The Disjunctive Datalog System DLV*

Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri,
Gerald Pfeifer, and Giorgio Terracina

Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
{alviano,faber,leone,perri,terracina}@mat.unical.it, gerald@pfeifer.com

Abstract. DLV is one of the most successful and widely used answer set programming (ASP) systems. It supports a powerful language extending Disjunctive Datalog with many expressive constructs, including aggregates, strong and weak constraints, functions, lists, and sets. The system provides database connectivity offering a simple way for powerful reasoning on top of relational databases. In this paper, we provide an ample overview of the DLV system. We illustrate its input language and give indications on how to use it for representing knowledge. We also provide a panorama on the system architecture and the main optimizations it incorporates. We then focus on DLV^{DB}, an extension of the basic system which allows for tight coupling with traditional database systems. Finally, we report on a number industrial applications which rely on DLV.

1 Introduction

In this paper, we provide an overview of the disjunctive datalog system DLV [26]. The DLV project has been active for more than fourteen years, and has led to the development and continuous enhancement of the DLV system. DLV is widely used by researchers all over the world, and has stimulated quite some interest also in industry (see Section 6).

The key reasons for the success of DLV can be summarized as follows:
Advanced knowledge modeling capabilities. DLV provides support for declarative problem solving in several respects:

- High expressiveness of its knowledge representation language, extending disjunctive datalog with many expressive constructs, including aggregates [17], weak constraints [4], functions, lists, and sets [7]. These constructs not only increase the expressiveness of the language; they also improve its knowledge modeling power, enhancing DLV's usability in real-world contexts.
- Full declarativeness: ordering of rules and subgoals is immaterial, the computation is sound and complete, and its termination is guaranteed.
- Front-ends for dealing with specific applications.

* This research has been partly supported by Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l., and by MIUR under the PRIN project LoDeN.

Solid Implementation. Much effort has been spent on sophisticated algorithms and techniques for improving the performance (see Sections 4.1 and 5), including

- Database optimization techniques: magic sets [10, 1], indexing and join ordering methods [25].
- Search optimization techniques: heuristics [18, 28], backjumping techniques [34, 31], pruning operators [8, 16].
- Parallel evaluation [9, 30].
- Evaluation in mass-memory [40].

Interoperability. A number of mechanisms have been implemented to allow DLV to interact with external systems:

- Interoperability with relational DBMSs: ODBC interface and DLV^{DB} [40].
- Interoperability with Semantic Web reasoners: DLVHEX [14].
- Calling external (C++) functions from DLV programs: DLVEX [6].
- Calling DLV from Java programs: Java Wrapper [33].

In the following, we introduce the language constructs of DLV by examples, provide some use-cases of how knowledge can be represented in the DLV language. Subsequently, we provide an overview of the architecture and techniques of DLV, and we then focus on DLV ^{DB} – DLV version working (mostly) in mass-memory. Finally, we provide information on industrial products that rely on DLV.

2 The language of DLV

In this section, we describe the language of the DLV system by examples, providing the intuitive meaning of the main constructs. For further details and the formal definition, we refer to [26, 7, 17]. We first introduce the basic language, which is based on the founding work by Gelfond and Lifschitz [20] and then we illustrate a number of extensions including aggregates [17], weak constraints [4], complex terms [7], queries and database interoperability constructs [40].

Basic Language. The main construct in the DLV language is a rule, an expression of the form $Head$:- $Body$., where $Body$ is a conjunction of literals and $Head$ is a disjunction of atoms. Informally, a rule can be read as follows: "if $Body$ is true then $Head$ is true". A rule without a body is called a *fact*, since it models an unconditional truth (for simplicity :- is omitted); whereas a rule with an empty head, called *strong constraint*, is used to model a condition that must be false in any possible solution. A set of rules is called *program*. The semantics of a program is given by its *answer sets* [20]. A program can be used to model a problem to be solved: the problem's solutions correspond to the answer sets of the program (which are computed by DLV). Therefore, a program may have no answer set (if the problem has no solution), one (if the problem has a unique solution) or several (if the problem has more than one possible solutions).

As an example consider the problem of automatically creating an assessment test from a given database of questions where each question is identified by a unique string, covers a particular topic, and requires an estimated time to be answered. The input data about questions can be represented by means of a set of facts of type $question(q, topic, time)$; in addition, facts of the form $relatedTopic(topic)$ specify the topics related to the subject of the test.

For instance, consider the case in which only four questions are given, represented by facts: $question(q1, computerscience, 8)$, $question(q2, computerscience, 15)$, $question(q3, mathematics, 15)$, and $question(q4, mathematics, 25)$. Moreover, suppose that computer science is the only topic to be covered by the test, therefore $relatedTopic(computerscience)$ is also part of the input facts. The program consisting only of these facts has one answer set A_1 containing exactly the five facts.

Assessment creation amounts to selecting a set of questions from the database, according to a given specification. To single out questions related to the subject of the test, one can write the rule:

$$relatedQuestion(Q) :- question(Q, Topic, Time), relatedTopic(Topic).$$

that can be read: "Q is a question related to the test if Q has a topic related to some of the subjects that have to be assessed". Adding this rule to the input facts reported earlier yields one answer set $A_2 = A_1 \cup \{relatedQuestion(q1), relatedQuestion(q2)\}$.

For determining all the possible subsets of related questions the following disjunctive rule can be used:

$$inTest(Q) \vee discard(Q) :- relatedQuestion(Q).$$

Intuitively, this rule can be read as: "if Q identifies a related question, then either Q is taken in the test or Q is discarded." This rule has the effect of associating each possible choice of related questions with an answer set of the program. Indeed, the answer sets of the program $\mathcal{P}$ consisting of the above two rules and the input facts are:

$$A_3 = A_2 \cup \{discard(q1), discard(q2)\}, \quad A_4 = A_2 \cup \{inTest(q1), discard(q2)\},$$
$$A_5 = A_2 \cup \{discard(q1), inTest(q2)\}, \quad A_5 = A_2 \cup \{inTest(q1), inTest(q2)\}$$

corresponding to the four possible choices of questions $\{\}, \{q1\}, \{q2\}, \{q1, q2\}$. Note that the answer sets are minimal with respect to subset inclusion. Indeed, for each question Q there is no answer set in which both $inTest(Q)$ and $discard(Q)$ appear.

