRule* is examined and possibly simplified or even eliminated. In particular, body literals (positive and negative ones) which are already known to be true can be dropped. Moreover, since *InstantiateRule* computes variable substitutions by considering only positive body variables (sufficient because of safety), it may occur that some negative literal in the created rule instance is already known to be false. In this case the rule instance is already satisfied and need not be considered further.

For formalising these ideas, we partition the set $S$ of significant ground atoms into two subsets: $S^T$, containing those significant ground atoms that are already known to be true, and $S^{PT}$, containing those significant ground atoms that can become true ("potentially true"). $S^T$ is initialized with the input facts and then extended by those heads of instantiated rules which have been transformed into facts once the simplification below has been applied (essentially when all positive atoms are in $S^T$ and all negative literals are known to be true as well). The heads of all other instantiated rules belong to $S^{PT}$ (essentially all atoms from disjunctive heads and rules in which the body is not known to be true yet).

Once a rule instance $R$ is generated, the following actions are carried out for simplifying the program: i) if a positive literal $Q \in B^+(R)$ and $Q \in S^T$, then delete $Q$ from $B(R)$; ii) if a negative body literal not $Q$ over predicate $q$ is in $B(R)$, $Q \notin S$, and all the rules defining $q$ have been already instantiated, then delete not $Q$ from $B(R)$; iii) if a negative body literal not $Q$ is in $B(R)$ and $Q \in S^T$, then remove the ground instance $R$.

Note that, while for positive literals it suffices to check whether it is in $S^T$ for deciding its truth, for negative literals it is not sufficient to check whether its atom is not in $S$, but it should also not be added later. In case the input program is non-disjunctive and stratified,the modular evaluation (which respects the ordering previously described) together with the simplification above, allows the DLV grounder for completely evaluating the program.

To formalize this, we say that a literal with predicate $q$ is *solved* if (i) $q$ is defined solely by non-disjunctive rules (i.e., all rules with $q$ in the head are non-disjunctive), and (ii) $q$ does not depend (even transitively) on any unstratified predicate [3] or disjunctive predicate (i.e., a predicate defined by a disjunctive rule). The component ordering obtained by the Component Graph ensures that when a rule with a solved body predicate $q$ is instantiated, the rules defining $q$ have been instantiated previously; thus the extension of $q$ has already been

determined and, moreover, it completely belongs to the $S^T$ component of $S$. That is, the truth values of all ground literals that are instances of solved predicates are fully determined by the instantiator.

It follows that, after the simplification, none of the solved predicates occur in the rules of the ground instantiation $\Pi$ produced by the instantiator; rather, all the predicates occurring in the rules of $\Pi$ will be not solved and will be evaluated during the answer set search phase. In case the input program is non-disjunctive and stratified, all predicates are solved; thus all generated rule instances are either simplified to facts and added to $S^T$ or deleted. The program has a single answer set, coinciding with the set $S^T$.

### 4.4 Dealing with Weak Constraints and Aggregates

We now provide an overview of how the instantiation process handles the two major linguistic extensions, aggregates and weak constraints.

Concerning aggregates, in the following we suppose that programs respect aggregate stratification as defined in [14], which intuitively forbids recursion through aggregates. As previously described, the instantiation proceeds bottom-up following the dependencies among predicates. In the presence of aggregates, admissible component sequences must also conform with the aggregate stratification. Then, the instantiation of a rule with aggregates is performed by first evaluating non-aggregate body literals, and then applying variable substitutions to aggregates.

In more detail, let $r$ be the rule $H :\!\!- B, aggr.$, where $H$ is the head of the rule, $B$ is the conjunction of the non-aggregate body literals, and $aggr$ is an aggregate literal over a symbolic set $\{Vars{:}Conj\}$. A variable appearing in $r$ is said to be *local* if it appears solely in $aggr$, otherwise it is said to be *global*. The instantiation of $r$ proceeds by first evaluating the instantiation of the literals in $B$, thus computing a substitution $\theta$ for the global variables of $Conj$. Then, the (partially bound) conjunction $\theta(Conj)$ is instantiated by using the extensions of predicates appearing in $Conj$ (since the instantiation process respects also aggregate stratification, all extensions are already available). Thus, a set of pairs $\{\langle\theta_1(Vars) : \theta_1(\theta(Conj))\rangle,...,\langle\theta_n(Vars) : \theta_n(\theta(Conj))\rangle\}$is generated, where each $\theta_i$ is a possible substitution for the local variables in $\theta(Conj)$.