At this point, some strong constraints can be used to single out some solutions respecting a number of specification requirements. For instance, suppose we are interested in tests containing only questions requiring less than 10 minutes to be answered. The following constraint models this requirement:

$$:- inTest(Q), question(Q, Topic, Time), Time >= 10.$$

The program obtained by adding this constraint to $\mathcal{P}$ has only two answer sets A_3 and A_4.

Aggregate Functions. More involved properties requiring operations on sets of values can be expressed by aggregates, a DLV construct similar to aggregation in the SQL language. DLV supports five aggregate functions, namely $\#sum$, $\#count$, $\#times$, $\#max$, $\#min$.

In our running example we might want to restrict the included questions to be solvable in an estimated time of less than 60 minutes. This can be achieved by the following strong constraint:

$$:\!-\ \text{not}\#sum\{Time, Q : inTest(Q), question(Q, Topic, Time)\} < 60.$$

The aggregate sums up the estimated solution times of all questions in the test, and the constraint will eliminate all scenarios in which this sum is not less than 60.

Optimization Constructs. The DLV language also allows for specifying optimization problems (i.e. problems where some goal function must be minimized or maximized). This can be achieved by using *weak constraints*. From a syntactic point of view, a weak constraint is like a strong one where the implication symbol $:\!-$ is replaced by $:\sim$. Contrary to strong constraints, 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 "try to falsify B", or "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). The answer sets minimize the sum of weights of the violated (unsatisfied) 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.

As an example, if we want to prefer quick-to-answer questions in tests, the following weak constraint represent this desideratum.

$$:\sim inTest(Q), question(Q, Topic, Time).\ [Time : 1]$$

Intuitively, each question in the test increases the total weight of the solution by its estimated solution time. Thus solutions where the total weight is minimal are preferred.

Complex Terms. The DLV language allows for the use of complex terms. In particular, it supports function symbols, lists, and sets. Prolog-like syntax is allowed for both function symbols and lists, while sets are explicitly represented by listing the elements in brackets.

As an example, we enrich the question database for allowing two types of questions, open and multiple choice. Input questions are now represented by facts like the following

$$question(q1, math, open(text), 10).$$
$$question(q2, physics, multiplechoice(text, \{c1, c2, c3\}, \{w1, w2, w3\}), 3).$$

where function symbols *open* and *multiplechoice* are used for representing the two different types of questions. In particular, *open* is a unary function whose

only parameter represents the text of the question, while *multiplechoice* has three parameters, the text of the question, a set containing correct answers and another set of wrong answers.

The use of sets allows for modeling multi-valued attributes, while function symbols can be used for modeling "semi-structured" information.

Handling complex terms is facilitated by a number of built-in predicates. For instance, the following rule uses the *#member* built-in for selecting correct answers given by a student in the test.

$$correctAnswer(Student, QID, Ans) :- inTest(QID), answer(Student, QID, Ans), \\ question(QID, To, multiplechoice(Tx, Cs, Ws), Ti), \#member(Ans, Cs).$$

Queries. The DLV language offers the possibility to express conjunctive queries. From a syntactic point of view, a query in DLV is a conjunction of literals followed by a question mark. Since a DLV program may have more than one answer set, there are two different reasoning modes, brave and cautious, to compute a query answer. In the brave (resp. cautious) mode, a query answer is true if the corresponding conjunction is true in some (resp. all) answer sets.

For instance, the answers to the following simple query are the questions having as topic *computerscience* that are contained in some (resp. all) answer sets of the program when brave (resp.cautious) reasoning is used.

$$inTest(Q), question(Q, computerscience, T)?$$

Database Interoperability. The DLV system supports interoperability with databases by means of *#import*/*#export* commands for importing and exporting relations from/to a DBMS. The *#import* command reads tuples from a specified table of a relational database and stores them as facts with a predicate name provided by the user.

In our example, questions can be retrieved from a database by specifying in the program the following directive.

$$\#import(questionDB, \text{“}user\text{”}, \text{“}passwd\text{”}, \text{“}SELECT * FROM\ question\text{”}, question).$$

where *questionDB* is the name of the database, "*user*" and "*passwd*" are the data for the user authentication, "$SELECT * FROM\ question$" is an SQL query that constructs the table that will be imported and *question* is the predicate name which will be used for constructing the new facts.

In a similar way the *#export* command allows for exporting the extension of a predicate in an answer set to a database.

3 Knowledge Representation

In this section, we illustrate the usage of DLV as a tool for knowledge representation and reasoning. We consider a number of problems, from classical deductive database applications to search and optimization problems, and show how the language of DLV can be used to encode them in a highly declarative fashion.

3.1 Deductive Databases

We next present two problems motivated by classical deductive database applications, namely *Same Generation* and *Simple Paths*. For the first one, we provide an encoding consisting of positive datalog rules, while we encode the second one by using complex terms (lists).

Same Generation. Given a parent-child relationship (an acyclic directed graph), we want to find all pairs of persons belonging to the same generation. Two persons are of the same generation, if either (i) they are siblings, or (ii) they are children of two persons of the same generation.