Note that, similar to rule simplification, we materialize only those pairs whose truth value cannot be determined yet (that is, instances of unsolved literals) and process the others dynamically, (partially) evaluating the aggregate already during instantiation. The same process is repeated for all further substitutions of the literals in $B$.

Instantiation of weak constraints, similar to that of strong contraints, is performed after the evaluation of all rules, basically computing the relational join of literals and simplifying the produced ground weak constraints as described in Section 4.3. Note that for a weak constraint also a weight and a level could be specified, each of them can either be an integer or a variable. In case of variables, the instantiation of the body literals also provides a substitution for them. Note

also that the body of a weak constraint could become empty after the simplification step, just like for strong constraints. This means that the weak constraint is unconditionally violated, but differently from strong constraints this does not cause the program to be inconsistent, but only causes the penalty of this weak constraint to be present for each answer set. Therefore the violated weak constraint is stored in a dedicated structure in order to be treated later on when computing the costs of each answer set.

### 4.5 Finitely Ground Programs

The presence of recursive function symbols within DLV programs has a strong impact on the grounding process, which might even not terminate. All common reasoning tasks on such programs are indeed undecidable, in the general case. Despite this, the DLV instantiation procedure does allow for dealing with recursive function symbols, and it is guaranteed to terminate on the class of *finitely-ground ($\mathcal{FG}$)* programs defined in [5]. Intuitively, for each program $\mathcal{P}$ in this class, there exists a finite ground program $\mathcal{P}'$ having exactly the same answer sets as $\mathcal{P}$. $\mathcal{P}'$ is computable for $\mathcal{FG}$ programs, thus answer sets of $\mathcal{P}$ are computable as well. Moreover, each computable function can be expressed by a $\mathcal{FG}$ program. Since $\mathcal{FG}$ programs express any computable function, membership in this class is obviously not decidable, but it has been proved to be semi-decidable [5].

$\mathcal{FG}$ programs are defined by exploiting the fix-point $\Phi^\infty$ of an operator $\Phi$ [2] that acts on a module of a program $\mathcal{P}$ in order to: (i) select only those ground rules whose positive body is contained in a set of ground atoms consisting of the heads of a given set of rules; (ii) perform further simplifications by deleting all those rules whose body is certainly false or whose head is certainly already true w.r.t. a given set of ground atoms $A$, and simplifies the remaining rules by removing from the bodies all literals that are true w.r.t. $A$. The proper composition of consecutive applications of $\Phi^\infty$ to all program modules according to one admissible component sequence $\gamma$ (that is, an ordering which respects dependencies induced by the Component Graph) produces an instantiation $I_\gamma(\mathcal{P})$ of $\mathcal{P}$ which drops many useless rules w.r.t. answer sets computation. The program $\mathcal{P}$ is finitely-ground if $I_\gamma(\mathcal{P})$ is finite for every admissible component ordering $\gamma$.

The way in which the instantiation $I_\gamma(\mathcal{P})$ is computed has a number of relevant similarities with the DLV instantiation approach. First of all, the application of the operator $\Phi$ is performed by considering the components of the Dependency Graph one at a time and following one of the orderings induced by the Component Graph, exactly as in the case of the bottom-up evaluation performed by the DLV instantiator. Moreover, for every component $C$, the ground rules produced by the application of $\Phi$ only contain ground atoms appearing in the heads of ground rules produced by the evaluation of the previous modules. In the DLV approach this corresponds to the use of the set $S$ of ground atoms significant for the instantiation (see the algorithm in Figure 2). The operator $\Phi$ also

[2] For details we refer the reader to [5].