Suppose that the input is provided by facts like $parent(thomas, joseph)$ stating that *thomas* is a parent of *joseph*. Then, this problem can be encoded by the following program, which computes a relation $samegeneration(X, Y)$ containing all facts such that X is of the same generation as Y:

$$\begin{aligned} &samegeneration(X, Y) \text{ :- } parent(P, X), parent(P, Y).\\ &samegeneration(X, Y) \text{ :- } parent(P1, X), parent(P2, Y),\\ &\qquad\qquad\qquad\qquad samegeneration(P1, P2). \end{aligned}$$

Simple Paths. Given a directed graph, a *simple path* is a sequence of nodes, each one appearing exactly once, such that from each one (but the last) there is an edge to the next in the sequence.

The following program exploits complex terms for deriving all simple paths for a directed graph, starting from a given *edge* relation:

$$\begin{aligned} &path([X, Y]) \text{ :- } edge(X, Y).\\ &path([X|[Y|W]]) \text{ :- } edge(X, Y),\ path([Y|W]),\ \text{not } \#member(X, [Y|W]). \end{aligned}$$

The first rule builds a simple path as a list of two nodes directly connected by an edge. The second rule constructs a new path adding an element to the list representing an existing path. The new element will be added only if there is an edge connecting it to the head of an already existing path. The built-in predicate $\#member$ allows to avoid the insertion of an element that is already included in the list; without this check, the construction would never terminate in the presence of circular paths (note that, by default, DLV disallow programs which might not terminate [7]).

3.2 Search Problems

Here we illustrate two different usages of the DLV language for solving search problems. On the one hand we consider the *Seating problem* for showing how a search problem can be encoded in a DLV program whose answer sets correspond to the problem solutions. On the other hand, we consider a problem of number and graph theory, namely *Ramsey Numbers*, for showing how to build a DLV program whose answer sets witness that a property does not hold, i.e., the property at hand holds if and only if the DLV program has no answer set.

Seating. Consider the problem of generating a seating arrangement for k guests, with m tables and n chairs per table. Guests who like each other should sit at the same table; guests who dislike each other should sit at different tables.

Suppose that the number of chairs per table is specified by $nChairs(X)$ and that $person(P)$ and $table(T)$ represent the guests and the available tables, respectively. Then, we can generate a seating arrangement by the following program:

$$
\begin{aligned}
&at(P,T) \vee not\_at(P,T) \text{ :- } person(P), table(T).\\
&\text{ :- } table(T), nChairs(C), \text{not } \#count\{P : at(P,T)\} \leq C.\\
&\text{ :- } person(P), \text{not } \#count\{T : at(P,T)\} = 1.\\
&\text{ :- } like(P1,P2), at(P1,T), \text{not } at(P2,T).\\
&\text{ :- } dislike(P1,P2), at(P1,T), at(P2,T).
\end{aligned}
$$

The disjunctive rule guesses whether person P sits at table T or not, thus generating all possible assignments of persons to tables (even those where a person is not assigned to any table or it is assigned to more than one). The strong constraints discard assignments that do not respect the problem specification. In particular the first constraint, for each table T, counts the number of persons assigned to T and ensures that it does not exceed the number of chairs per table, whereas the second one, imposes that each person is seated at precisely one table. Finally, the last two constraints ensure that persons who like each other are seated at the same table and persons who dislike each other are not.

Ramsey Numbers The Ramsey number $R(k,m)$ is the least integer n such that, no matter how we color the arcs of the complete undirected graph (clique) with n nodes using two colors, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique).

Ramsey numbers exist for all pairs of positive integers k and m [32]. We next show a program $\mathcal{P}$ that allows for deciding whether a given integer n is <u>not</u> the Ramsey Number $R(3,4)$. By varying the input number n, we can determine $R(3,4)$, as described below. Let $\mathcal{F}$ be the collection of facts for input predicate arc encoding a complete graph with n nodes. $\mathcal{P}$ is the following program:

$$
\begin{aligned}
&blue(X,Y) \vee red(X,Y) \text{ :- } arc(X,Y).\\
&\text{ :- } red(X,Y),\ red(X,Z),\ red(Y,Z).\\
&\text{ :- } blue(X,Y),\ blue(X,Z),\ blue(Y,Z),\\
&\quad blue(X,W),\ blue(Y,W),\ blue(Z,W).
\end{aligned}
$$

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red clique (i.e., a complete graph) with 3 nodes, and the second constraint eliminates the colorings containing a blue clique with 4 nodes. The program $\mathcal{P} \cup \mathcal{F}$ has an answer set if and only if there is a coloring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular n, then n is <u>not</u> $R(3,4)$, that is, $n < R(3,4)$. On the other hand, if $\mathcal{P} \cup \mathcal{F}$ has no answer set, then $n \geq R(3,4)$. Thus, the smallest n such that no answer set is found is the Ramsey number $R(3,4)$.

3.3 Optimization Problems

In this section, we present two optimization problems, the first one is a classical graph theory problem, while the second one concerns exam scheduling.

Maximal Cut. Given a graph $G = (V, E)$ we want to compute the maximal cuts of the graph, i.e. a partition of V in two sets V_1 and V_2 such that the number of edges of G having one endpoint in V_1 and one endpoint in V_2 is maximal.

Suppose that the graph G is specified by facts over predicates *node* and *edge*. Then, the following program compute the maximal cuts of G:

$$
\begin{array}{l}
v1(X) \vee v2(X) :\!- node(X). \\
:\!\sim v1(X), v2(Y), \mathrm{not}\, edge(X, Y). \ [1:1] \\
:\!\sim v2(X), v1(Y), \mathrm{not}\, edge(X, Y). \ [1:1]
\end{array}
$$

Here the disjunctive rule guesses whether $node(X)$ is in the subset V_1 or V_2, thus generating all the possible partitions of nodes into subsets. Then, the two weak constraints allow for preferring partitions where the number of edges with both nodes assigned to the same subset is minimum.