performs a simplification on the produced ground rules (which possibly become new facts) by taking into account facts and ground rules previously determined, similarly to the simplification performed by DLV described in Section 4.3. This gives the intuition that the ground program produced by DLV coincides with the instantiation $I_\gamma(\mathcal{P})$ if $\gamma$ is the ordering exploited by DLV. However, in some cases, the DLV instantiation can be actually smaller than $I_\gamma(\mathcal{P})$. Indeed, in case of components with recursive rules, the computation of $\Phi^\infty$ simulates the semi-naive approach of DLV, in which head atoms resulting from a rule instantiation immediately become members of the domains for the next iteration; but the simplification step applied by $\Phi$ only considers information coming from previous components, while the DLV simplification also considers information derived during the evaluation of the current component, in previous iterations. Summarizing, the ground program $\Pi$ generated by the algorithm *Instantiate*, according to a component ordering $\gamma$, is not bigger than $I_\gamma(P)$. Hence, if $I_\gamma(P)$ is finite, $\Pi$ is finite as well, thus proving the following result.

**Proposition 2.** *Let $\mathcal{P}$ be a* DLV *program, and $\Pi$ be the ground program generated by the algorithm* Instantiate*. Then, if $\mathcal{P}$ is a $\mathcal{FG}$ program, $\Pi$ is finite.* □

Note that for applications in which termination needs to be guaranteed a priori, the DLV grounder has been endowed with a checker which allows the user to statically recognize if the input program belongs to a class for which the grounding process terminates, the class of argument-restricted programs [28]. However, if the user is confident that the program can be grounded in finite time, even if it does not belong to the class of argument-restricted programs, then she can disable the checker by specifying a command-line option. Moreover, the DLV grounder provides another way for guaranteeing termination: the possibility to specify, by means of another command-line option, the maximum allowed nesting level for functional terms.

## 5 Optimization Techniques

Much effort has been spent on sophisticated algorithms and optimization techniques aimed at improving the performance of the DLV instantiator. In the following we briefly recall the most relevant ones, providing references to detailed descriptions of the respective techniques.

Some of the techniques exploited for optimizing the instantiation procedure descend from the database field. For instance, the DLV instantiator implements a **program rewriting** [12] strategy descending from query optimization techniques in relational algebra. According to this technique, program rules are automatically rewritten by pushing projections and selections down the execution tree as much as possible; this allows for reducing in many cases the size of the program instantiation. Another rewriting-based optimization technique used in DLV are **dynamic magic sets** [1], an extension of the Magic Sets technique originally defined for standard Datalog for optimizing query answering over logic programs. The Magic Sets technique rewrites the input program for identifying

a subset of the program instantiation which is sufficient for answering the query. The restriction of the instantiation is obtained by means of additional "magic" predicates, whose extensions represent relevant atoms w.r.t. the query. Dynamic Magic Sets, specifically conceived for disjunctive programs, inherit the benefits provided by standard magic sets and additionally allow for exploiting the information provided by the magic predicates also during the nondeterministic answer set search.

Another group of techniques descending from databases concerns the instantiation process of each rule of the program. In particular, since rule instantiation is essentially performed by evaluating the relational join of the positive body literals, an optimal ordering of literals in the body is a key issue for the efficiency of the instantiation procedure, just like for join computation. Indeed, a good ordering may dramatically affect the overall instantiation time. The DLV instantiator exploits a well-motivated **body reordering** criterion [26], which determines the position in the body of each literal by taking into account two factors: one is a measure of how much the choice of a literal $L$ reduces the search space for possible substitutions and the other takes into account the binding of the variables of $L$ (since preferring literals with already bound variables, possible inconsistencies may be detected quickly).

Moreover, to guarantee good performance also in case of problems with huge amount of input data, the DLV instantiator exploits an efficient main-memory indexing technique [8]. In particular, it implements a kind of **on demand indexing**, where a generic argument can be indexed (not necessarily the first one), and indices are computed during the evaluation and only if they can really be exploited. Moreover, the argument to be indexed is not predetermined, but is established during the computation according to a heuristic. For optimizing the rule instantiation task, a **backjumping algorithm** [34] is employed. In particular, given a rule $r$ to be grounded, this algorithm exploits both the semantical and the structural information about $r$ for computing efficiently the ground instances of $r$, avoiding the generation of "useless" rules. That is, from each rule only a relevant subset of its ground instances are computed, avoiding the generation of "useless" instances, but fully preserving the semantic of the program.