Exam Scheduling. Here we have to schedule the exams for several university courses in three time slots t_1, t_2, and t_3 at the end of the semester. In other words, each course should be assigned exactly to one of these three time slots. Specific instances I of this problem are provided by sets $\mathcal{F}_I$ of facts specifying the exams to be scheduled. An example fact is $exam(cs1, lee, cs, 1)$ specifying the exam identified as $cs1$, taken by *lee*, of the first year of the curriculum *cs*.

Several exams can be assigned to the same time slot (the number of available rooms is sufficiently high), but the scheduling has to respect the following specifications:

$S1$ Two exams given by the same professor cannot run in parallel, i.e., in the same time slot.

$S2$ Exams of the same curriculum should be assigned to different time slots, if possible. If $S2$ is unsatisfiable for a curriculum C, one should:

($S2_1$) first of all, minimize the overlap between exams of the same year of C,

($S2_2$) then, minimize the overlap between exams of different years of C.

This problem can be encoded in the DLV language by the following program $\mathcal{P}$:

$$
\begin{array}{l}
at(Id, t_1) \vee at(Id, t_2) \vee at(Id, t_3) :\!- exam(Id, P, C, Y). \\
:\!- at(Id, T), at(Id', T), Id <> Id', exam(Id, P, C, Y), exam(Id', P, C', Y'). \\
:\!\sim at(Id, T), at(Id', T), exam(Id, P, C, Y), exam(Id', P', C, Y), Id <> Id'. \ [1:2] \\
:\!\sim at(Id, T), at(Id', T), exam(Id, P, C, Y), exam(Id', P', C, Y'), Y <> Y'. \ [1:1]
\end{array}
$$

The disjunctive rule generates the possible assignments of exams to time slots and the strong constraint discards the assignments of the same time slot to two exams of the same professor, as required by the specification $S1$. Finally, the two weak constraints state that exams of the same curriculum should *possibly*

not be assigned to the same time slot. However, the first one, which has higher priority (level 2), states this desire for the exams of the curriculum of the same year, while the second one, which has lower priority (level 1) states it for the exams of the curriculum of different years.

4 DLV Implementation

A main strength of DLV is its implementation which is based on solid theoretical foundations, and relies on sophisticated data structures and advanced optimization techniques. In this section we first outline the main aspects of the DLV computation, then we give an overview of the main techniques which were employed in the implementation. Finally, we describe the general architecture of the system.

The computation of the 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 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. The size of the instantiation is a critical aspect for the efficiency of the system: On the one hand, instantiated programs can require exponential space, on the other hand, the answer set search can take exponential time in the size of the grounded program. Therefore even a small reduction in the size of the generated instantiation can yield significant performance gains.

Answer set search is then performed by the *Model Generator (MG)* and the *Model Checker (MC)* on the program produced by the grounding. Roughly, the MG produces "candidate" answer sets, the stability of which is subsequently verified by the MC. MG is the non-deterministic core of the system, and it is implemented as a backtracking search similar to the Davis-Putnam-Logemann-Loveland (DPLL) procedure [13] for SAT solving. Basically, starting from the empty (partial) interpretation, the Model Generator repeatedly assumes truth-values for atoms (branching step), subsequently computing their deterministic consequences (propagation step). This is done until either an answer set candidate is found or an inconsistency is detected. Candidate answer sets are then checked by the *Model Checker* module; whereas, if an inconsistency is detected, chosen literals have to be undone. For disjunctive programs, model checking is as hard as the problem solved by the Model Generator, while it is trivial for non-disjunctive programs.

4.1 Main Optimization Techniques

Many optimization techniques have been incorporated into the DLV engine, including database techniques for efficient instantiation, advanced pruning operators, look-ahead and look-back techniques for model generation, and innovative

techniques for answer-set checking. In the following, we recall the most relevant ones which have been adopted in the main phases of the evaluation.

Instantiation Phase. DLV implements several relevant optimization techniques for the instantiation, mainly descending from the databases field, aimed at reducing both the size of the instantiation and the time needed for generating it. For instance, the DLV instantiator implements a *Program Rewriting* [15] strategy descending from query optimization techniques in relational algebra which allows for reducing in many cases the size of the program instantiation. According to this technique, program rules are automatically rewritten by pushing projections and selections down the execution tree as much as possible. Another rewriting-based optimization technique used in DLV are *Dynamic Magic Sets* [10, 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 non-deterministic answer set search.

Another group of techniques concerns the instantiation process of each rule of the program. In particular, since computing all the possible instantiations of a rule is, basically, analogous to computing all the answers of a conjunctive query joining the extensions of literals of the rule body, DLV uses a *Join Ordering* [25] strategy for determining an efficient evaluation order of the literals in the rule and a main-memory *On-demand Indexing* technique, where a generic argument can be indexed (established according to a heuristic), indices are computed on demand during the evaluation. In addition, the rule instantiation procedure of DLV implements a *BackJumping* algorithm [31] which exploits both semantic and structural information about the rule for computing efficiently only a relevant subset of its ground instances, avoiding the generation of "useless" instances, while fully preserving the semantics 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 [9, 30] which allow for three levels of parallelism during the instantiation process, namely, components, rules and single rule level.

Model Generation Phase. One of the main optimizations used in the model generation phase concerns the propagation step, where an advanced pruning operator [8, 16] is applied that allows to prune the search space by combining an extension of the well-founded operator for disjunctive programs.

The efficiency of the whole model generation process depends also on two crucial features: a good heuristic (branching rule) to choose the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation); and a smart recovery procedure for undoing the choices

causing inconsistencies. To this end, both *look-ahead* [18] and *look-back* [34, 28] techniques and heuristics have been implemented in DLV. In a lookahead heuristic [18] each possible choice literal is tentatively assumed, its consequences are computed, and some characteristic values on the result are recorded. The look-ahead heuristics of [18] "layers" several criteria based on peculiar properties of DLV programs, and basically drives the search towards "supported" interpretations (since answer sets are supported interpretations – cf. [27, 29, 3]). In look-back heuristics choices are usually made in such a way that the atoms most involved in conflicts are chosen first. Look-back heuristics are mainly employed in conjunction with *backjumping*, where the set of chosen literals that are relevant for an inconsistency are detected, and the system goes back in the search until at least one choice that caused the inconsistency is undone. The *backjumping* technique of DLV makes use of a *reason calculus* [34] that allows for determining the relevance for an inconsistency; in particular, the information about the choices ("reasons") whose truth-values have caused truth-values of other deterministically derived atoms is collected and used for backjumping.

Model Checking Phase. A crucial step in the computation of the answer sets is model checking. There are two main reason for the importance of the model checking step: the exponential number of possible models (model candidates), and the hardness of stable model checking. Note that, when disjunction is allowed in the head, deciding whether a given model is a stable model of a propositional ASP program is co-NP complete in general [12]. For this phase DLV adopts a technique based on a transformation $\mathcal{T}$, which reduces stable model checking to UNSAT, i.e., to deciding whether a given CNF formula is unsatisfiable. Thus, the stability of a candidate answer set M for a program P is verified by calling a SAT solver on the CNF formula obtained by applying $\mathcal{T}$ to P. The transformation consumes logarithmic space and no new symbols are added.

4.2 DLV Architecture

The system architecture of DLV is shown in Figure 1. Upon startup, the input specified by the user is parsed and transformed into the internal data structures of the system. The input can be read from text files, but, as already mentioned, DLV also provides an interface to relational databases via ODBC. The *Intelligent Grounder* (IG) module efficiently generates a ground instantiation $Ground(\mathcal{P})$ of the input, using techniques described in Section 4.1. Note that for stratified programs the IG module already computes the single answer set and does not produce any instantiation. The subsequent computations, which constitute the non-deterministic part of the DLV system, are then performed on $Ground(\mathcal{P})$ by the *Model Generator* and the *Model Checker* as outlined in Section 4.1.

Once an answer set has been found, the Model Generator may resume in order to look for further answer sets. This process is continued until either no more answer sets exist or an explicitly specified number of answer sets has been computed.

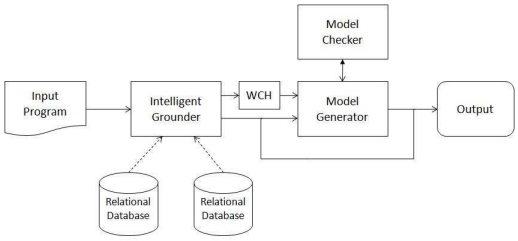

Fig. 1. General architecture of the DLV system.

Note that, in presence of weak constraints, after the instantiation of the program, the computation is governed by the WCH module and consists of two phases: (i) the first phase determines the cost of an optimal answer set together with one "witnessing" optimal answer set and, (ii) the second phase computes all answer sets having that optimal cost. It is worthwhile noting that both the IG and the MG also have built-in support for weak constraints, which is activated (and therefore incurs higher computational cost) only if weak constraints are present in the input. The MC, instead, does not need to provide any support for weak constraints, since these do not affect answer-set checking at all.

5 Reasoning on Top of Databases: DLV^{DB}

In real world applications, reasoning is often done on existing data sources; in these contexts, current deductive database systems show some limitations, namely: *(i)* the amount of data that can be handled is limited since most of them work in main memory; *(ii)* the interaction with external (and autonomous) sources of data, like databases, is not trivial and, in several cases, not allowed at all; *(iii)* the efficiency of existing solutions is still not sufficient for their utilization in complex reasoning tasks involving massive amount of data.

DLV^{DB} comes as a database oriented extension of DLV aiming to overcome these drawbacks. As it will be clear in the following, this extension is significantly more complex than the simple *#import*/*#export* commands introduced previously. In this section we provide a brief description of its main characteristics, inspiring ideas, and possible applications.

5.1 Main Features

The language supported by DLV^{DB} consists of disjunctive and unstratified programs, with aggregates and strong constraints; moreover, it provides the possibility to introduce DBMS-stored function calls directly in the programs as external built-ins. Weak constraints and complex terms are not supported yet.

The basic idea underlying DLV^{DB} is the translation of the input DLV program into a query plan composed of standard SQL queries. The evaluation strategy adopted by the system puts its basis on the sharp distinction existing be-

tween the grounding and the model generation phases. Two distinct strategies are adopted in case the input program is non-disjunctive and stratified or not.

If a program is non-disjunctive and stratified, it has a unique stable model corresponding exactly to its ground instantiation. The evaluation of these programs can be done by translating each rule into a corresponding SQL statement, and in the composition of a suitable query plan on the DBMS; the evaluation of recursive rules is carried out with an improved semi-naïve approach.

In presence of disjunctive rules or unstratified negation in a program, its ground instantiation is no more sufficient to compute its stable models. Then, grounding and model generation phases must both take place. The evaluation strategy, in this case, moves most of the grounding into the database, by the execution of suitable SQL queries. This phase generates two kinds of data: ground atoms (facts) valid in every stable model (and thus not requiring further elaboration in the model generation phase) and ground rules, summarizing possible values for a predicate and the conditions under which these can be inferred.

Facts compose the so called *solved* part of the program, whereas ground rules form the *residual program*. One of the main challenges in DLV^{DB} is to keep the smallest amount of information as possible in main memory; consequently, the residual program generated by the system is as small as possible.

The minimal residual program is then loaded into the main memory, and the model generation is carried out with the standard DLV techniques, described previously.