In the last few years,in order to make use of modern multi-core/multi-processor computers, a parallel version of the DLV instantiator has been realized, based on a number of strategies [7, 33] specifically conceived for the instantiation task. More in detail, the **parallel instantiator** is based on three levels of parallelism: *components*, *rules* and *single rule* level. The first level allows for instantiating in parallel subprograms of the program in input: it is especially useful when handling programs containing parts that are, somehow, independent. The second one allows for the parallel evaluation of rules within a given subprogram: it is useful when the number of rules in the subprograms is large. The third one allows for the parallel evaluation of a single rule: it is crucial for the parallelization of programs with few rules, where the first two levels are almost not applicable. Moreover, the parallel instantiator is endowed with mechanisms for dealing with two important issues that may strongly affect the performance of a real

implementation: *load balancing* and *granularity control*. Indeed, if the workload is not uniformly distributed to the available processors then the benefits of parallelization are not fully obtained; moreover, if the amount of work assigned to each parallel processing unit is too small then the (unavoidable) overheads due to creation and scheduling of parallel tasks might overcome the advantages of parallel evaluation.

## 6 Related Work

In this section we briefly discuss some of the main differences and similarities of the DLV Intelligent Grounder with respect to the other two most popular ASP instantiators, namely, lparse [31], and gringo [17].

Concerning lparse, it accepts a different class of input programs, and follows different strategies for the computation. Indeed, lparse accepts logic programs respecting *domain restrictions*. This condition enforces each variable in a rule to occur in a positive body literal, called domain literal, which (i) is not mutually recursive with the head, and (ii) is neither unstratified nor dependent (also transitively) on an unstratified literal. For instance, the program consisting of rules $\mathcal{P} = \{a(X) :\!\text{-}\, b(X), c(X). \quad b(X) :\!\text{-}\, a(X).\}$ is not accepted by lparse. To instantiate a rule $r$, lparse employs a nested loop that scans the extensions of the domain predicates occurring in the body of $r$, and generates ground instances accordingly. It is therefore a comparatively simple and fast instantiation method, at least for applications with few domains or for domains with small extensions. However, lparse may generate useless rules as they may contain non-domain body literals that are not derivable by the program. The DLV instantiator incorporates several database optimization techniques and builds the domains dynamically, hence the instantiation generated by DLV is generally a subset of that generated by lparse. Thus, in case of applications where the size of domain extensions are very large (as in many industrial applications), lparse may take significantly more time and produce a larger instantiation than DLV.

Concerning gringo, versions up to 3.0 also accepted only domain restricted programs; however, the notion of domain literal was an extension of that of lparse, so gringo could handle all the programs accepted by lparse but not vice versa. For example, program $\mathcal{P}$above was accepted by gringo, while the following one, encoding reachability, could not be handled prior to version 3.0: $\{r(X,Y) :\!\text{-}\, arc(X,Y). \quad r(X,Y) :\!\text{-}\, arc(X,U), r(U,Y).\}$. The current gringo releases (since version 3.0) removed domain restrictions and instead requirespro-grams to be safe as in DLV, and evaluate them according to a grounding algorithm based on the semi-naive schema, very similar to the one in the DLV instantiator. It is worth noting that, passing from domain restrictedness to the more general notion of safety, also the gringo grounding process may not terminate, just like for DLV. However, while gringo leaves the responsibility to check whether the input program has a finite grounding to the user, DLV implements some checks for guaranteeing termination (see Section 4.5), and the user can choose to disable them.