DLV^{DB} also ports DLV built-in predicates to databases, and extends this functionality to any stored function defined in the database (in the following, we call them external built-ins). The evaluation of such external built-ins is completely carried out during the grounding (this is true even for disjunctive or unstratified programs). As a consequence, their handling can be carried out completely within the SQL statements generated for the query plan. By convention, given an external built-in $\#f(X_1, \ldots, X_n, O)$ only the last variable O can be considered as an output parameter, whereas all the other variables must be intended as input for f and, thus, they must be safely bound to some other variables in the rule body. This corresponds to the database function call $f(X_1, \ldots, X_n) = O$. For example, consider the rule:

$$mergedNames(ID, Name) :- person(ID, FirstName, LastName), \\ \#concat(FirstName, LastName, Name).$$

This rule is translated into:

```
INSERT INTO mergedNames (SELECT  person.ID,
concat(person.FirstName,person.LastName) FROM  person);
```

In order to allow for a strict coupling between DLV and DBMSs, a set of auxiliary directives has been designed so as to instruct DLV^{DB} on how to map intended input/output data onto DLV predicates; details on this aspect are given in the next section.

As for current and future work, we plan to add the following features to the system: *(i)* support for complex terms, *(ii)* introduction of techniques for the

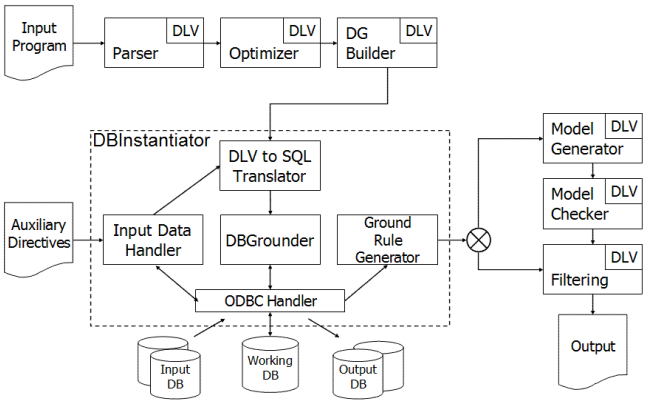

Fig. 2. Architecture of DLV^{DB}.

distribution of the evaluation on multiple databases, and *(iii)* introduction of techniques for improving query answering like unfolding and static filtering.

5.2 DLV DB Architecture

Figure 2 illustrates the architecture of DLV^{DB}. In the figure, the boxes marked with DLV have already been developed in the DLV system. An input program $\mathcal{P}$ is first analyzed by the Parser which encodes the rules in the intensional database (IDB) and stores in the working database facts specified directly in the input program (if any). Then the Optimizer applies basic syntactic rewritings and the Dependency Graph Builder computes the dependency graph of the program, its connected components and a topological ordering of these components. Finally, the DB Instantiator module, the core of the system, is activated.

The DB Instantiator module receives the Dependency Graph (DG) generated by the Dependency Graph Builder and some auxiliary directives. Communication with databases is performed via ODBC. This allows DLV^{DB} both to be independent from a particular DBMS and to handle databases distributed over the Internet. Only strictly necessary information is transferred from the databases to the system in order to limit the inherent inefficiency of these operations.

If the input program is non-disjunctive and stratified, the result of the instantiation step is directly fetched to the filtering module; otherwise the Ground Rule Generator module produces the residual program. This is transferred in main memory to the standard DLV Model Generator for the identification of the stable models. Note that all the data derived to be true in every stable model by the DB Instantiator are kept inside the database.

As previously pointed out, DLV^{DB} can be coupled with external databases through some auxiliary directives. Intuitively, the user must specify the working database and can specify a set of table definitions; each specified table must

be mapped onto one of the program predicates. Facts can reside on separate databases or they can be obtained as views on different tables. A `USE` or `CREATE` directive can be used to specify input or output data, respectively. Finally, the user can choose to copy the entire output of the evaluation or parts thereof in a database different from the working one.

5.3 Using DLV ^{DB} for Data Integration

Data integration systems provide a transparent access to different and possibly distributed sources. The user is provided with a uniform view of available information by the so-called *global schema*, which queries can be posed upon. The integration system is then in charge of accessing the single sources separately and merging data relevant for the query, guided by mapping rules that specify relationships holding between the sources and the global schema [2, 23].

The global schema may contain integrity constraints (such as key dependencies, inclusion dependencies, etc.). The main issues in data integration arise when original sources independently satisfy the integrity constraints but, when they are merged through the mappings, they become inconsistent. As an example, consider students of two universities; each student has an unique ID in his university, but two different students in different universities may share the same ID. Clearly, when they are combined in a global database, the key constraint on student IDs of the global schema will be violated.

Most of the solutions to these problems are based on database repair approaches. Basically, a repair is a new database satisfying constraints of the global schema with minimal differences from the source data. Note that multiple repairs can exist for the same database. Then, answering queries over globally inconsistent sources consists in computing those answers that are true in every possible repair; these are called *consistent* answers in the literature.

Answer Set Programming is a powerful tool in this context, as demonstrated for example by the approaches formalized in [2, 5, 24]. In fact, if mappings and constraints on the global schema are expressed as disjunctive datalog programs, and the query Q as a union of conjunctions on the global schema, the database repairs correspond to the stable models of the program, and the consistent answers to Q correspond to the answers of Q under cautious reasoning.

In this context, DLV^{DB} provides: *(i)* the needed expressiveness to build multiple repairs and to perform cautious reasoning on them (not provided by standard SQL), *(ii)* the capability to deal with the massive amounts of data typical of real world data integration scenarios (not provided by available deductive systems), and *(iii)* an easy way to interact with autonomous and distributed databases, a frequent setting in data integration processes.

Example 1. To have an intuition on the simplicity to use DLV^{DB} as a data integration engine, consider two student relations $s1(SID, Name)$ and $s2(SID, Name)$ of two different universities, and assume that the global schema is designed so as to merge these lists, but keeping SID as a key for the global database.

A program defining the mappings for the global relation sG and handling the possible repairs for key constraint violations over student IDs is:

$$sR(ID, N) \text{ :- } s1(ID, N). \quad sR(ID, N) \text{ :- } s2(ID, N).$$
$$sC(ID, N1) \vee sC(ID, N2) \text{ :- } sR(ID, N1), sR(ID, N2), N1 \neq N2.$$
$$sG(ID, N) \text{ :- } sR(ID, N), \text{not } sC(ID, N).$$

Here the first two rules load all possible data from the sources, whereas the third one avoids to put conflicting tuples in the global relation sG. Note that the disjunctive rule allows the generation of the minimal repairs by singling out conflicting tuples only.

Now, assume that $s1$ contains $\{s1(1234, John),\ s1(2345, Andrew)\}$ and $s2$ contains $\{s2(1234, David)\}$. There is globally a conflict between John and David because they have the same ID. Then, there are two repairs for sG, namely $\{sG(1234, John),\ sG(2345, Andrew)\}$ and $\{sG(1234, David),\ sG(2345, Andrew)\}$.

If the user poses the query $q1(N) \text{ :- } sG(ID, N)$, the only consistent answer is: $Andrew$, but if the user asks for $q2(ID) \text{ :- } sG(ID, N)$, the consistent answers are: $\{1234, 2345\}$.

Finally, if the actual content of s1 and s2 is stored in two database tables s1r on database DB1 and s2r on database DB2, in order to perform the query evaluation on a database named workdb, the following auxiliary directives are sufficient:

USEDB workdb:myname:mypasswd.
USE s1r MAPTO s1 FROM DB1:u1:pw1. USE s2r MAPTO s2 FROM DB2:u2:pw2.

6 Spin-Off and Applications

DLV is widely used by researchers all over the world, and, importantly, it has stimulated quite some interest also in industry. Indeed, even if the industrial exploitation of DLV has started fairly recently, it already has a history of applications on the industrial level.

The industrial application of DLV is mostly managed by two spin-off companies of the University of Calabria, EXEURA s.r.l. and DLVSYSTEM s.r.l. . EXEURA develops products and applications in the area of knowledge management based on DLV; while DLVSYSTEM maintains the DLV system and provides consulting on its use.

In this section we present some of the industrial applications of DLV. In particular, we first mention some industrial products of EXEURA incorporating DLV as computational core. Then, we recall a number of industrial applications based on DLV or on DLV-based products.

DLV-based Industrial Products. OntoDLV [35, 36], OLEX [11, 39], HiLeX [38, 37], are three Knowledge Management products of EXEURA based on DLV.

OntoDLV [35, 36] is a system for ontology specification and reasoning. The language of OntoDLV, called OntoDLP, is an extension of (disjunctive) ASP with all the main ontology constructs including classes, inheritance, relations,

and axioms. Importantly, OntoDLV supports a powerful interoperability mechanism with OWL, allowing the user to retrieve information from external OWL Ontologies and to exploit this data in OntoDLP ontologies and queries. OntoDLV facilitates the development of complex applications in a user-friendly visual environment; it features a rich Application Programming Interface (API) [19], and it is endowed with a robust persistency-layer for saving information transparently on a DBMS, and it seamlessly integrates DLV [26].

OLEX [11, 39] (OntoLog Enterprise Categorizer System) is a corporate classification system supporting the entire content classification life-cycle, including document storage and organization, ontology construction, pre-processing and classification. OLEX employs a reasoning-based approach to text classification which combines: (*i*) ontologies for the formal representation of the domain knowledge; (*ii*) pre-processing technologies for a symbolic representation of texts and (*iii*) ASP as categorization rule language and DLV as ASP engine. Logic rules, indeed, provides a natural and powerful way to encode how document contents may relate to ontology concepts.

HiLεX [38, 37] is an advanced system for ontology-based information extraction from semi-structured and unstructured documents. HiLεX implements a semantic approach to the information extraction problem able to deal with different document formats (html, pdf, doc, ...). HiLεX is based on OntoDLP for describing ontologies, and supports a language that is founded on the concept of *ontology descriptor*. A "descriptor" looks like a production rule in a formal attribute grammar, where syntactic items are replaced by ontology elements. The obtained specification is rewritten in ASP and evaluated by means of the DLV system.

Industrial Applications. Commercial applications based on DLV include:

Team Building in the Gioia-Tauro Seaport. A system based on DLV has been developed to automatically produce an optimal allocation of the available personnel of the international seaport of Gioia Tauro [21]. The system currently employed by the transshipment company ICO BLG can build new teams satisfying a number of constraints or complete the allocation automatically when the roles of some key employees are fixed manually.

E-Tourism. IDUM [22] is an intelligent e-tourism system. IDUM system helps both employees and customers of a travel agency in finding the best possible travel solution in a short time. In IDUM an ontology modeling the tourism scenario was developed by using OntoDLV, and is automatically filled by processing the offers received by a travel agent with HiLεX. IDUM mimics the behavior of the typical employee of a travel agency by running a set of specifically devised logic programs that reason on the information contained in the tourism ontology.

Automatic Itinerary Search. In this application, a Web portal has been conceived for making the public transportation system of the Italian region Calabria more accessible, including both public and private companies. The system specifies locations and time tabling of start/transfers/arrival, as well as other information on the trip, like walking directions, duration, etc. A set of specifically devised ASP programs are used to build the required itineraries.

e-Government. An application of the OLEX system has been developed which classifies legal acts and decrees issued by public authorities. The system was validated with the help of the employees of the Calabrian Region administration, and it performed very well by obtaining a mean precision of 96% on real-world documents.

e-Medicine. OLEX has been used to develop a system capable of automatically classifying case histories and documents containing clinical diagnoses. The system was commissioned with the goal of conducting epidemiological analyses, by the ULSS n.8 (which is, a local authority for health services) of the area of Asolo, in the Italian region Veneto. The system has been deployed and is currently used by the personnel of the ULSS of Asolo.