## References


1. Alviano, M., Faber, W.: Dynamic Magic Sets and super-coherent answer set programs. AI Communications 24(2), 125–145 (2011)
2. Anger, C., Konczak, K., Linke, T.: `NoMoRe`: A System for Non-Monotonic Reasoning. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR'01. LNAI, vol. 2173, pp. 406–410. (Sep 2001)
3. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Washington DC (1988)
4. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. AMAI 50(3–4), 333–361 (2007)
5. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: ICLP 2008. vol. 5366, pp. 407–424. (Dec 2008)
6. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M., Sirianni, M., Terracina, G., Veltri, P.: The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In: LPNMR 2011, pp. 388–403 (2011)
7. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. J. Of Algorithms 63(1–3), 34–54 (2008)
8. Catalano, G., Leone, N., Perri, S.: On demand indexing techniques for the dlv instantiator. In: Proceedings of the Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'08). Udine, Italy (2008)
9. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys 33(3), 374–425 (2001)
10. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. LPNMR 2009. LNCS 5753, pp. 637–654 (2009)
11. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (Sep 1997)
12. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In: INAP Organizing Committee (ed.) DDLP'99. pp. 135–139. Prolog Association of Japan (Sep 1999)
13. Faber, W., Leone, N., Perri, S., Pfeifer, G.: Efficient Instantiation of Disjunctive Databases. Tech. Rep. DBAI-TR-2001-44, TU Wien, Austria (Nov 2001), online at http://www.dbai.tuwien.ac.at/local/reports/dbai-tr-2001-44.pdf
14. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the dlv system. TPLP 8(5–6), 545–580 (2008)
15. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007. pp. 386–392.(Jan 2007)
16. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR'07. LNAI 4483, pp. 3–17. (2007)
17. Gebser, M., Schaub, T., Thiele, S.: Gringo : A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR'07. LNAI 4483, pp 266–271. (2007)
18. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: ICLP/SLP 1988. pp. 1070–1080. MIT Press, Cambridge, Mass. (1988)

19. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
20. Grasso, G., Iiritano, S., Leone, N., Lio, V., Ricca, F., Scalise, F.: An asp-based system for team-building in the gioia-tauro seaport. In: PADL 2010. LNCS 5937, pp. 40–42. (2010)
21. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV Applications for Knowledge Management. In: Erdem, E., Lin, F., Schaub, T. (eds.). LPNMR 2009. LNCS 5753, pp. 591–597. (2009)
22. Ielpa, S.M., Iiritano, S., Leone, N., Ricca, F.: An ASP-Based System for e-Tourism. In: Erdem, E., Lin, F., Schaub, T. (eds.). LPNMR 2009. LNCS 5753, pp. 368–381. (2009)
23. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM TOCL 7(1), 1–37 (Jan 2006)
24. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005. pp. 915–917. ACM Press (Jun 2005)
25. Leone, N., Lio, V., Terracina, G.: $DLV^{DB}$: Adding Efficient Data Management Features to ASP. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR-7. LNAI 2923, pp. 341–345. (Jan 2004)
26. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR'01. LNAI 2173, pp. 280–294. (Sep 2001)
27. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (Jul 2006)
28. Lierler, Y., Lifschitz, V.: One More Decidable Class of Finitely Ground Programs. In: ICLP 2009. LNCS 5649, pp. 489–493. (Jul 2009)
29. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR-7. LNAI, vol. 2923, pp. 346–350. (Jan 2004)
30. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. AI 157(1–2), 115–137 (2004)
31. Syrjänen T (2002) Lparse 1.0 User's Manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
32. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: Apt, K.R., Marek, V.W., Truszczyński, M., Warren, D.S. (eds.) The Logic Programming Paradigm – A 25-Year Perspective, pp. 375–398. (1999)
33. Perri, S., Ricca, F., Sirianni, M.: Parallel instantiation of ASP programs: techniques and experiments. TPLP (2012)
34. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. AMAI 51(2–4), 195–228 (2007)
35. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based system for enterprise ontologies. Journal of Logic and Computation (2009)
36. Ruffolo, M., Manna, M.: HiLeX: A System for Semantic Information Extraction from Web Documents. ICEIS (Selected Papers). Lecture Notes in Business Information Processing, vol. 3, pp. 194–209 (2008)
37. Rullo, P., Cumbo, C., Policicchio, V.L.: Learning rules with negation for text categorization. ACM Symposium on Applied Computing. pp. 409–416. ACM (2007)

38. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP 8, 129–165 (2008)
39. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)