References

1. Alviano, M., Faber, W., Greco, G., Leone, N.: Magic sets for disjunctive datalog programs. Tech. Rep. 09/2009, Dipartimento di Matematica, Università della Calabria, Italy (2009), http://www.wfaber.com/research/papers/TRMAT092009.pdf
2. Arenas, M., Bertossi, L.E., Chomicki, J.: Specifying and Querying Database Repairs using Logic Programs with Exceptions. In: Larsen, H.L., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H. (eds.) Proceedings of the Fourth International Conference on Flexible Query Answering Systems (FQAS 2000) (2000)
3. Baral, C., Gelfond, M.: Logic Programming and Knowledge Representation. Journal of Logic Programming 19/20, 73–148 (1994)
4. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE Transactions on Knowledge and Data Engineering 12(5), 845–860 (2000)
5. Calì, A., Lembo, D., Rosati, R.: Query rewriting and answering under constraints in data integration systems. In: Int. Joint Conference on Artificial Intelligence (IJCAI'03). pp. 16–21 (2003)
6. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Annals of Mathematics and Artificial Intelligence 50(3–4), 333–361 (2007)
7. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: Proceedings of the 24th International Conference on Logic Programming (ICLP 2008). Lecture Notes in Computer Science, vol. 5366, pp. 407–424. Springer, Udine, Italy (Dec 2008)
8. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Disjunctive Logic Programming Systems. Fundamenta Informaticae 71(2–3), 183–214 (2006)
9. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. Journal of Algorithms in Cognition, Informatics and Logics 63(1–3), 34–54 (2008)
10. Cumbo, C., Faber, W., Greco, G., Leone, N.: Enhancing the magic-set method for disjunctive datalog programs. In: Proceedings of the the 20th International Conference on Logic Programming – ICLP'04. Lecture Notes in Computer Science, vol. 3132, pp. 371–385 (2004)
11. Cumbo, C., Iiritano, S., Rullo, P.: OLEX - A Reasoning-Based Text Classifier. In: Proceedings of Logics in Artificial Intelligence, 9th European Conference, (JELIA 2004), September 27-30, 2004. Lecture Notes in Computer Science, vol. 3229, pp. 722–725. Lisbon, Portugal (2004)

12. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys 33(3), 374–425 (2001)
13. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Commun. ACM 5, 394–397 (1962)
14. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: International Joint Conference on Artificial Intelligence (IJCAI) 2005. pp. 90–96. Edinburgh, UK (Aug 2005)
15. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In: INAP Organizing Committee (ed.) Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99). pp. 135–139. Prolog Association of Japan (Sep 1999)
16. Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99). Lecture Notes in AI (LNAI), vol. 1730, pp. 177–191. Springer Verlag, El Paso, Texas, USA (Dec 1999)
17. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence (2010), accepted for publication
18. Faber, W., Leone, N., Pfeifer, G., Ricca, F.: On look-ahead heuristics in disjunctive logic programming. Annals of Mathematics and Artificial Intelligence 51(2–4), 229–266 (2007)
19. Gallucci, L., Ricca, F.: Visual Querying and Application Programming Interface for an ASP-based Ontology Language. In: Vos, M.D., Schaub, T. (eds.) Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). pp. 56–70 (2007)
20. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
21. 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: Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 5937, pp. 40–42. Springer (2010)
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.) Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009). Lecture Notes in Computer Science, vol. 5753, pp. 368–381. Springer (2009)
23. Lenzerini, M.: Data integration: A theoretical perspective. In: Proc. PODS 2002. pp. 233–246 (2002)
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: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005). pp. 915–917. ACM Press, Baltimore, Maryland, USA (Jun 2005)
25. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria. Lecture Notes in AI (LNAI), vol. 2173, pp. 280–294. Springer Verlag (Sep 2001)

26. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (Jul 2006)
27. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation 135(2), 69–112 (Jun 1997)
28. Maratea, M., Ricca, F., Faber, W., Leone, N.: Look-back techniques and heuristics in dlv: Implementation, evaluation and comparison to qbf solvers. Journal of Algorithms in Cognition, Informatics and Logics 63(1–3), 70–89 (2008)
29. Marek, V.W., Subrahmanian, V.: The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In: Proceedings of the 6th International Conference on Logic Programming – ICLP'89. pp. 600–617. MIT Press (1989)
30. Perri, S., Ricca, F., Sirianni, M.: A parallel asp instantiator based on dlv. In: Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010. pp. 73–82. ACM (2010)
31. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. Annals of Mathematics and Artificial Intelligence 51(2–4), 195–228 (2007)
32. Radziszowski, S.P.: Small Ramsey Numbers. The Electronic Journal of Combinatorics 1 (1994), revision 9: July 15, 2002
33. Ricca, F.: The DLV Java Wrapper. In: de Vos, M., Provetti, A. (eds.) Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation. pp. 305–316. Messina, Italy (Sep 2003), online at http://CEUR-WS.org/Vol-78/
34. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications – The European Journal on Artificial Intelligence 19(2), 155–172 (2006)
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. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV$^+$ System. Journal of Applied Logics 5(3), 545–573 (2007)
37. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: de Vos, M., Provetti, A. (eds.) Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation. pp. 248–262. Bath, UK (Jul 2005)
38. Ruffolo, M., Manna, M.: HiLeX: A System for Semantic Information Extraction from Web Documents. In: Manolopoulos, Y., Filipe, J., Constantopoulos, P., Cordeiro, J. (eds.) ICEIS (Selected Papers). Lecture Notes in Business Information Processing, vol. 3, pp. 194–209 (2008)
39. Rullo, P., Cumbo, C., Policicchio, V.L.: Learning rules with negation for text categorization. In: Cho, Y., Wainwright, R.L., Haddad, H., Shin, S.Y., Koo, Y.W. (eds.) Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15. pp. 409–416. ACM (2007)
40. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. Theory and Practice of Logic Programming 8, 129–165 (2008)